Correctness and Complexity of BMSSP in Isabelle/HOL

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

July 6, 2026

Abstract

This development formalizes the bounded multi-source shortest path (BMSSP) subproblem underlying the deterministic directed single-source shortest path algorithm of Duan, Mao, Mao, Shu, and Yin [2]. It proves three explicit claims. First, the executable `bmssp_distances` computes the exact reachable shortest-distance map for well-formed finite natural weighted digraphs; this executable instantiates the bucketed work-list with block size one, so it is a verified Dijkstra-style instantiation of the bucketed partition, proved directly against the shortest-path semantics rather than by refining the recursive cost relation. Second, a separate costed BMSSP recurrence—a recursive inductive relation, not runnable code—satisfies the $O(m \log^{2/3} n)$ bound under the abstract Insert/BatchPrepend/Pull operation-cost interface; a cost bridge shows the concrete bucketed operation costs fill the recurrence's cost slots at the logarithmic schedule scales, so the bound holds with the bucketed costs rather than free parameters, and a non-conditional runtime theorem charges those bucketed costs inside the recurrence under exactly the standing hypotheses of the abstract headline. Third, the bucketed partition operations realize the primitive costs required by that interface. The runtime locale is shown non-vacuous not only at one fixed graph but across a size-indexed family of directed paths of every size. The formalization does not state a single generated-code machine-step theorem for the executable, and the executable and the asymptotic analysis are separate checked artifacts that share the partition primitives. The entry also exports executable SML and includes a checked worked shortest-path example whose evaluated output is $[(0,0),(1,3),(2,5),(3,8),(4,6)]$.

AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

Contents

1 Background

The single-source shortest path problem asks for the distance from one source vertex to every reachable vertex of a directed graph with non-negative edge weights. Dijkstra's algorithm [1] is the classical starting point: it maintains tentative labels and repeatedly settles a vertex of minimum tentative distance. With Fibonacci heaps, Fredman and Tarjan [3] obtained the well-known $O(m + n \log n)$ implementation for graphs with n vertices and m edges. For sparse directed graphs this bound became the standard deterministic barrier: the logarithmic sorting-like term is paid by the priority-queue discipline that extracts vertices one by one.

Duan, Mao, Mao, Shu, and Yin [2] break this barrier by changing the recursive unit of work. Their algorithm is not simply Dijkstra with a faster heap. The central object is BMSSP, a bounded multi-source shortest path subproblem. A BMSSP call starts from a set of already relevant sources and an upper bound B. It only has to complete vertices whose shortest distances fall below the bound and whose shortest paths pass through the current source set. Recursive calls therefore work on ranges of distances rather than on one global queue of all unsettled vertices.

The formalization follows that structure. At the correctness level, the algorithm is expressed in terms of finite directed graphs, simple walks, shortest-path distances, and BMSSP pre/postconditions. At the cost level, the recursive proof is parameterized by an abstract partition interface with Insert, BatchPrepend, and Pull operations. The runtime theorem is proved once against that interface and the paper's logarithmic schedule. The concrete bucketed data structure is then shown to realize the interface with the critical operation bounds. This cost statement is separate from the functional-correctness theorem for the exported executable.

The essential improvement is the ratio inside the logarithm. A sorted-list or ordinary balanced-tree representation searches among all N live entries and naturally gives a $\log N$ term. The paper's partition structure searches among bucket boundaries, so Insert pays $O(\max(1, \log(N/M)))$, where M is the block-size parameter. With the schedule used by the recursive BMSSP analysis, this distinction is exactly what preserves the $O(m \log^{2/3} n)$ headline.

2 Roadmap of the Formalization

The formal development is organized by proof layer. The table below gives one or two primary entry points for each layer and the checked surface that a reader should look for there; supporting theories appear nearby in the session order.

Layer	Start here	Checked surface
Semantic target	`BMSSP_Correctness`	Finite weighted digraphs, shortest-path distances, BMSSP pre/postconditions, and root-call correctness for ordinary SSSP.
Recursive BMSSP	`BMSSP_Algorithm_Correctness`; `BMSSP_Recursive`	Correctness of FindPivots, base cases, partition-loop traces, recursive calls, and final assembly of completed vertices.
Partition interface	`BMSSP_Partition_Interface`; `BMSSP_Partition_Pull_Bridge`	Abstract Insert, BatchPrepend, and Pull contracts, with the algebraic cost predicates used by the recurrence proof.
Cost recurrence	`BMSSP_Top_Level_Bounds`	The logarithmic parameter schedule and the cost-model theorem `bmssp_runtime_bigo_target`.
Bucketed data structure	`BMSSP_Bucketed_Partition`	Concrete bucketed operations satisfying `bp_insert_cost_bound`, `bp_batch_prepend_cost_bound`, `bp_pull_cost_bound`, and the interface facts `bp_realises_partition_insert_cost_bound`, `bp_realises_partition_batch_cost_bound`, and `bp_realises_partition_pull_cost_bound`.
Executable artifact	`BMSSP_Executable_Headline`	The executable `bmssp_distances`, the checked example `example_bmssp_correct`, and the headline theorem `bmssp_correct_strong`.
Cost bridge	`BMSSP_Bucketed_Cost_Bridge`	Connects the bucketed operation costs to the recurrence cost slots at the logarithmic schedule scales; capstone `bucketed_refined_cost_bigo_sssp_time_target`.
Runtime family	`BMSSP_Path_Family`	A directed-path runtime instance of every size; the bound `path_family_runtime_bigo_target` with edge count `path_family_edge_count` growing without bound.
Non-conditional runtime	`BMSSP_Bucketed_Runtime`	The hypothesis-free `bmssp_bucketed_runtime_bigo_target`: the bucketed operation costs charged inside the recurrence yield $O(m \log^{2/3} n)$, via the decoupled amortised bound `finite_initial_label_decoupled_amortized`.

Several additional theories separate the mathematical layers used below: shortest-path semantics, abstract BMSSP correctness, costed recurrences, partition-interface costs, bucketed primitive operations, executable code, and the final executable wrapper. The cost-model theorems and the executable correctness theorem are deliberately separate claims.

3 Assumptions and Scope

The entry proves three related, but not identical, layers.

First, the semantic BMSSP development starts from finite weighted digraphs, simple walks, shortest-path distances, and a bounded BMSSP pre/postcondition. The recursive correctness proof is organized through FindPivots, partition loop obligations, complete sources, and bounded distance ranges.

Second, the asymptotic development is a cost-model proof. It solves the costed BMSSP recurrence under the abstract Insert/BatchPrepend/Pull interface, uses the logarithmic parameter schedule from the paper, and proves the `bmssp_runtime_bigo_target` theorem in the corresponding runtime locale. This is not stated as a machine-step bound for generated SML.

Third, the executable development proves functional correctness for the exported finite natural-weight graph representation. The public theorem `bmssp_correct_strong` applies to every G satisfying `nat_graph_well_formed` G and every source src $\in$ `nat_graph_vertices` G. It does not assume unique shortest paths. The well-formedness predicate is a representation-side condition: the directed edge list is duplicate-free and the total natural edge weight is below the executable sentinel `bmssp_infinity`.

4 The Headline Theorems

The primary executable correctness statement is

`bmssp_correct_strong` in `BMSSP_Executable_Headline`.

The theorem fixes a finite directed graph G of natural-numbered vertices with non-negative natural edge weights and a source vertex src. The statement is

> For every G with `nat_graph_well_formed` G and every src in `nat_graph_vertices` G:
>
> - the keys of bmssp_distances G src are distinct;
> - the key set is exactly the set of vertices reachable from src;
> - every pair (v, d) returned by bmssp_distances G src satisfies $d =$ nat_graph_dist G src v after coercion to real numbers.

Thus the executable theorem is not conditional on a unique-shortest-path or tie-breaking assumption. Its displayed hypotheses are the finite natural-graph representation condition and the source-membership condition.

The runtime and bucketed-interface statements are separate public claims. `bmssp_runtime_bigo_target` is the cost-model theorem for the abstract BMSSP recurrence and logarithmic parameter schedule. Its closed assumption-free form is stated inside the runtime locale; interpreting that locale supplies the reduced positive instance, reachability, and bounded outdegree hypotheses. To certify that this locale is inhabited rather than vacuous, the theory `BMSSP_Runtime_Instance` interprets it on a concrete unit-weight directed path $0 \to 1 \to 2 \to 3$: it discharges unique shortest walks (from the verified simple-walk enumerator), positive weights, full reachability, and bounded outdegree, and thereby obtains `path_runtime_bigo_target`, the same logarithmic running-time bound as a closed, assumption-free statement for an explicit non-trivial graph. The corollaries `bp_realises_partition_insert_cost_bound`, `bp_realises_partition_batch_cost_bound`, and `bp_realises_partition_pull_cost_bound` show that the bucketed Insert, BatchPrepend, and Pull operations realize the abstract primitive-cost predicates used by that recurrence. The more concrete theorems `bp_insert_cost_bound`, `bp_batch_prepend_cost_bound`, and `bp_pull_cost_bound` expose the amortised budgets for the bucketed implementation itself.

The session build checks all of these claims. The executable smoke test `example_bmssp_correct` also executes the generated code equations on a worked five-vertex example with bmssp_distances example_graph $0 = [(0,0),(1,3),(2,5),(3,8),(4,6)]$, proved by `by eval` inside the session build itself.

5 The Bucketed Partition Data Structure

The partition data structure is the main concrete contribution of the executable layer. It stores tentative labels as key/value pairs grouped into buckets. Each bucket contains an unsorted list of at most M entries in the regular state, and the bucket markers form a monotone directory of lower bounds. The directory, rather than the full set of entries, is the object searched by Insert. If N entries have been inserted since the last Pull, there are only on the order of N/M buckets, which gives the $\log(N/M)$ search term.

The formal model separates several invariants. The ordinary invariant states that bucket sizes are within the block-size bound, markers are sorted, markers are lower bounds for their bucket entries, keys are distinct, and the functional value memory agrees with the entries stored in buckets. The ordered invariant additionally states that adjacent bucket boundaries separate

the value ranges. The lazy invariant relaxes the size side condition: a bucket may temporarily grow up to the lazy threshold before a split is forced.

The amortised analysis is deliberately close to the paper's intuition. A lazy Insert searches the bucket directory, inserts into one bucket, and only splits when the bucket has accumulated enough excess entries. The potential function records overflow beyond M together with a small slack term for the number of buckets. The proofs use a scaled credit invariant, so the actual cost of a split is paid by the overflow accumulated by previous insertions. After the split the potential drops, and the amortised Insert cost is the directory-search charge plus a constant amount of update work.

BatchPrepend receives a list of new candidates that belong before the currently exposed range. The concrete operation rebuckets that batch and prepends the resulting buckets in one operation, which is why its accounting is per item rather than a repeated ordinary Insert cost. Pull removes a first bucket when the first-bucket side conditions hold, and its scanned work is bounded by the block size. The theorem names reflect these three conclusions: `bp_insert_cost_bound`, `bp_batch_prepend_cost_bound`, and `bp_pull_cost_bound`.

The session also contains a baseline sorted-list partition model used by the exact recurrence and abstraction proofs. Sorted lists refine the same abstract contracts, but they search on N entries and therefore do not supply the paper's N/M ratio. The bucketed structure is the implementation connected to the active partition-interface realization facts.

6 Executable Export and Worked Example

The final theory gives a concrete executable surface. Vertices and edge weights are natural numbers, graphs are finite edge lists, and the work-list used by the executable loop is the bucketed partition from `BMSSP_Bucketed_Partition`. The entry point `bmssp_distances` computes a finite association list of distances for the example graph.

The executable entry point is a finite natural-weight instantiation using the bucketed work-list. Its functional correctness is proved independently against the shortest-path semantics. The asymptotic result applies to the costed BMSSP model and bucketed operation interface; the two layers share the same partition primitives but are distinct artifacts. The executable layer uses a finite upper sentinel for natural labels; the abstract correctness layer uses the `bound = Fin real | Infinity` datatype.

The example has source 0 and five vertices. The hand-computed distances are:

$$d(0) = 0, \quad d(1) = 3, \quad d(2) = 5, \quad d(3) = 8, \quad d(4) = 6.$$

The Isabelle theorem

`example_bmssp_correct`

states that the executable function returns exactly this association list. Its proof is `by eval`. Thus the bundled Poly/ML evaluator executes the generated code equations during the build; if the executable algorithm or the expected map is wrong, the theory does not check.

The same theory contains a `value` command for human inspection and an `export_code` command that emits SML module `BMSSP`. The session ROOT file includes an export directive so the generated code can be materialized as `generated/BMSSP.ML`.

7 Checked Source

The remainder of this document is the checked Isabelle source. Definitions, theorem statements, proof scripts, and document text are all processed by the same session build.

theory *BMSSP-Correctness*
  imports *Complex-Main*
begin

8 Correctness Interface for BMSSP

This is the entry point of the correctness layer. It deliberately avoids committing to the paper's data structure, cost accounting, perturbation reduction, or executable representation. The purpose here is to state the mathematical contract that every later algorithmic refinement has to meet.

The ambient object is a finite directed graph with non-negative real edge weights and a distinguished source vertex. Distances are not imported from a library shortest-path algorithm. Instead, they are defined directly as minima of finite sets of simple-walk weights. This gives the later proofs a small, transparent semantic target: a label is correct exactly when it agrees with the minimum simple-walk distance from the source.

BMSSP means bounded multi-source shortest path. A recursive BMSSP call is not asked to solve all of single-source shortest paths. It receives a set of sources, a bound, and a current label function. It must complete the vertices whose shortest paths pass through the source set and whose distances lie below the bound returned by the call. This bounded contract is the reason the paper can recurse on distance ranges rather than repeatedly extracting one global minimum vertex.

The definitions below therefore introduce three pieces of vocabulary that recur throughout the development. A bound can be finite or infinite; a bound tree is the set of vertices relevant to a source set below such a bound; and the BMSSP postcondition says that the output labels are complete on exactly that tree. Later theories refine how the tree is assembled: FindPivots identifies useful sources, the partition loop splits the distance range, and

the bucketed data structure realizes the required Insert, BatchPrepend, and Pull operations. The final theorems of this file show that a successful root BMSSP call is already enough to establish ordinary single-source shortest-path correctness.

datatype *bound* = *Fin real* | *Infinity*

fun *below-bound* :: *real* ⇒ *bound* ⇒ *bool* **where**
  below-bound x Infinity ⟷ *True*
| *below-bound x* (*Fin b*) ⟷ $x < b$

fun *bound-le* :: *bound* ⇒ *bound* ⇒ *bool* **where**
  bound-le - *Infinity* ⟷ *True*
| *bound-le Infinity* (*Fin* -) ⟷ *False*
| *bound-le* (*Fin a*) (*Fin b*) ⟷ $a \leq b$

The datatype *bound* is intentionally small. The predicate *below-bound* is strict for finite bounds, matching the paper's range convention, while *bound-le* is the preorder used when a recursive call returns a possibly smaller upper bound. Keeping this type separate from raw reals makes the top-level call with *Infinity* explicit and avoids ad hoc sentinel values in the correctness proof.

locale *finite-weighted-digraph* =
  fixes $V :: 'v\ set$
    and $E :: ('v \times 'v)\ set$
    and $w :: 'v \Rightarrow 'v \Rightarrow real$
    and $s :: 'v$
  assumes *finite-V*: *finite V*
    and *source-in-V*: $s \in V$
    and *edge-in-V*: $(u, v) \in E \Longrightarrow u \in V \wedge v \in V$
    and *nonneg-weight*: $(u, v) \in E \Longrightarrow 0 \leq w\ u\ v$
begin

fun *walk* :: *'v list* ⇒ *bool* **where**
  walk [] ⟷ *False*
| *walk* [*x*] ⟷ $x \in V$
| *walk* (x # y # *xs*) ⟷ $x \in V \wedge (x, y) \in E \wedge$ *walk* (y # *xs*)

fun *walk-weight* :: *'v list* ⇒ *real* **where**
  walk-weight [] = 0
| *walk-weight* [*x*] = 0
| *walk-weight* (x # y # *xs*) = $w\ x\ y$ + *walk-weight* (y # *xs*)

definition *walk-betw* :: *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
  walk-betw a p b ⟷ $p \neq [] \wedge$ *hd* $p = a \wedge$ *last* $p = b \wedge$ *walk p*

definition *simple-walk-betw* :: *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
  simple-walk-betw a p b ⟷ *walk-betw a p b* ∧ *distinct p*

definition *reachable* :: $'v \Rightarrow 'v \Rightarrow bool$ **where**
reachable a b $\longleftrightarrow$ $(\exists\, p.$ *simple-walk-betw a p b*$)$

definition *simple-walk-weights* :: $'v \Rightarrow 'v \Rightarrow real\ set$ **where**
simple-walk-weights a b = *walk-weight* ‘ $\{p.$ *simple-walk-betw a p b*$\}$

definition *dist* :: $'v \Rightarrow 'v \Rightarrow real$ **where**
dist a b = *Min* (*simple-walk-weights a b*)

definition *shortest-walk* :: $'v \Rightarrow 'v\ list \Rightarrow 'v \Rightarrow bool$ **where**
shortest-walk a p b $\longleftrightarrow$ *simple-walk-betw a p b* $\wedge$ *walk-weight p* = *dist a b*

definition *through* :: $'v\ set \Rightarrow 'v \Rightarrow bool$ **where**
through S v $\longleftrightarrow$ $(\exists\, u{\in}S.\ \exists\, p.$ *shortest-walk s p v* $\wedge$ *u* $\in$ *set p*$)$

definition *bound-tree* :: $'v\ set \Rightarrow bound \Rightarrow 'v\ set$ **where**
bound-tree S B =
$\{v \in V.$ *reachable s v* $\wedge$ *through S v* $\wedge$ *below-bound* (*dist s v*) *B*$\}$

definition *complete-on* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bool$ **where**
complete-on d U $\longleftrightarrow$ $(\forall\, v{\in}U.$ *reachable s v* $\longrightarrow$ *d v* = *dist s v*$)$

definition *sssp-correct* :: $('v \Rightarrow real) \Rightarrow bool$ **where**
sssp-correct d $\longleftrightarrow$ $(\forall\, v{\in}V.$ *reachable s v* $\longrightarrow$ *d v* = *dist s v*$)$

definition *bmssp-pre* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow bool$ **where**
bmssp-pre d S B $\longleftrightarrow$
S $\subseteq$ *V* $\wedge$
$(\forall\, v{\in}V.$ *reachable s v* $\longrightarrow$ *below-bound* (*dist s v*) *B* $\longrightarrow$
d v $\neq$ *dist s v* $\longrightarrow$ *through S v*$)$

definition *bmssp-post* ::
$('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow bound \Rightarrow 'v\ set \Rightarrow bool$ **where**
bmssp-post d S B d′ B′ U $\longleftrightarrow$
bound-le B′ B $\wedge$ *U* = *bound-tree S B′* $\wedge$ *complete-on d′ U*

The graph locale makes the semantic model finite at the point where finiteness is needed. A walk is a list of vertices following edges; a simple walk is a walk with no repeated vertices; and *local.dist* is the minimum weight over simple walks. Because all edge weights are non-negative, restricting to simple walks preserves shortest distances while making the set minimized by *local.dist* finite.

The central BMSSP contract is expressed by *bmssp-pre* and *bmssp-post*. The precondition says that every not-yet-complete reachable vertex below the input bound must be reachable through the current source set, as formalized by *through*. The postcondition returns a new bound, an output set, and an updated label function; it demands that the output set is exactly *bound-tree* for the returned bound and that labels are complete there via *complete-*

on. This is intentionally weaker than global SSSP correctness, because the recursive algorithm only solves the range assigned to the current call.

lemma *walk-set-subset*:
  assumes *walk p*
  shows *set p* $\subseteq$ *V*
  ⟨*proof*⟩

lemma *simple-walk-length-le-card*:
  assumes *simple-walk-betw a p b*
  shows *length p* $\leq$ *card V*
⟨*proof*⟩

lemma *finite-simple-walks*:
  shows *finite* {*p*. *simple-walk-betw a p b*}
⟨*proof*⟩

lemma *finite-simple-walk-weights*:
  shows *finite* (*simple-walk-weights a b*)
  ⟨*proof*⟩

lemma *simple-walk-weights-nonempty*:
  assumes *reachable a b*
  shows *simple-walk-weights a b* $\neq$ {}
  ⟨*proof*⟩

lemma *dist-is-simple-walk-weight*:
  assumes *reachable a b*
  shows *dist a b* $\in$ *simple-walk-weights a b*
  ⟨*proof*⟩

lemma *shortest-walk-exists*:
  assumes *reachable a b*
  obtains *p* **where** *shortest-walk a p b*
⟨*proof*⟩

lemma *shortest-walk-hd*:
  assumes *shortest-walk a p b*
  shows *p* $\neq$ [] *hd p* = *a last p* = *b*
  ⟨*proof*⟩

lemma *reachable-refl*:
  assumes *v* $\in$ *V*
  shows *reachable v v*
  ⟨*proof*⟩

lemma *reachable-source-through*:
  assumes *reachable s v*
  shows *through* {*s*} *v*
⟨*proof*⟩

lemma *top-bmssp-pre*:
assumes $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
shows *bmssp-pre d* $\{s\}$ *Infinity*
⟨*proof*⟩

lemma *bound-tree-source-infinity*:
assumes $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
shows *bound-tree* $\{s\}$ *Infinity* $= V$
⟨*proof*⟩

lemma *bound-tree-source-infinity-reachable*:
shows *bound-tree* $\{s\}$ *Infinity* $= \{v \in V.$ *reachable s v*$\}$
⟨*proof*⟩

lemma *bmssp-post-complete-bound-tree*:
assumes *bmssp-post d S B d′ B′ U*
and $v \in$ *bound-tree S B′*
shows $d'\ v =$ *dist s v*
⟨*proof*⟩

lemma *bmssp-success-completes-requested-tree*:
assumes *bmssp-post d S B d′ B U*
and $v \in$ *bound-tree S B*
shows $d'\ v =$ *dist s v*
⟨*proof*⟩

lemma *sssp-correctI*:
assumes $\bigwedge v.\ [\![v \in V;$ *reachable s v*$]\!] \Longrightarrow d\ v =$ *dist s v*
shows *sssp-correct d*
⟨*proof*⟩

The remaining lemmas discharge the semantic obligations introduced above. Finiteness of simple walks justifies the use of *Min*; reachability of the source shows that the root source set covers every reachable vertex; and $[\![$*bmssp-post ?d ?S ?B ?d′ ?B′ ?U*; *?v* $\in$ *bound-tree ?S ?B′*$]\!] \Longrightarrow$ *?d′ ?v* $=$ *local.dist s ?v* extracts a pointwise distance equality from the BMSSP postcondition. The final two theorems are the bridge from a root BMSSP result to the ordinary SSSP predicate *sssp-correct*. Later files prove BMSSP postconditions for concrete recursive executions, and then reuse these root theorems unchanged.

theorem *successful-root-bmssp-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *post*: *bmssp-post d* $\{s\}$ *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

theorem *successful-root-bmssp-sssp-correct*:

assumes *post*: *bmssp-post d* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

end

end
theory *BMSSP-Algorithm-Correctness*
imports *BMSSP-Correctness*
begin

9 Abstract BMSSP Correctness

This theory is the correctness skeleton of the BMSSP development. The previous entry-point theory defines distances, bounds, and the basic bounded multi-source postcondition. Here we refine that interface just far enough to express the recursive algorithm described in the paper, while still abstracting from running time and from the concrete partition data structure.

The key strengthening is that sources used by a recursive call are required to be complete sources: their current labels already equal their true shortest-path distances. This matters because BMSSP does not merely carry a set of vertices; it carries a set of vertices whose shortest-path trees may certify the rest of the current bounded range. A vertex that is not solved yet must either be solved directly by the current round or lie on a shortest path through one of those complete sources.

At this level the algorithm is decomposed into three abstract obligations:

- *FindPivots* must either complete a bounded vertex or delegate it to a complete pivot.
- The partition loop driven by *Pull*, recursive BMSSP calls, edge relaxation, and *BatchPrepend* must solve the bounded problem induced by the pivot set.
- The final assembly of recursive-loop vertices with the completed *FindPivots* vertices is then proved correct.

The main theorem of the file, *abstract-bmssp-correct*, says that any implementation satisfying those obligations establishes the full BMSSP postcondition. Later theories instantiate the obligations with a concrete recursive loop, then the bucketed partition structure, and finally the executable example. None of the runtime arguments appear here; this file is the semantic hinge that lets the later cost proofs remain separate from the shortest-path correctness proof.

Two abstractions matter for navigating the rest of the development. First, this layer abstracts from the partition data structure entirely: any implementation satisfying the three obligations above is acceptable, so the later

sorted-list and bucketed-partition refinements both plug in here without disturbing the correctness proof. Second, this layer abstracts from running time entirely: the cost predicates and amortized budgets live one layer further down, in `BMSSP_Top_Level_Bounds.thy` and `BMSSP_Bucketed_Partition.thy`, so that a reader interested only in correctness can stop here.

context *finite-weighted-digraph*
begin

definition *complete-vertices* :: $('v \Rightarrow real) \Rightarrow 'v\ set$ **where**
$complete\text{-}vertices\ d = \{v \in V.\ reachable\ s\ v \wedge d\ v = dist\ s\ v\}$

definition *through-complete* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow 'v \Rightarrow bool$ **where**
$through\text{-}complete\ d\ S\ v \longleftrightarrow through\ (S \cap complete\text{-}vertices\ d)\ v$

definition *bmssp-pre-full* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow bool$ **where**
$bmssp\text{-}pre\text{-}full\ d\ S\ B \longleftrightarrow$
$S \subseteq V \wedge$
$(\forall v{\in}V.\ reachable\ s\ v \longrightarrow below\text{-}bound\ (dist\ s\ v)\ B \longrightarrow$
$d\ v \neq dist\ s\ v \longrightarrow through\text{-}complete\ d\ S\ v)$

definition *bmssp-post-full* ::
$('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow bound \Rightarrow 'v\ set \Rightarrow bool$ **where**
$bmssp\text{-}post\text{-}full\ d\ S\ B\ d'\ B'\ U \longleftrightarrow$
$bound\text{-}le\ B'\ B \wedge U = bound\text{-}tree\ S\ B' \wedge complete\text{-}on\ d'\ U$

definition *find-pivots-post* ::
$('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow 'v\ set \Rightarrow bool$ **where**
$find\text{-}pivots\text{-}post\ d\ S\ B\ d'\ P\ W \longleftrightarrow$
$S \subseteq V \wedge$
$P \subseteq S \wedge$
$W \subseteq bound\text{-}tree\ S\ B \wedge$
$complete\text{-}on\ d'\ W \wedge$
$(\forall v{\in}V.\ reachable\ s\ v \longrightarrow below\text{-}bound\ (dist\ s\ v)\ B \longrightarrow$
$d\ v = dist\ s\ v \longrightarrow d'\ v = dist\ s\ v) \wedge$
$(\forall v{\in}bound\text{-}tree\ S\ B.\ v \in W \vee through\text{-}complete\ d'\ P\ v)$

definition *partition-loop-post* ::
$('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow bound \Rightarrow 'v\ set \Rightarrow bool$ **where**
$partition\text{-}loop\text{-}post\ d\ P\ B\ d'\ B'\ U \longleftrightarrow bmssp\text{-}post\text{-}full\ d\ P\ B\ d'\ B'\ U$

definition *final-bmssp-assembly* ::
$'v\ set \Rightarrow bound \Rightarrow 'v\ set \Rightarrow 'v\ set \Rightarrow 'v\ set \Rightarrow bool$ **where**
$final\text{-}bmssp\text{-}assembly\ S\ B'\ W\ U\text{-}loop\ U \longleftrightarrow$
$U = U\text{-}loop \cup \{v \in W.\ below\text{-}bound\ (dist\ s\ v)\ B'\}$

definition *abstract-bmssp-step* ::
$('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow bound \Rightarrow 'v\ set \Rightarrow bool$ **where**
$abstract\text{-}bmssp\text{-}step\ d\ S\ B\ d'\ B'\ U \longleftrightarrow$

(∃ *d-fp P W U-loop*.
find-pivots-post d S B d-fp P W ∧
partition-loop-post d-fp P B d′ B′ U-loop ∧
complete-on d′ W ∧
final-bmssp-assembly S B′ W U-loop U)

The definitions above separate the proof into named contracts. The set *complete-vertices* records exactly the vertices solved by a label function, and *through-complete* refines the earlier *through*-predicate by requiring the intermediary source to be solved. The strengthened precondition *bmssp-pre-full* is therefore the recursive invariant: every unresolved bounded vertex is certified by a shortest path through a complete source.

The predicate *find-pivots-post* is intentionally asymmetric. It returns a new label function, a pivot subset, and a set of vertices completed directly by the pivot search. Vertices in the current bounded tree are then either in that completed set or are delegated to the pivot subset via *through-complete*. The partition loop contract *partition-loop-post* is just the full BMSSP postcondition specialized to those pivots. Finally, *final-bmssp-assembly* describes the set union that reconstructs the caller's solved range from the recursive-loop output and the vertices completed by FindPivots.

This decomposition is useful because each later theory can prove one small interface. The algorithmic theory proves an instance of *abstract-bmssp-step*; the data-structure theories prove that the partition operations realize the abstract loop assumptions; the top-level theory only needs the postcondition produced here.

lemma *bmssp-post-full-imp-post*:
assumes *bmssp-post-full d S B d′ B′ U*
shows *bmssp-post d S B d′ B′ U*
⟨*proof*⟩

lemma *top-bmssp-pre-full*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *source-complete*: *d s* = *dist s s*
shows *bmssp-pre-full d* {*s*} *Infinity*
⟨*proof*⟩

lemma *top-bmssp-pre-full-reachable*:
assumes *source-complete*: *d s* = *dist s s*
shows *bmssp-pre-full d* {*s*} *Infinity*
⟨*proof*⟩

lemma *through-complete-imp-through*:
assumes *through-complete d S v*
shows *through S v*
⟨*proof*⟩

lemma *bound-tree-mono*:

assumes $S \subseteq T$
shows *bound-tree* $S\ B \subseteq$ *bound-tree* $T\ B$
⟨*proof*⟩

lemma *bound-tree-bound-mono*:
assumes *bound-le* $B'\ B$
shows *bound-tree* $S\ B' \subseteq$ *bound-tree* $S\ B$
⟨*proof*⟩

lemma *complete-on-subset*:
assumes *complete-on* $d\ T\ S \subseteq T$
shows *complete-on* $d\ S$
⟨*proof*⟩

lemma *source-in-own-bound-tree*:
assumes *xS*: $x \in S$
and *reach-x*: *reachable* $s\ x$
and *below-x*: *below-bound* $(dist\ s\ x)\ B$
shows $x \in$ *bound-tree* $S\ B$
⟨*proof*⟩

lemma *sources-subset-bound-tree*:
assumes *reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(dist\ s\ x)\ B$
shows $S \subseteq$ *bound-tree* $S\ B$
⟨*proof*⟩

lemma *sources-subset-bound-tree-if-label-below*:
assumes *complete*: *complete-on* $d\ S$
and *reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)\ B$
shows $S \subseteq$ *bound-tree* $S\ B$
⟨*proof*⟩

lemma *bmssp-post-full-source-card-le-if-label-below-output*:
assumes *post*: *bmssp-post-full* $d\ S\ B\ d'\ B'\ U$
and *complete*: *complete-on* $d\ S$
and *reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)\ B'$
shows *card* $S \leq$ *card* U
⟨*proof*⟩

The first group of lemmas is structural. It relates the strengthened complete-source invariant to the base BMSSP vocabulary and proves the monotonicity facts needed when the recursive call returns a smaller bound. The source-subset lemmas are small but important: they show that a complete source whose label is below the returned bound belongs to the returned *bound-tree*. That is the formal reason the recursive call cannot discard the pivots it was asked to solve.

lemma *find-pivots-establishes-pivot-pre*:
  assumes *pre*: *bmssp-pre-full d S B*
    and *fp*: *find-pivots-post d S B d′ P W*
  shows *bmssp-pre-full d′ P B*
⟨*proof*⟩

lemma *conservative-find-pivots-post*:
  assumes *pre*: *bmssp-pre-full d S B*
  defines $W \equiv \{v \in$ *bound-tree S B*. *d v* = *dist s v*$\}$
  shows *find-pivots-post d S B d S W*
⟨*proof*⟩

The next two lemmas explain the handoff from FindPivots to recursion. The theorem ⟦*bmssp-pre-full ?d ?S ?B*; *find-pivots-post ?d ?S ?B ?d′ ?P ?W*⟧ ⟹ *bmssp-pre-full ?d′ ?P ?B* is the crucial invariant transfer: after FindPivots, the pivot set satisfies *bmssp-pre-full* under the updated labels. If a vertex was already outside the old source tree, the preservation clause of *find-pivots-post* keeps its label correct; if it was inside the tree and not directly completed, the cover clause delegates it through a complete pivot.

The conservative postcondition is a sanity instance of the same interface. It says the abstract FindPivots contract is not empty: keeping the same sources and naming the already-complete part of the tree is sufficient. The executable development uses stronger concrete pivot selection, but this lemma is helpful for understanding the shape of the contract independently of any particular implementation.

lemma *final-assembly-bound-tree*:
  assumes *fp*: *find-pivots-post d S B d-fp P W*
    and *loop*: *partition-loop-post d-fp P B d′ B′ U-loop*
    and *asm*: *final-bmssp-assembly S B′ W U-loop U*
  shows *U* = *bound-tree S B′*
⟨*proof*⟩

lemma *final-assembly-complete*:
  assumes *fp*: *find-pivots-post d S B d-fp P W*
    and *loop*: *partition-loop-post d-fp P B d′ B′ U-loop*
    and *compW*: *complete-on d′ W*
    and *asm*: *final-bmssp-assembly S B′ W U-loop U*
  shows *complete-on d′ U*
⟨*proof*⟩

Final assembly is where the two solved regions are reconciled. The loop has solved *bound-tree* for the pivot set below the returned bound. The FindPivots phase may also have solved vertices in the caller's original tree; only those below the returned bound are retained in the caller's output set. Lemma ⟦*find-pivots-post ?d ?S ?B ?d-fp ?P ?W*; *partition-loop-post ?d-fp ?P ?B ?d′ ?B′ ?U-loop*; *final-bmssp-assembly ?S ?B′ ?W ?U-loop ?U*⟧ ⟹ *?U* = *bound-tree ?S ?B′* proves that this union is not merely contained in the

desired set but exactly equals the caller's *bound-tree*. Lemma ⟦*find-pivots-post ?d ?S ?B ?d-fp ?P ?W*; *partition-loop-post ?d-fp ?P ?B ?d′ ?B′ ?U-loop*; *complete-on ?d′ ?W*; *final-bmssp-assembly ?S ?B′ ?W ?U-loop ?U*⟧ ⟹ *complete-on ?d′ ?U* then proves that labels are complete on that exact set.

theorem *abstract-bmssp-correct*:
  assumes *step*: *abstract-bmssp-step d S B d′ B′ U*
  shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

theorem *abstract-bmssp-correct-with-pre*:
  assumes *pre*: *bmssp-pre-full d S B*
    and *step*: *abstract-bmssp-step d S B d′ B′ U*
  shows *bmssp-post-full d S B d′ B′ U*
  ⟨*proof*⟩

lemma *abstract-bmssp-step-exposes-pivot-pre*:
  assumes *pre*: *bmssp-pre-full d S B*
    and *step*: *abstract-bmssp-step d S B d′ B′ U*
  obtains *d-fp P W U-loop* **where**
    find-pivots-post d S B d-fp P W
    bmssp-pre-full d-fp P B
    partition-loop-post d-fp P B d′ B′ U-loop
    complete-on d′ W
    final-bmssp-assembly S B′ W U-loop U
⟨*proof*⟩

corollary *abstract-bmssp-correct-weak*:
  assumes *abstract-bmssp-step d S B d′ B′ U*
  shows *bmssp-post d S B d′ B′ U*
  ⟨*proof*⟩

The theorem *abstract-bmssp-step ?d ?S ?B ?d′ ?B′ ?U* ⟹ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* is the file's main assembly result: from an *abstract-bmssp-step* witness it extracts the FindPivots contract, the recursive partition-loop contract, and the final set assembly, then rebuilds *bmssp-post-full*. The variant ⟦*bmssp-pre-full ?d ?S ?B*; *abstract-bmssp-step ?d ?S ?B ?d′ ?B′ ?U*⟧ ⟹ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* keeps the precondition in its statement for clients whose proof scripts naturally carry it, while ⟦*bmssp-pre-full ?d ?S ?B*; *abstract-bmssp-step ?d ?S ?B ?d′ ?B′ ?U*; ⋀*d-fp P W U-loop*. ⟦*find-pivots-post ?d ?S ?B d-fp P W*; *bmssp-pre-full d-fp P ?B*; *partition-loop-post d-fp P ?B ?d′ ?B′ U-loop*; *complete-on ?d′ W*; *final-bmssp-assembly ?S ?B′ W U-loop ?U*⟧ ⟹ *?thesis*⟧ ⟹ *?thesis* exposes the pivot precondition needed by the recursive implementation.

The final top-level theorems connect this abstract step to ordinary single-source shortest paths. Once the root call uses source set $\{s\}$ and bound *Infinity*, the root correctness lemmas from the entry-point theory turn the BMSSP postcondition into *sssp-correct*.

theorem *abstract-top-level-sssp-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *step*: *abstract-bmssp-step d* $\{s\}$ *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

theorem *abstract-top-level-reachable-sssp-correct*:
assumes *step*: *abstract-bmssp-step d* $\{s\}$ *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

theorem *initialized-abstract-top-level-sssp-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *source-complete*: *d s* = *dist s s*
and *step*: *abstract-bmssp-step d* $\{s\}$ *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

theorem *initialized-abstract-top-level-reachable-sssp-correct*:
assumes *source-complete*: *d s* = *dist s s*
and *step*: *abstract-bmssp-step d* $\{s\}$ *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

end

end
theory *BMSSP-Shortest-Path-Lemmas*
imports *BMSSP-Correctness*
begin

10 Shortest-Path Prefix Facts

The algorithm's tree-based correctness arguments repeatedly use the fact that vertices appearing earlier on a shortest path have no larger true distance. This theory proves that fact from the finite non-negative weighted graph model, without assuming uniqueness of shortest paths.

context *finite-weighted-digraph*
begin

lemma *walk-nonempty*:
assumes *walk p*
shows $p \neq []$
⟨*proof*⟩

lemma *walk-weight-nonneg*:
assumes *walk p*
shows *0* $\leq$ *walk-weight p*

⟨*proof*⟩

lemma *dist-nonneg*:
assumes *reachable a b*
shows *0* ≤ *dist a b*
⟨*proof*⟩

lemma *walk-take-Suc*:
assumes *walk p i* < *length p*
shows *walk* (*take* (*Suc i*) *p*)
⟨*proof*⟩

lemma *walk-weight-take-Suc-le*:
assumes *walk p i* < *length p*
shows *walk-weight* (*take* (*Suc i*) *p*) ≤ *walk-weight p*
⟨*proof*⟩

lemma *simple-walk-prefix*:
assumes *p*: *simple-walk-betw a p b*
and *i*: *i* < *length p*
shows *simple-walk-betw a* (*take* (*Suc i*) *p*) (*p* ! *i*)
⟨*proof*⟩

lemma *dist-le-simple-walk-weight*:
assumes *p*: *simple-walk-betw a p b*
shows *dist a b* ≤ *walk-weight p*
⟨*proof*⟩

lemma *shortest-walk-prefix-dist-le*:
assumes *p*: *shortest-walk a p b*
and *u*: *u* ∈ *set p*
shows *dist a u* ≤ *dist a b*
⟨*proof*⟩

lemma *through-witness-dist-le*:
assumes *through S v*
obtains *u p* **where** *u* ∈ *S shortest-walk s p v u* ∈ *set p dist s u* ≤ *dist s v*
⟨*proof*⟩

lemma *walk-snoc*:
assumes *walk-p*: *walk p*
and *nonempty*: *p* ≠ []
and *last-p*: *last p* = *u*
and *edge*: (*u*, *v*) ∈ *E*
shows *walk* (*p* @ [*v*])
⟨*proof*⟩

lemma *walk-weight-snoc*:
assumes *p* ≠ [] *last p* = *u*

shows *walk-weight* $(p\ @\ [v]) = \mathit{walk\text{-}weight}\ p + w\ u\ v$
⟨*proof*⟩

lemma *walk-append-tl*:
assumes *walk-p*: *walk p*
and *walk-q*: *walk q*
and *p-ne*: $p \neq []$
and *q-ne*: $q \neq []$
and *join*: *last p* = *hd q*
shows *walk* $(p\ @\ tl\ q)$
⟨*proof*⟩

lemma *walk-weight-append-tl*:
assumes *p-ne*: $p \neq []$
and *q-ne*: $q \neq []$
and *join*: *last p* = *hd q*
shows *walk-weight* $(p\ @\ tl\ q) = \mathit{walk\text{-}weight}\ p + \mathit{walk\text{-}weight}\ q$
⟨*proof*⟩

lemma *walk-betw-append-tl*:
assumes *p*: *walk-betw a p b*
and *q*: *walk-betw b q c*
shows *walk-betw a* $(p\ @\ tl\ q)$ *c*
⟨*proof*⟩

lemma *walk-weight-append-tl-betw*:
assumes *p*: *walk-betw a p b*
and *q*: *walk-betw b q c*
shows *walk-weight* $(p\ @\ tl\ q) = \mathit{walk\text{-}weight}\ p + \mathit{walk\text{-}weight}\ q$
⟨*proof*⟩

lemma *walk-suffix-append*:
assumes *walk* $(xs\ @\ ys)$ $ys \neq []$
shows *walk ys*
⟨*proof*⟩

lemma *walk-prefix-append*:
assumes *walk* $(xs\ @\ ys)$ $xs \neq []$
shows *walk xs*
⟨*proof*⟩

lemma *walk-remove-cycle*:
assumes *walk-p*: *walk* $(xs\ @\ y\ \#\ ys\ @\ y\ \#\ zs)$
shows *walk* $(xs\ @\ y\ \#\ zs)$
⟨*proof*⟩

lemma *walk-weight-remove-cycle-le*:
assumes *walk-p*: *walk* $(xs\ @\ y\ \#\ ys\ @\ y\ \#\ zs)$
shows *walk-weight* $(xs\ @\ y\ \#\ zs) \leq \mathit{walk\text{-}weight}\ (xs\ @\ y\ \#\ ys\ @\ y\ \#\ zs)$

⟨*proof*⟩

lemma *walk-betw-remove-cycle*:
assumes *p*: *walk-betw a* (*xs* @ *y* # *ys* @ *y* # *zs*) *b*
shows *walk-betw a* (*xs* @ *y* # *zs*) *b*
⟨*proof*⟩

lemma *walk-betw-to-simple-walk-le-exists*:
assumes *p*: *walk-betw a p b*
shows ∃ *q*. *simple-walk-betw a q b* ∧ *walk-weight q* ≤ *walk-weight p*
⟨*proof*⟩

lemma *walk-betw-to-simple-walk-le*:
assumes *p*: *walk-betw a p b*
obtains *q* **where** *simple-walk-betw a q b walk-weight q* ≤ *walk-weight p*
⟨*proof*⟩

lemma *walk-suffix-from-index*:
assumes *p*: *simple-walk-betw a p b*
and *i*: *i* < *length p*
shows *walk-betw* (*p* ! *i*) (*drop i p*) *b*
⟨*proof*⟩

lemma *take-Suc-append-tl-drop*:
assumes *i* < *length p*
shows *take* (*Suc i*) *p* @ *tl* (*drop i p*) = *p*
⟨*proof*⟩

lemma *shortest-walk-prefix-shortest*:
assumes *sp*: *shortest-walk a p b*
and *i*: *i* < *length p*
shows *shortest-walk a* (*take* (*Suc i*) *p*) (*p* ! *i*)
⟨*proof*⟩

lemma *walk-nth-edge*:
assumes *walk p*
and *Suc i* < *length p*
shows (*p* ! *i*, *p* ! *Suc i*) ∈ *E*
⟨*proof*⟩

lemma *simple-walk-snoc*:
assumes *p*: *simple-walk-betw s p u*
and *edge*: (*u*, *v*) ∈ *E*
and *fresh*: *v* ∉ *set p*
shows *simple-walk-betw s* (*p* @ [*v*]) *v*
⟨*proof*⟩

lemma *dist-triangle-edge*:
assumes *edge*: (*u*, *v*) ∈ *E*

and *reach-u*: *reachable s u*
shows *reachable s v dist s v* $\leq$ *dist s u + w u v*
⟨*proof*⟩

definition *sound-label* **where**
sound-label d $\longleftrightarrow$ $(\forall\, v{\in}V.$ *reachable s v* $\longrightarrow$ *dist s v* $\leq$ *d v*)

definition *relax-edge* **where**
relax-edge d u v = $(\lambda x.$ *if x = v then min* (*d v*) (*d u + w u v*) *else d x*)

definition *tight-edge-step* **where**
tight-edge-step u v $\longleftrightarrow$
(*u*, *v*) $\in$ *E* $\wedge$ *reachable s u* $\wedge$ *dist s v = dist s u + w u v*

lemma *relax-edge-le*:
relax-edge d u v x $\leq$ *d x*
⟨*proof*⟩

lemma *shortest-walk-successively-tight*:
assumes *sp*: *shortest-walk s p v*
shows *successively tight-edge-step p*
⟨*proof*⟩

lemma *relax-edge-sound*:
assumes *sound*: *sound-label d*
and *edge*: (*u*, *v*) $\in$ *E*
and *reach-u*: *reachable s u*
shows *sound-label* (*relax-edge d u v*)
⟨*proof*⟩

lemma *relax-tight-edge-completes*:
assumes *sound*: *sound-label d*
and *complete-u*: *d u = dist s u*
and *tight*: *tight-edge-step u v*
shows *sound-label* (*relax-edge d u v*)
and *relax-edge d u v v = dist s v*
⟨*proof*⟩

fun *relax-path* **where**
relax-path d [] = *d*
| *relax-path d* [*x*] = *d*
| *relax-path d* (*x* # *y* # *xs*) = *relax-path* (*relax-edge d x y*) (*y* # *xs*)

fun *relax-edges* **where**
relax-edges d [] = *d*
| *relax-edges d* ((*u*, *v*) # *es*) = *relax-edges* (*relax-edge d u v*) *es*

fun *path-edges* **where**
path-edges [] = []

| *path-edges* [*x*] = []
| *path-edges* (*x* # *y* # *xs*) = (*x*, *y*) # *path-edges* (*y* # *xs*)

lemma *relax-edges-append*:
  relax-edges d (*xs* @ *ys*) = *relax-edges* (*relax-edges d xs*) *ys*
  ⟨*proof*⟩

lemma *relax-edges-le*:
  relax-edges d es x ≤ *d x*
⟨*proof*⟩

lemma *relax-path-eq-relax-edges*:
  relax-path d p = *relax-edges d* (*path-edges p*)
  ⟨*proof*⟩

lemma *relax-edge-preserves-other*:
  assumes *x* ≠ *v*
  shows *relax-edge d u v x* = *d x*
  ⟨*proof*⟩

lemma *relax-edge-preserves-complete-other*:
  assumes *x* ≠ *v d x* = *dist s x*
  shows *relax-edge d u v x* = *dist s x*
  ⟨*proof*⟩

lemma *relax-edge-preserves-complete-sound*:
  assumes *sound*: *sound-label d*
    and *complete-x*: *d x* = *dist s x*
    and *edge*: (*u*, *v*) ∈ *E*
    and *reach-u*: *reachable s u*
  shows *relax-edge d u v x* = *dist s x*
⟨*proof*⟩

lemma *relax-edges-sound*:
  assumes *sound*: *sound-label d*
    and *edges*: ⋀*u v*. (*u*, *v*) ∈ *set es* ⟹ (*u*, *v*) ∈ *E*
    and *reaches*: ⋀*u v*. (*u*, *v*) ∈ *set es* ⟹ *reachable s u*
  shows *sound-label* (*relax-edges d es*)
  ⟨*proof*⟩

lemma *relax-edges-preserves-complete-if-not-targeted*:
  assumes *complete-x*: *d x* = *dist s x*
    and *not-target*: ⋀*u*. (*u*, *x*) ∉ *set es*
  shows *relax-edges d es x* = *dist s x*
  ⟨*proof*⟩

lemma *relax-edges-preserves-complete-sound*:
  assumes *sound*: *sound-label d*
    and *complete-x*: *d x* = *dist s x*

and *edges*: $\bigwedge u\ v.\ (u,\ v) \in set\ es \Longrightarrow (u,\ v) \in E$
and *reaches*: $\bigwedge u\ v.\ (u,\ v) \in set\ es \Longrightarrow reachable\ s\ u$
shows *relax-edges d es x* = *dist s x*
⟨*proof*⟩

lemma *relax-path-tight-sound-complete*:
assumes *nonempty*: $p \neq []$
and *sound*: *sound-label d*
and *complete-hd*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
shows *sound-label* (*relax-path d p*)
and *relax-path d p* (*last p*) = *dist s* (*last p*)
⟨*proof*⟩

lemma *relax-edges-with-tight-path-prefix-complete*:
assumes *nonempty*: $p \neq []$
and *sound*: *sound-label d*
and *complete-hd*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *es*: *es* = *path-edges p* @ *extra*
and *extra-edges*: $\bigwedge u\ v.\ (u,\ v) \in set\ extra \Longrightarrow (u,\ v) \in E$
and *extra-reaches*: $\bigwedge u\ v.\ (u,\ v) \in set\ extra \Longrightarrow reachable\ s\ u$
shows *sound-label* (*relax-edges d es*)
and *relax-edges d es* (*last p*) = *dist s* (*last p*)
⟨*proof*⟩

end

end
theory *BMSSP-Base-Case*
imports *BMSSP-Algorithm-Correctness BMSSP-Shortest-Path-Lemmas*
begin

11 Concrete Base Case

The algorithm's level-0 BMSSP call is a bounded Dijkstra-style search from a singleton source. It explores the bounded tree below the input bound and stops once the next extracted distance would exceed the first k vertices. Operationally, the paper describes this with a priority queue. For correctness, only one property of that queue matters here: vertices are extracted in nondecreasing true distance order.

This theory turns that observation into a compact semantic base case. Since the graph is finite, the bounded tree can be listed in sorted distance order. If the list has at most k vertices, the base case completes the whole bounded tree and returns the original bound. If the list is longer, the k-th distance becomes the returned finite bound and the completed set is exactly the part of the bounded tree strictly below that bound. The final lemma

packages this as a BMSSP postcondition without assuming any recursive postcondition for the base case itself.

context *finite-weighted-digraph*
begin

definition *min-dist-vertex* :: *'v set* ⇒ *'v* **where**
  min-dist-vertex A = (*SOME x*. *x* ∈ *A* ∧ *dist s x* = *Min* (*dist s* ‘ *A*))

lemma *finite-bound-tree* [*simp*]: *finite* (*bound-tree S B*)
  ⟨*proof*⟩

lemma *min-dist-vertex-prop*:
  assumes *finite A A* ≠ {}
  shows *min-dist-vertex A* ∈ *A* ∧ *dist s* (*min-dist-vertex A*) = *Min* (*dist s* ‘ *A*)
⟨*proof*⟩

lemma *min-dist-vertex-in*:
  assumes *finite A A* ≠ {}
  shows *min-dist-vertex A* ∈ *A*
  ⟨*proof*⟩

lemma *min-dist-vertex-min*:
  assumes *finite A A* ≠ {} *y* ∈ *A*
  shows *dist s* (*min-dist-vertex A*) ≤ *dist s y*
⟨*proof*⟩

lemma *ordered-extraction-exists*:
  assumes *finite A*
  shows ∃ *xs*. *set xs* = *A* ∧ *distinct xs* ∧ *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) *xs*
⟨*proof*⟩

definition *closest-vertices* :: *'v set* ⇒ *'v list* **where**
  closest-vertices A =
    (*SOME xs*. *set xs* = *A* ∧ *distinct xs* ∧ *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) *xs*)

lemma *closest-vertices-properties*:
  assumes *finite A*
  shows *set* (*closest-vertices A*) = *A*
    and *distinct* (*closest-vertices A*)
    and *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) (*closest-vertices A*)
⟨*proof*⟩

definition *base-case-order* :: *'v* ⇒ *bound* ⇒ *'v list* **where**
  base-case-order x B = *closest-vertices* (*bound-tree* {*x*} *B*)

definition *base-case-vertices* :: *nat* ⇒ *'v* ⇒ *bound* ⇒ *'v set* **where**
  base-case-vertices k x B =
    (*let xs* = *base-case-order x B in*
      if length xs ≤ *k then set xs*

else $\{v \in set\ (take\ (Suc\ k)\ xs).\ dist\ s\ v < dist\ s\ (xs\ !\ k)\}$)

definition *base-case-bound* :: $nat \Rightarrow 'v \Rightarrow bound \Rightarrow bound$ **where**
base-case-bound k x B =
(*let xs* = *base-case-order x B in*
if length xs $\leq$ *k then B else Fin* (*dist s* (*xs* ! *k*)))

definition *base-case-result* :: $nat \Rightarrow 'v \Rightarrow bound \Rightarrow bound \times 'v\ set$ **where**
base-case-result k x B =
(*base-case-bound k x B*, *base-case-vertices k x B*)

The helper *closest-vertices* is a choice of a distinct list sorted by true distance. It is intentionally non-executable: the base-case proof is semantic, and later executable theories provide their own finite procedure. The public base-case result is described by *base-case-result*, whose components are *base-case-bound* and *base-case-vertices*. The strict inequality in *base-case-vertices* mirrors the main BMSSP convention for finite bounds.

lemma *base-case-order-set*:
set (*base-case-order x B*) = *bound-tree* $\{x\}$ *B*
⟨*proof*⟩

lemma *base-case-order-sorted*:
sorted-wrt ($\lambda u\ v.$ *dist s u* $\leq$ *dist s v*) (*base-case-order x B*)
⟨*proof*⟩

lemma *base-case-order-distinct*:
distinct (*base-case-order x B*)
⟨*proof*⟩

lemma *in-set-take-dist-lt-nth*:
assumes *sorted*: *sorted-wrt* ($\lambda u\ v.$ *dist s u* $\leq$ *dist s v*) *xs*
and *y*: *y* $\in$ *set xs*
and *lt*: *dist s y* $<$ *dist s* (*xs* ! *k*)
and *k*: *k* $<$ *length xs*
shows *y* $\in$ *set* (*take k xs*)
⟨*proof*⟩

lemma *base-case-success*:
assumes *length* (*base-case-order x B*) $\leq$ *k*
shows *base-case-vertices k x B* = *bound-tree* $\{x\}$ *B*
⟨*proof*⟩

lemma *below-bound-less-trans*:
assumes *a* $<$ *b below-bound b B*
shows *below-bound a B*
⟨*proof*⟩

lemma *Fin-below-bound-le*:
assumes *below-bound a B*

shows *bound-le* (*Fin a*) *B*
⟨*proof*⟩

lemma *base-case-partial*:
assumes *len*: *k* < *length* (*base-case-order x B*)
shows *base-case-vertices k x B* = *bound-tree* {*x*} (*Fin* (*dist s* ((*base-case-order x B*) ! *k*)))
⟨*proof*⟩

lemma *base-case-result-correct*:
assumes *S* = {*x*}
shows *case base-case-result k x B of* (*B*′, *U*) ⇒
U = *bound-tree S B*′ ∧ *complete-on* (λ*v*. *if v* ∈ *U then dist s v else d v*) *U*
⟨*proof*⟩

lemma *base-case-bound-le*:
bound-le (*base-case-bound k x B*) *B*
⟨*proof*⟩

lemma *base-case-result-bound-le*:
case base-case-result k x B of (*B*′, *U*) ⇒ *bound-le B*′ *B*
⟨*proof*⟩

lemma *base-case-result-bmssp-post*:
assumes *S* = {*x*}
shows *case base-case-result k x B of* (*B*′, *U*) ⇒
bmssp-post d S B (λ*v*. *if v* ∈ *U then dist s v else d v*) *B*′ *U*
⟨*proof*⟩

The proof splits on whether the sorted bounded tree has already fit inside the base-case budget. In the success case, *length* (*base-case-order ?x ?B*) ≤ *?k* ⟹ *base-case-vertices ?k ?x ?B* = *bound-tree* {*?x*} *?B* says the returned set is the entire input tree. In the partial case, *?k* < *length* (*base-case-order ?x ?B*) ⟹ *base-case-vertices ?k ?x ?B* = *bound-tree* {*?x*} (*Fin* (*local.dist s* (*base-case-order ?x ?B* ! *?k*))) identifies the returned set with the same tree under the newly returned finite bound. Lemma *?S* = {*?x*} ⟹ *case base-case-result ?k ?x ?B of* (*B*′, *U*) ⇒ *bmssp-post ?d ?S ?B* (λ*v*. *if v* ∈ *U then local.dist s v else ?d v*) *B*′ *U* is the result consumed by the recursive relation: updating labels to *local.dist* on the returned set establishes the BMSSP postcondition.

lemma *base-case-label-sound*:
assumes *sound*: *sound-label d*
shows *sound-label* (λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*)
⟨*proof*⟩

lemma *finite-base-case-vertices* [*simp*]:
finite (*base-case-vertices k x B*)
⟨*proof*⟩

lemma *card-base-case-vertices-le*:
card (*base-case-vertices k x B*) ≤ *k*
⟨*proof*⟩

The remaining facts are bookkeeping for the costed and recursive layers. The base-case label update preserves *sound-label*, the returned set is finite, and *card* (*base-case-vertices ?k ?x ?B*) ≤ *?k* records the intended cap. The strict cutoff in the partial case is what makes this cardinality bound hold: ties at the k-th distance are not included.

end

end
theory *BMSSP-Partition-Interface*
imports *BMSSP-Correctness*
begin

12 Partition Data-Structure Interface

The algorithm's non-base BMSSP loop depends on a partition data structure with *Insert*, *BatchPrepend*, and *Pull*. This theory records the mathematical contract of those operations. It also gives a sorted reference implementation of the functional *Pull* contract. The separate concrete partition-state theory refines this view with explicit active entries, retained value memory, costed operations, and pull-to-split bridge theorems.

The interface is intentionally small. A partition view is just a finite set of keys with a real-valued label for each key. In the algorithm those labels are tentative distances, but the interface does not mention graphs. This separation is what lets the correctness proof talk about the mathematical effect of a pull, while the bucketed refinement later proves that a concrete representation implements the same effect with the desired costs.

Three design choices are worth making explicit. First, inserts are minimum-updates: inserting an existing key keeps the smaller label. Second, batch prepends are admissible only when every prepended label is below every remaining partition label. This is the abstract form of the "prepend a lower block" operation in the paper. Third, a pull does not merely return any small set. It returns a prefix of the partition order, a separating bound, and a residual view whose values are unchanged.

The sorted reference implementation at the end of the file is not the efficient implementation used for the final headline. It is a specification device. Sorting the keys by label makes the pull contract transparent, and the bucketed structure is later shown to realise the same contract while satisfying the paper-tight amortised bounds.

record *'k partition-view* =
keys-of :: *'k set*

```
  value-of :: 'k ⇒ real

definition min-update where
  min-update D x b =
    (| keys-of = insert x (keys-of D),
      value-of = (value-of D)(x := if x ∈ keys-of D then min (value-of D x) b else
b) |)

definition insert-spec where
  insert-spec D x b D' ⟷
     keys-of D' = insert x (keys-of D) ∧
     value-of D' x = (if x ∈ keys-of D then min (value-of D x) b else b) ∧
     (∀ y. y ≠ x ⟶ value-of D' y = value-of D y)

definition batch-min-update where
  batch-min-update D xs = fold (λ(x, b) D'. min-update D' x b) xs D

definition pull-separates where
  pull-separates D M B S x D' ⟷
     S ⊆ keys-of D ∧
     card S ≤ M ∧
     keys-of D' = keys-of D − S ∧
     value-of D' = value-of D ∧
     (∀ u∈S. ∀ v∈keys-of D'. value-of D u ≤ value-of D' v) ∧
     (if keys-of D' = {} then x = B
      else (∀ u∈S. value-of D u < x) ∧ (∀ v∈keys-of D'. x ≤ value-of D' v))

definition batch-prepend-admissible where
  batch-prepend-admissible D xs ⟷
     (∀ (x, b)∈set xs. ∀ y∈keys-of D. b ≤ value-of D y)

definition partition-upper-bound where
  partition-upper-bound D B ⟷ (∀ u∈keys-of D. value-of D u < B)

definition partition-insert-cost-bound where
  partition-insert-cost-bound c t ⟷ c ≤ t

definition partition-batch-cost-bound where
  partition-batch-cost-bound c t xs ⟷ c ≤ t * length xs

definition partition-pull-cost-bound where
  partition-pull-cost-bound c S ⟷ c ≤ card S
```

The three cost predicates are deliberately parametric and algebraic. They do not prescribe an implementation, only the local budget that an implementation must provide to plug into the runtime recurrence. The sorted reference pull has a linear cost, while the bucketed implementation later supplies the stronger logarithmic and amortised bounds required by the paper.

The summation lemmas below are the interface's first payoff: once individual operations satisfy their local predicates, loop-level proofs can sum costs by list induction without opening the implementation of the data structure.

lemma *partition-insert-costs-sum-bound*:
  assumes *list-all* (λc. *partition-insert-cost-bound c t*) *cs*
  shows *sum-list cs* $\leq$ *t* $*$ *length cs*
  ⟨*proof*⟩

lemma *partition-batch-costs-sum-bound*:
  assumes *list-all2* (λc *xs*. *partition-batch-cost-bound c t xs*) *cs xss*
  shows *sum-list cs* $\leq$ *t* $*$ *sum-list* (*map length xss*)
  ⟨*proof*⟩

lemma *partition-pull-costs-sum-bound*:
  assumes *list-all2* (λc *S*. *partition-pull-cost-bound c S*) *cs Ss*
  shows *sum-list cs* $\leq$ *sum-list* (*map card Ss*)
  ⟨*proof*⟩

lemma *partition-batch-cost-bound-append*:
  assumes *xs*: *partition-batch-cost-bound c-x t xs*
    and *ys*: *partition-batch-cost-bound c-y h ys*
    and *h-le-t*: *h* $\leq$ *t*
  shows *partition-batch-cost-bound* (*c-x* + *c-y*) *t* (*xs* @ *ys*)
⟨*proof*⟩

lemma *batch-prepend-admissibleI*:
  assumes $\bigwedge$*x b y*. $[\![$(*x*, *b*) $\in$ *set xs*; *y* $\in$ *keys-of D*$]\!] \Longrightarrow$
    b $\leq$ *value-of D y*
  shows *batch-prepend-admissible D xs*
  ⟨*proof*⟩

lemma *batch-prepend-admissible-below-all*:
  assumes *vals*: $\bigwedge$*x b*. (*x*, *b*) $\in$ *set xs* $\Longrightarrow$ *b* $\leq$ *L*
    and *lower*: $\bigwedge$*y*. *y* $\in$ *keys-of D* $\Longrightarrow$ *L* $\leq$ *value-of D y*
  shows *batch-prepend-admissible D xs*
⟨*proof*⟩

lemma *sorted-value-less-nth-in-take*:
  fixes *f* :: *'a* $\Rightarrow$ *'b*::*linorder*
  assumes *sorted*: *sorted-wrt* (λu *v*. *f u* $\leq$ *f v*) *xs*
    and *y*: *y* $\in$ *set xs*
    and *lt*: *f y* $<$ *f* (*xs* ! *k*)
    and *k*: *k* $<$ *length xs*
  shows *y* $\in$ *set* (*take k xs*)
⟨*proof*⟩

The sorted reference view chooses one canonical order of the partition keys: any distinct list sorted by *value-of*. Isabelle's Hilbert choice is used only to avoid committing to a concrete sorting algorithm in the specification.

The theorem *partition-key-order-properties* immediately recovers the facts needed by later proofs: the order contains exactly the keys, has no duplicates, and is sorted by label.

The reference pull then takes the first block of that order. If at most M keys remain, the pull consumes everything and returns the outer bound. Otherwise it uses the M-th key as the separating bound and pulls exactly the keys with smaller labels. This strict comparison matches the BMSSP lower split used by the correctness proof.

definition *partition-key-order* **where**
  partition-key-order D =
    (*SOME xs. set xs* = *keys-of D* ∧ *distinct xs* ∧
      sorted-wrt (λ*u v. value-of D u* ≤ *value-of D v*) *xs*)

definition *sorted-pull-cost* **where**
  sorted-pull-cost D = *length* (*partition-key-order D*)

lemma *partition-key-order-properties*:
  assumes *finite-keys*: *finite* (*keys-of D*)
  shows *set* (*partition-key-order D*) = *keys-of D*
    and *distinct* (*partition-key-order D*)
    and *sorted-wrt* (λ*u v. value-of D u* ≤ *value-of D v*) (*partition-key-order D*)
⟨*proof*⟩

definition *sorted-pull-set* **where**
  sorted-pull-set M D =
    (*let xs* = *partition-key-order D in*
      if length xs ≤ *M then set xs*
      else {*u* ∈ *keys-of D. value-of D u* < *value-of D* (*xs* ! *M*)})

definition *sorted-pull-bound* **where**
  sorted-pull-bound M B D =
    (*let xs* = *partition-key-order D in*
      if length xs ≤ *M then B else value-of D* (*xs* ! *M*))

definition *remove-keys-view* **where**
  remove-keys-view D S =
    (| *keys-of* = *keys-of D* − *S*, *value-of* = *value-of D* |)

lemma *min-update-keys* [*simp*]:
  keys-of (*min-update D x b*) = *insert x* (*keys-of D*)
  ⟨*proof*⟩

lemma *min-update-value-same*:
  assumes *y* ≠ *x*
  shows *value-of* (*min-update D x b*) *y* = *value-of D y*
  ⟨*proof*⟩

lemma *min-update-value-key-new*:

assumes $x \notin$ *keys-of* D
shows *value-of* (*min-update* D x b) $x = b$
⟨*proof*⟩

lemma *min-update-value-key-old*:
assumes $x \in$ *keys-of* D
shows *value-of* (*min-update* D x b) x = *min* (*value-of* D x) b
⟨*proof*⟩

lemma *min-update-value-le-old*:
assumes $x \in$ *keys-of* D
shows *value-of* (*min-update* D y b) $x \leq$ *value-of* D x
⟨*proof*⟩

lemma *min-update-value-le-inserted*:
value-of (*min-update* D x b) $x \leq b$
⟨*proof*⟩

theorem *min-update-insert-spec*:
insert-spec D x b (*min-update* D x b)
⟨*proof*⟩

lemma *min-update-preserves-upper-bound*:
assumes *upper*: *partition-upper-bound* D B
and *b-lt*: $b < B$
shows *partition-upper-bound* (*min-update* D x b) B
⟨*proof*⟩

lemma *batch-min-update-keys-subset*:
keys-of (*batch-min-update* D *xs*) $\subseteq$ *keys-of* $D \cup$ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *batch-min-update-keys*:
keys-of (*batch-min-update* D *xs*) = *keys-of* $D \cup$ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *batch-min-update-value-le-old*:
assumes $x \in$ *keys-of* D
shows *value-of* (*batch-min-update* D *xs*) $x \leq$ *value-of* D x
⟨*proof*⟩

lemma *batch-min-update-value-le-pair*:
assumes (x, b) $\in$ *set xs*
shows *value-of* (*batch-min-update* D *xs*) $x \leq b$
⟨*proof*⟩

lemma *sorted-pull-cost-linear*:
assumes *finite-keys*: *finite* (*keys-of* D)
shows *sorted-pull-cost* D = *card* (*keys-of* D)

⟨*proof*⟩

The minimum-update lemmas establish the functional behaviour of inserts and batches. They are intentionally elementary: keys are added, old values are never increased, inserted values are respected, and an upper bound is preserved when every inserted value is below that bound. These facts are used by the operational refresh step after a recursive child call has returned.

lemma *batch-min-update-preserves-upper-bound*:
  assumes *upper*: *partition-upper-bound D B*
    and *values-lt*: $\bigwedge x\ b.\ (x,\ b) \in set\ xs \Longrightarrow b < B$
  shows *partition-upper-bound* (*batch-min-update D xs*) *B*
  ⟨*proof*⟩

lemma *pull-separates-pulled-smallest*:
  assumes *pull-separates D M B S x D′*
    and $u \in S$
    and $v \in$ *keys-of D′*
  shows *value-of D u* $\leq$ *value-of D′ v*
  ⟨*proof*⟩

lemma *pull-separates-remaining-keys*:
  assumes *pull-separates D M B S x D′*
  shows *keys-of D′* $=$ *keys-of D* $-$ *S*
  ⟨*proof*⟩

lemma *pull-separates-card*:
  assumes *pull-separates D M B S x D′*
  shows *card S* $\leq$ *M*
  ⟨*proof*⟩

lemma *pull-separates-subset*:
  assumes *pull-separates D M B S x D′*
  shows $S \subseteq$ *keys-of D*
  ⟨*proof*⟩

lemma *pull-separates-preserves-upper-bound*:
  assumes *pull*: *pull-separates D M B0 S x D′*
    and *upper*: *partition-upper-bound D B*
  shows *partition-upper-bound D′ B*
    ⟨*proof*⟩

The pull lemmas unpack the abstract separator contract. A successful pull gives a source set of size at most M, a residual key set with the pulled keys removed, and a bound that lies between the pulled prefix and the residual suffix. The admissibility lemma for batch prepend is the property needed by the BMSSP loop: if every new batch value is below the separator, then the batch may be prepended in front of the residual partition.

lemma *pull-separates-empty-bound*:

assumes *pull-separates D M B S x D′*
and *keys-of D′ = {}*
shows *x = B*
⟨*proof*⟩

lemma *pull-separates-nonempty-bound*:
assumes *pull-separates D M B S x D′*
and *keys-of D′ ≠ {}*
and *v ∈ keys-of D′*
shows *x ≤ value-of D′ v*
⟨*proof*⟩

lemma *pull-separates-bound-le*:
assumes *pull*: *pull-separates D M B S x D′*
and *upper*: *partition-upper-bound D B*
shows *x ≤ B*
⟨*proof*⟩

lemma *pull-separates-batch-prepend-admissible*:
assumes *pull*: *pull-separates D M B S x D′*
and *vals*: *⋀u b. (u, b) ∈ set xs ⟹ b ≤ x*
shows *batch-prepend-admissible D′ xs*
⟨*proof*⟩

theorem *pull-separates-exact-split*:
assumes *pull*: *pull-separates D M B S x D′*
and *upper*: *⋀u. u ∈ keys-of D ⟹ value-of D u < B*
shows *S = {u ∈ keys-of D. value-of D u < x}*
⟨*proof*⟩

The final theorems connect the sorted reference operation to the abstract pull contract. *sorted-pull-separates* proves that the reference operation really returns a separating prefix, and *sorted-pull-exact-split* proves that the same prefix is exactly the strict lower split at the returned bound whenever all labels are below the outer bound.

These two facts are specification tools. The executable bucketed partition does not sort all keys on every pull; instead, it proves the same separator and split properties for its bucket state and then plugs those properties into the generic BMSSP correctness and cost layers.

theorem *sorted-pull-separates*:
fixes *M :: nat*
and *B :: real*
assumes *finite-keys*: *finite (keys-of D)*
and *S-def*: *S = sorted-pull-set M D*
and *x-def*: *x = sorted-pull-bound M B D*
and *D′-def*: *D′ = remove-keys-view D S*
shows *pull-separates D M B S x D′*
⟨*proof*⟩

theorem *sorted-pull-exact-split*:
  assumes *finite-keys*: *finite* (*keys-of D*)
    and *upper*: $\bigwedge u.\ u \in$ *keys-of* $D \Longrightarrow$ *value-of* $D\ u < B$
  shows *sorted-pull-set* $M\ D =$
    $\{u \in$ *keys-of* $D.$ *value-of* $D\ u <$ *sorted-pull-bound* $M\ B\ D\}$
⟨*proof*⟩

end
theory *BMSSP-Bucketed-Partition-Internal*
  imports *BMSSP-Partition-Interface*
    HOL−Library.Discrete-Functions
    HOL−Library.Landau-Symbols
begin

13 Bucketed Partition Data Structure

This is the main data-structure theory for the paper-faithful partition layer. The BMSSP paper uses a bucketed structure whose operation bounds are expressed in terms of the ratio between the number of inserted elements and the block size. That ratio is the essential distinction from a sorted-list model: Insert must pay for a search over buckets, not for a search over all stored elements.

The formal model is functional. A partition state contains a block size, an upper bound, a list of buckets, and a value memory. Each bucket has a marker and an unsorted list of key/value entries. The marker is a lower bound for the values in the bucket, and adjacent markers delimit the value ranges used by Pull. The list-level operations below first establish a simple exact model, then a lazy model that permits buckets to grow up to the split threshold before they are rebuilt.

The theory has three layers. The first layer defines the functional state, invariants, Insert, BatchPrepend, and Pull, and proves that their views refine the abstract partition interface. The second layer adds primitive step counters and a potential function for lazy splitting. The third layer names the paper-facing operations and proves the three exported cost bounds: Insert at the ratio-log budget, BatchPrepend at the matching batched budget, and Pull at an amortized linear-in-M budget. The baseline sorted-list partition state used by the recurrence proofs is not used here.

The amortized analysis uses a scaled credit potential $\Phi(P) =$ *4* $\cdot \sum_b$ *max 0* (*length* (*bp-bucket-entries b*) $-$ *bp-block-size P*), summed over the buckets of P. The factor of four is the constant that absorbs the per-Insert charge: each Insert raises Φ by at most four credits, which is exactly the budget that pays for the $O(M)$ rebucket work performed during a Pull when a bucket has overflowed past the lazy split threshold. This bookkeeping is what makes the bound paper-tight rather than merely amortized-but-loose:

an Insert that does not split the bucket pays only for the bucket walk $O(log\ (N\ /\ M))$, an Insert that triggers a rebucket pays the same physical cost plus a refund of credits to Φ, and Pull discharges the credits accumulated by its bucket's prior Inserts to fund the rebucket step. Without the factor four the per-Insert charge would not cover the rebucket, and the resulting Insert bound would degrade to $O(N\ /\ M)$; with the factor four it stays at the paper rate.

```
record 'k bp-bucket =
  bp-marker :: real
  bp-bucket-entries :: ('k × real) list

record 'k bucketed-partition =
  bp-block-size :: nat
  bp-upper-bound :: real
  bp-buckets :: 'k bp-bucket list
  bp-values :: 'k ⇒ real

definition bp-bucket-entries-flat :: 'k bp-bucket list ⇒ ('k × real) list where
  bp-bucket-entries-flat bs = concat (map bp-bucket-entries bs)

definition bp-entries :: 'k bucketed-partition ⇒ ('k × real) list where
  bp-entries P = bp-bucket-entries-flat (bp-buckets P)

definition bp-entry-keys :: ('k × real) list ⇒ 'k set where
  bp-entry-keys xs = fst ‘ set xs

definition bp-bucket-keys :: 'k bp-bucket ⇒ 'k set where
  bp-bucket-keys b = bp-entry-keys (bp-bucket-entries b)

definition bp-view :: 'k bucketed-partition ⇒ 'k partition-view where
  bp-view P =
    (| keys-of = bp-entry-keys (bp-entries P),
      value-of = bp-values P |)

definition bp-bucket-sizes-ok :: 'k bucketed-partition ⇒ bool where
  bp-bucket-sizes-ok P ⟷
    (∀ b∈set (bp-buckets P).
      length (bp-bucket-entries b) ≤ bp-block-size P)

definition bp-lazy-bucket-sizes-ok :: 'k bucketed-partition ⇒ bool where
  bp-lazy-bucket-sizes-ok P ⟷
    (∀ b∈set (bp-buckets P).
      length (bp-bucket-entries b) ≤ 2 * bp-block-size P)

definition bp-bucket-markers-sorted :: 'k bucketed-partition ⇒ bool where
  bp-bucket-markers-sorted P ⟷
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (bp-buckets P)
```

definition *bp-bucket-markers-lower-bound* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-bucket-markers-lower-bound P ⟷
(∀ *b*∈*set* (*bp-buckets P*).
∀ *p*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd p*)

definition *bp-values-consistent* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-values-consistent P ⟷
(∀ *p*∈*set* (*bp-entries P*). *bp-values P* (*fst p*) = *snd p*)

definition *bp-distinct-keys* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-distinct-keys P ⟷ *distinct* (*map fst* (*bp-entries P*))

fun *bp-bucket-boundaries-ok* :: *'k bp-bucket list* ⇒ *bool* **where**
bp-bucket-boundaries-ok [] ⟷ *True*
| *bp-bucket-boundaries-ok* [*b*] ⟷ *True*
| *bp-bucket-boundaries-ok* (*b* # *c* # *bs*) ⟷
(∀ *p*∈*set* (*bp-bucket-entries b*). *snd p* ≤ *bp-marker c*) ∧
bp-bucket-boundaries-ok (*c* # *bs*)

definition *bp-bucket-boundaries-state-ok* ::
'k bucketed-partition ⇒ *bool* **where**
bp-bucket-boundaries-state-ok P ⟷
bp-bucket-boundaries-ok (*bp-buckets P*)

definition *bp-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-invariant P ⟷
0 < *bp-block-size P* ∧
bp-distinct-keys P ∧
bp-bucket-sizes-ok P ∧
bp-bucket-markers-sorted P ∧
bp-bucket-markers-lower-bound P ∧
bp-values-consistent P

definition *bp-ordered-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-ordered-invariant P ⟷
bp-invariant P ∧ *bp-bucket-boundaries-state-ok P*

definition *bp-lazy-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-lazy-invariant P ⟷
0 < *bp-block-size P* ∧
bp-distinct-keys P ∧
bp-lazy-bucket-sizes-ok P ∧
bp-bucket-markers-sorted P ∧
bp-bucket-markers-lower-bound P ∧
bp-values-consistent P

definition *bp-lazy-ordered-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-lazy-ordered-invariant P ⟷
bp-lazy-invariant P ∧ *bp-bucket-boundaries-state-ok P*

The invariant vocabulary separates three concerns. The basic view of a state is *bp-view*: it exposes only the key set and the value memory required by the abstract partition interface. The structural invariant *bp-invariant* says that the block size is positive, keys are unique, strict bucket sizes are at most *bp-block-size*, markers are sorted, markers are lower bounds for their entries, and the stored value memory agrees with the bucket entries. The ordered invariant *bp-ordered-invariant* adds the boundary condition between adjacent buckets, which is what makes a first-bucket Pull sound.

The lazy variants are used for the amortized Insert proof. In *bp-lazy-invariant*, buckets may temporarily contain up to twice the block size. This slack is exactly what lets Insert avoid rebuilding on every operation. Once a bucket crosses the lazy threshold, the state is rebuilt into strict buckets and the potential drops enough to pay for the split.

definition *bp-empty* :: *nat* ⇒ *real* ⇒ *'k bucketed-partition* **where**
  bp-empty M B =
    (| *bp-block-size* = *M*,
      bp-upper-bound = *B*,
      bp-buckets = [],
      bp-values = (λ-. *B*) |)

definition *bp-singleton-bucket* :: *'k* × *real* ⇒ *'k bp-bucket* **where**
  bp-singleton-bucket p =
    (| *bp-marker* = *snd p*, *bp-bucket-entries* = [*p*] |)

lemma *bp-singleton-bucket-simps* [*simp*]:
  bp-marker (*bp-singleton-bucket p*) = *snd p*
  bp-bucket-entries (*bp-singleton-bucket p*) = [*p*]
  ⟨*proof*⟩

definition *bp-make-bucket* :: (*'k* × *real*) *list* ⇒ *'k bp-bucket* **where**
  bp-make-bucket xs =
    (| *bp-marker* = *snd* (*hd xs*), *bp-bucket-entries* = *xs* |)

fun *bp-bucketize-sorted-entries-aux* ::
  nat ⇒ *nat* ⇒ (*'k* × *real*) *list* ⇒ *'k bp-bucket list* **where**
  bp-bucketize-sorted-entries-aux 0 M xs = []
| *bp-bucketize-sorted-entries-aux* (*Suc fuel*) *M xs* =
    (*if M* = *0* ∨ *xs* = []
     then []
     else bp-make-bucket (*take M xs*) #
       bp-bucketize-sorted-entries-aux fuel M (*drop M xs*))

definition *bp-bucketize-sorted-entries* ::
  nat ⇒ (*'k* × *real*) *list* ⇒ *'k bp-bucket list* **where**
  bp-bucketize-sorted-entries M xs =
    bp-bucketize-sorted-entries-aux (*length xs*) *M xs*

definition *bp-bucketize-entries* ::
nat ⇒ (*'k* × *real*) *list* ⇒ *'k bp-bucket list* **where**
bp-bucketize-entries M xs = *bp-bucketize-sorted-entries M* (*sort-key snd xs*)

lemma *bp-bucketize-sorted-entries-aux-empty* [*simp*]:
bp-bucketize-sorted-entries-aux fuel M [] = []
⟨*proof*⟩

definition *bp-rebucket* :: *'k bucketed-partition* ⇒ *'k bucketed-partition* **where**
bp-rebucket P =
P(|*bp-buckets* := *bp-bucketize-entries* (*bp-block-size P*) (*bp-entries P*)|)

fun *bp-local-insert-bucket* ::
nat ⇒ *'k* × *real* ⇒ *'k bp-bucket list* ⇒ *'k bp-bucket list* **where**
bp-local-insert-bucket M p [] = *bp-bucketize-entries M* [*p*]
| *bp-local-insert-bucket M p* [*b*] =
(*if snd p* < *bp-marker b*
then bp-bucketize-entries M [*p*] @ [*b*]
else bp-bucketize-entries M (*p* # *bp-bucket-entries b*))
| *bp-local-insert-bucket M p* (*b* # *c* # *bs*) =
(*if snd p* < *bp-marker b*
then bp-bucketize-entries M [*p*] @ *b* # *c* # *bs*
else if snd p < *bp-marker c*
then bp-bucketize-entries M (*p* # *bp-bucket-entries b*) @ *c* # *bs*
else b # *bp-local-insert-bucket M p* (*c* # *bs*))

definition *bp-lazy-bucket-insert-entries* ::
nat ⇒ *'k* × *real* ⇒ *'k bp-bucket* ⇒ *'k bp-bucket list* **where**
bp-lazy-bucket-insert-entries M p b =
(*if length* (*bp-bucket-entries b*) < *2* ∗ *M*
then [*b*(|*bp-bucket-entries* := *p* # *bp-bucket-entries b*|)]
else bp-bucketize-entries M (*p* # *bp-bucket-entries b*))

fun *bp-lazy-insert-bucket* ::
nat ⇒ *'k* × *real* ⇒ *'k bp-bucket list* ⇒ *'k bp-bucket list* **where**
bp-lazy-insert-bucket M p [] = *bp-bucketize-entries M* [*p*]
| *bp-lazy-insert-bucket M p* [*b*] =
(*if snd p* < *bp-marker b*
then bp-bucketize-entries M [*p*] @ [*b*]
else bp-lazy-bucket-insert-entries M p b)
| *bp-lazy-insert-bucket M p* (*b* # *c* # *bs*) =
(*if snd p* < *bp-marker b*
then bp-bucketize-entries M [*p*] @ *b* # *c* # *bs*
else if snd p < *bp-marker c*
then bp-lazy-bucket-insert-entries M p b @ *c* # *bs*
else b # *bp-lazy-insert-bucket M p* (*c* # *bs*))

fun *bp-insert-bucket* :: *'k* × *real* ⇒ *'k bp-bucket list* ⇒ *'k bp-bucket list* **where**
bp-insert-bucket p [] = [*bp-singleton-bucket p*]

```
| bp-insert-bucket p (b # bs) =
    (if snd p ≤ bp-marker b
     then bp-singleton-bucket p # b # bs
     else b # bp-insert-bucket p bs)

definition bp-delete-key-from-bucket :: 'k ⇒ 'k bp-bucket ⇒ 'k bp-bucket where
  bp-delete-key-from-bucket x b =
    b(|bp-bucket-entries := filter (λp. fst p ≠ x) (bp-bucket-entries b)|)

definition bp-delete-key :: 'k ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-delete-key x P =
    P(|bp-buckets := map (bp-delete-key-from-bucket x) (bp-buckets P)|)

definition bp-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-insert x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in P0(|bp-buckets := bp-insert-bucket (x, b') (bp-buckets P0),
            bp-values := (bp-values P)(x := b')|))

definition bp-local-insert-state ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-local-insert-state x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in P0(|bp-buckets :=
            bp-local-insert-bucket (bp-block-size P0) (x, b') (bp-buckets P0),
            bp-values := (bp-values P)(x := b')|))

definition bp-lazy-insert-state ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-lazy-insert-state x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in P0(|bp-buckets :=
            bp-lazy-insert-bucket (bp-block-size P0) (x, b') (bp-buckets P0),
            bp-values := (bp-values P)(x := b')|))

definition bp-batch-prepend ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-batch-prepend xs P =
    fold (λ(x, b) P'. bp-insert x b P') xs P

fun bp-batch-value-update ::
  ('k × real) list ⇒ ('k ⇒ real) ⇒ 'k ⇒ real where
```

```
  bp-batch-value-update [] f = f
| bp-batch-value-update ((x, b) # xs) f =
    bp-batch-value-update xs (f(x := b))

fun bp-batch-max-value :: real ⇒ ('k × real) list ⇒ real where
  bp-batch-max-value beta [] = beta
| bp-batch-max-value beta ((x, b) # xs) =
    bp-batch-max-value (max beta b) xs

fun bp-rebase-first-bucket-marker ::
  real ⇒ 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-rebase-first-bucket-marker beta [] = []
| bp-rebase-first-bucket-marker beta (b # bs) =
    b(| bp-marker := beta |) # bs

fun bp-drop-empty-prefix :: 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-drop-empty-prefix [] = []
| bp-drop-empty-prefix (b # bs) =
    (if bp-bucket-entries b = [] then bp-drop-empty-prefix bs else b # bs)

definition bp-bucketed-batch-prepend-state ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition where
  bp-bucketed-batch-prepend-state xs P =
    (case xs of
       [] ⇒ P
     | p # ps ⇒
       (let beta = bp-batch-max-value (snd p) ps;
          new = bp-bucketize-entries (bp-block-size P) xs;
          old = bp-rebase-first-bucket-marker beta
            (bp-drop-empty-prefix (bp-buckets P))
        in P(| bp-buckets := new @ old,
               bp-values := bp-batch-value-update xs (bp-values P) |)))

definition bp-delete-keys-from-bucket :: 'k set ⇒ 'k bp-bucket ⇒ 'k bp-bucket where
  bp-delete-keys-from-bucket S b =
    b(| bp-bucket-entries := filter (λp. fst p ∉ S) (bp-bucket-entries b) |)

definition bp-delete-keys :: 'k set ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition
where
  bp-delete-keys S P =
    P(| bp-buckets := map (bp-delete-keys-from-bucket S) (bp-buckets P) |)

fun bp-min-value :: real ⇒ ('k × real) list ⇒ real where
  bp-min-value B [] = B
| bp-min-value B ((x, b) # xs) = min b (bp-min-value B xs)

definition bp-bucket-below-bound :: 'k bp-bucket ⇒ real ⇒ bool where
  bp-bucket-below-bound b beta ⟷
```

```
    (∀ p∈set (bp-bucket-entries b). snd p < beta)

definition bp-first-bucket-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
    'k set × real × 'k bucketed-partition where
  bp-first-bucket-pull M B P =
    (case bp-buckets P of
       b # c # bs ⇒
         (let S = bp-bucket-keys b
          in (S, bp-marker c, bp-delete-keys S P))
     | - ⇒ ({}, B, P))

definition bp-pull-set :: nat ⇒ 'k bucketed-partition ⇒ 'k set where
  bp-pull-set M P =
    (if length (bp-entries P) ≤ M
     then bp-entry-keys (bp-entries P)
     else {})

definition bp-pull-bound :: nat ⇒ real ⇒ 'k bucketed-partition ⇒ real where
  bp-pull-bound M B P =
    (if length (bp-entries P) ≤ M
     then B
     else bp-min-value B (bp-entries P))

definition bp-conservative-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
    'k set × real × 'k bucketed-partition where
  bp-conservative-pull M B P =
    (let S = bp-pull-set M P;
       beta = bp-pull-bound M B P
     in (S, beta, bp-delete-keys S P))

definition bp-can-first-bucket-pull ::
  nat ⇒ 'k bucketed-partition ⇒ bool where
  bp-can-first-bucket-pull M P ⟷
    (case bp-buckets P of
       b # c # bs ⇒
         length (bp-entries P) > M ∧
         length (bp-bucket-entries b) ≤ M ∧
         bp-bucket-below-bound b (bp-marker c) ∧
         bp-bucket-entries-flat (c # bs) ≠ []
     | - ⇒ False)

definition bp-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
    'k set × real × 'k bucketed-partition where
  bp-pull M B P =
    (if bp-can-first-bucket-pull M P
     then bp-first-bucket-pull M B P
```

else bp-conservative-pull M B P)

lemma *bp-can-first-bucket-pullE*:
assumes *bp-can-first-bucket-pull M P*
obtains *b c bs* **where**
bp-buckets P = *b* # *c* # *bs*
length (*bp-entries P*) > *M*
length (*bp-bucket-entries b*) ≤ *M*
bp-bucket-below-bound b (*bp-marker c*)
bp-bucket-entries-flat (*c* # *bs*) ≠ []
⟨*proof*⟩

lemma *bp-entry-keys-filter-neq* [*simp*]:
bp-entry-keys (*filter* (λ*p*. *fst p* ≠ *x*) *xs*) = *bp-entry-keys xs* − {*x*}
⟨*proof*⟩

lemma *bp-entry-keys-filter-notin* [*simp*]:
bp-entry-keys (*filter* (λ*p*. *fst p* ∉ *S*) *xs*) = *bp-entry-keys xs* − *S*
⟨*proof*⟩

lemma *bp-bucket-keys-alt* [*simp*]:
bp-bucket-keys b = *fst* ‘ *set* (*bp-bucket-entries b*)
⟨*proof*⟩

lemma *bp-entries-empty* [*simp*]:
bp-entries (*bp-empty M B*) = []
⟨*proof*⟩

lemma *bp-bucket-entries-flat-append* [*simp*]:
bp-bucket-entries-flat (*bs* @ *cs*) =
bp-bucket-entries-flat bs @ *bp-bucket-entries-flat cs*
⟨*proof*⟩

lemma *bp-bucket-entries-flat-rebase-first-bucket-marker* [*simp*]:
bp-bucket-entries-flat (*bp-rebase-first-bucket-marker beta bs*) =
bp-bucket-entries-flat bs
⟨*proof*⟩

lemma *bp-bucket-entries-flat-drop-empty-prefix* [*simp*]:
bp-bucket-entries-flat (*bp-drop-empty-prefix bs*) =
bp-bucket-entries-flat bs
⟨*proof*⟩

lemma *length-bp-rebase-first-bucket-marker* [*simp*]:
length (*bp-rebase-first-bucket-marker beta bs*) = *length bs*
⟨*proof*⟩

lemma *length-bp-drop-empty-prefix-le* [*simp*]:
length (*bp-drop-empty-prefix bs*) ≤ *length bs*

⟨*proof*⟩

lemma *bp-entry-keys-rebase-first-bucket-marker* [*simp*]:
bp-entry-keys
(*bp-bucket-entries-flat* (*bp-rebase-first-bucket-marker beta bs*)) =
bp-entry-keys (*bp-bucket-entries-flat bs*)
⟨*proof*⟩

lemma *bp-entry-keys-drop-empty-prefix* [*simp*]:
bp-entry-keys (*bp-bucket-entries-flat* (*bp-drop-empty-prefix bs*)) =
bp-entry-keys (*bp-bucket-entries-flat bs*)
⟨*proof*⟩

lemma *bp-drop-empty-prefix-set-subset*:
set (*bp-drop-empty-prefix bs*) ⊆ *set bs*
⟨*proof*⟩

lemma *bp-drop-empty-prefix-head-nonempty*:
assumes *bp-drop-empty-prefix bs* = *b* # *bs*′
shows *bp-bucket-entries b* ≠ []
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-drop-empty-prefix*:
assumes *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*bp-drop-empty-prefix bs*)
⟨*proof*⟩

lemma *bp-bucket-markers-sorted-drop-empty-prefix*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*bp-drop-empty-prefix bs*)
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-rebase-first-bucket-marker* [*simp*]:
bp-bucket-boundaries-ok (*bp-rebase-first-bucket-marker beta bs*) =
bp-bucket-boundaries-ok bs
⟨*proof*⟩

lemma *bp-batch-value-update-notin*:
assumes *x* ∉ *fst* ‘ *set xs*
shows *bp-batch-value-update xs f x* = *f x*
⟨*proof*⟩

lemma *bp-batch-value-update-in-set-distinct*:
assumes *distinct*: *distinct* (*map fst xs*)
and *member*: (*x*, *b*) ∈ *set xs*
shows *bp-batch-value-update xs f x* = *b*
⟨*proof*⟩

lemma *batch-min-update-value-distinct-disjoint*:
assumes *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *fst ' set xs* ∩ *keys-of D* = {}
shows *value-of* (*batch-min-update D xs*) =
bp-batch-value-update xs (*value-of D*)
⟨*proof*⟩

lemma *bp-batch-max-value-upper*:
assumes ⋀*p*. *p* ∈ *set xs* ⟹ *snd p* ≤ *gamma*
and *beta* ≤ *gamma*
shows *bp-batch-max-value beta xs* ≤ *gamma*
⟨*proof*⟩

lemma *bp-batch-max-value-ge-initial*:
beta ≤ *bp-batch-max-value beta xs*
⟨*proof*⟩

lemma *bp-batch-max-value-ge-member*:
assumes *p* ∈ *set xs*
shows *snd p* ≤ *bp-batch-max-value beta xs*
⟨*proof*⟩

lemma *bp-batch-max-value-ge-member-Cons*:
assumes *q* ∈ *set* (*p* # *ps*)
shows *snd q* ≤ *bp-batch-max-value* (*snd p*) *ps*
⟨*proof*⟩

lemma *bp-bucket-entries-flat-bucketize-sorted-entries-aux*:
assumes *0* < *M*
and *length xs* ≤ *fuel*
shows *bp-bucket-entries-flat*
(*bp-bucketize-sorted-entries-aux fuel M xs*) = *xs*
⟨*proof*⟩

lemma *bp-bucket-entries-flat-bucketize-sorted-entries*:
assumes *0* < *M*
shows *bp-bucket-entries-flat* (*bp-bucketize-sorted-entries M xs*) = *xs*
⟨*proof*⟩

lemma *bp-bucket-entries-flat-bucketize-entries*:
assumes *0* < *M*
shows *bp-bucket-entries-flat* (*bp-bucketize-entries M xs*) = *sort-key snd xs*
⟨*proof*⟩

lemma *set-bp-bucket-entries-flat-bucketize-entries*:
assumes *0* < *M*
shows *set* (*bp-bucket-entries-flat* (*bp-bucketize-entries M xs*)) = *set xs*
⟨*proof*⟩

lemma *bp-entry-keys-bucketize-entries* [*simp*]:
assumes *0 < M*
shows *bp-entry-keys* (*bp-bucket-entries-flat* (*bp-bucketize-entries M xs*)) =
bp-entry-keys xs
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-aux-sizes-ok*:
assumes *0 < M*
shows ∀ *b*∈*set* (*bp-bucketize-sorted-entries-aux fuel M xs*).
length (*bp-bucket-entries b*) ≤ *M*
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-sizes-ok*:
assumes *0 < M*
shows ∀ *b*∈*set* (*bp-bucketize-sorted-entries M xs*).
length (*bp-bucket-entries b*) ≤ *M*
⟨*proof*⟩

lemma *bp-bucketize-entries-sizes-ok*:
assumes *0 < M*
shows ∀ *b*∈*set* (*bp-bucketize-entries M xs*).
length (*bp-bucket-entries b*) ≤ *M*
⟨*proof*⟩

lemma *bp-sorted-wrt-snd-sort-key* [*simp*]:
sorted-wrt (λ*p q*. *snd p* ≤ *snd q*) (*sort-key snd xs*)
⟨*proof*⟩

lemma *sorted-wrt-hd-snd-le*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *sorted*: *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
and *xs*: *xs* ≠ []
and *p*: *p* ∈ *set xs*
shows *snd* (*hd xs*) ≤ *snd p*
⟨*proof*⟩

lemma *sorted-wrt-snd-take-drop-le*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *sorted*: *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
and *p*: *p* ∈ *set* (*take n xs*)
and *q*: *q* ∈ *set* (*drop n xs*)
shows *snd p* ≤ *snd q*
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-aux-marker-in-set*:
assumes *b* ∈ *set* (*bp-bucketize-sorted-entries-aux fuel M xs*)
shows ∃ *p*∈*set xs*. *bp-marker b* = *snd p*
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-aux-markers-lower-bound*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *M-pos*: *0* < *M*
and *sorted*: *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
shows ∀ *b*∈*set* (*bp-bucketize-sorted-entries-aux fuel M xs*).
∀ *p*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd p*
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-markers-lower-bound*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *0* < *M*
and *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
shows ∀ *b*∈*set* (*bp-bucketize-sorted-entries M xs*).
∀ *p*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd p*
⟨*proof*⟩

lemma *bp-bucketize-entries-markers-lower-bound*:
assumes *0* < *M*
shows ∀ *b*∈*set* (*bp-bucketize-entries M xs*).
∀ *p*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd p*
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-aux-markers-sorted*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *M-pos*: *0* < *M*
and *sorted*: *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*bp-bucketize-sorted-entries-aux fuel M xs*)
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-markers-sorted*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *0* < *M*
and *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*bp-bucketize-sorted-entries M xs*)
⟨*proof*⟩

lemma *bp-bucketize-entries-markers-sorted*:
assumes *0* < *M*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*bp-bucketize-entries M xs*)
⟨*proof*⟩

lemma *bp-bucketize-entries-marker-in-set*:
assumes *b* ∈ *set* (*bp-bucketize-entries M xs*)
shows ∃ *p*∈*set xs*. *bp-marker b* = *snd p*
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-aux-boundaries-ok*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *M-pos*: *0* < *M*
and *sorted*: *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
shows *bp-bucket-boundaries-ok*
(*bp-bucketize-sorted-entries-aux fuel M xs*)
⟨*proof*⟩

lemma *bp-bucketize-sorted-entries-boundaries-ok*:
fixes *xs* :: ('*k* × *real*) *list*
assumes *0* < *M*
and *sorted-wrt* (λ*p q*. *snd p* ≤ *snd q*) *xs*
shows *bp-bucket-boundaries-ok* (*bp-bucketize-sorted-entries M xs*)
⟨*proof*⟩

lemma *bp-bucketize-entries-boundaries-ok*:
assumes *0* < *M*
shows *bp-bucket-boundaries-ok* (*bp-bucketize-entries M xs*)
⟨*proof*⟩

lemma *distinct-map-fst-sort-key*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*sort-key f xs*))
⟨*proof*⟩

lemma *bp-rebucket-view* [*simp*]:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-rebucket P*) = *bp-view P*
⟨*proof*⟩

lemma *bp-rebucket-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-boundaries-state-ok*:
assumes *inv*: *bp-invariant P*
shows *bp-bucket-boundaries-state-ok* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-ordered-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-ordered-invariant* (*bp-rebucket P*)
⟨*proof*⟩

Rebucketing is the canonical repair operation. It sorts the flat entry list by value, chunks it into blocks of size *bp-block-size*, and rebuilds markers from the first entry of each block. The view theorem *0* < *bp-block-size ?P* ⟹ *bp-view* (*bp-rebucket ?P*) = *bp-view ?P* says this repair is observationally invisible at

the abstract partition interface: keys and remembered values do not change. The invariant theorems above then show that rebucketing restores strict bucket sizes, sorted markers, lower-bound markers, and adjacent bucket boundaries. Later regularized operations use this fact when a lazy step has accumulated enough debt to warrant a rebuild.

lemma *bp-ordered-invariant-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-invariant P*
⟨*proof*⟩

lemma *bp-ordered-invariant-boundaries-state-ok*:
assumes *bp-ordered-invariant P*
shows *bp-bucket-boundaries-state-ok P*
⟨*proof*⟩

lemma *bp-invariant-lazy-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-lazy-invariant P*
⟨*proof*⟩

lemma *bp-ordered-invariant-lazy-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-lazy-ordered-invariant P*
⟨*proof*⟩

lemma *bp-lazy-ordered-invariant-lazy-invariant*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-lazy-invariant P*
⟨*proof*⟩

lemma *bp-lazy-ordered-invariant-boundaries-state-ok*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-bucket-boundaries-state-ok P*
⟨*proof*⟩

lemma *bp-rebucket-invariant-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-invariant* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-boundaries-state-ok-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-bucket-boundaries-state-ok* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-ordered-invariant-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-ordered-invariant* (*bp-rebucket P*)

⟨*proof*⟩

lemma *set-bp-bucket-entries-flat-local-insert-bucket*:
assumes *0 < M*
shows *set (bp-bucket-entries-flat (bp-local-insert-bucket M p bs)) =*
insert p (set (bp-bucket-entries-flat bs))
⟨*proof*⟩

lemma *set-bp-bucket-entries-flat-lazy-bucket-insert-entries*:
assumes *0 < M*
shows *set (bp-bucket-entries-flat*
(bp-lazy-bucket-insert-entries M p b)) =
insert p (set (bp-bucket-entries b))
⟨*proof*⟩

lemma *set-bp-bucket-entries-flat-lazy-insert-bucket*:
assumes *0 < M*
shows *set (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)) =*
insert p (set (bp-bucket-entries-flat bs))
⟨*proof*⟩

lemma *length-bp-bucket-entries-flat-lazy-bucket-insert-entries* [*simp*]:
assumes *0 < M*
shows *length (bp-bucket-entries-flat*
(bp-lazy-bucket-insert-entries M p b)) =
Suc (length (bp-bucket-entries b))
⟨*proof*⟩

lemma *length-bp-bucket-entries-flat-lazy-insert-bucket* [*simp*]:
assumes *0 < M*
shows *length (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)) =*
Suc (length (bp-bucket-entries-flat bs))
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-distinct-keys*:
assumes *M-pos*: *0 < M*
and *distinct*: *distinct (map fst (bp-bucket-entries-flat bs))*
and *fresh*: *fst p ∉ bp-entry-keys (bp-bucket-entries-flat bs)*
shows *distinct*
(map fst (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)))
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-sizes-ok*:
assumes *M-pos*: *0 < M*
and *size*: *length (bp-bucket-entries b) ≤ 2 ∗ M*
shows *∀ c∈set (bp-lazy-bucket-insert-entries M p b).*
length (bp-bucket-entries c) ≤ 2 ∗ M
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-sizes-ok*:
assumes *M-pos*: $0 < M$
and *sizes*: $\forall\, b{\in}set\ bs.\ length\ (bp\text{-}bucket\text{-}entries\ b) \leq 2 * M$
shows $\forall\, b{\in}set\ (bp\text{-}lazy\text{-}insert\text{-}bucket\ M\ p\ bs).$
$length\ (bp\text{-}bucket\text{-}entries\ b) \leq 2 * M$
⟨*proof*⟩

lemma *bp-local-insert-bucket-sizes-ok*:
assumes *M-pos*: $0 < M$
and *sizes*: $\forall\, b{\in}set\ bs.\ length\ (bp\text{-}bucket\text{-}entries\ b) \leq M$
shows $\forall\, b{\in}set\ (bp\text{-}local\text{-}insert\text{-}bucket\ M\ p\ bs).$
$length\ (bp\text{-}bucket\text{-}entries\ b) \leq M$
⟨*proof*⟩

lemma *bp-local-insert-bucket-markers-lower-bound*:
assumes *M-pos*: $0 < M$
and *lower*: $\forall\, b{\in}set\ bs.\ \forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\ b).$
$bp\text{-}marker\ b \leq snd\ p$
shows $\forall\, b{\in}set\ (bp\text{-}local\text{-}insert\text{-}bucket\ M\ p\ bs).$
$\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ b).\ bp\text{-}marker\ b \leq snd\ q$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-markers-lower-bound*:
assumes *M-pos*: $0 < M$
and *lower*: $\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ b).\ bp\text{-}marker\ b \leq snd\ q$
and *p-ge*: $bp\text{-}marker\ b \leq snd\ p$
shows $\forall\, c{\in}set\ (bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b).$
$\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ c).\ bp\text{-}marker\ c \leq snd\ q$
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-markers-lower-bound*:
assumes *M-pos*: $0 < M$
and *lower*: $\forall\, b{\in}set\ bs.\ \forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\ b).$
$bp\text{-}marker\ b \leq snd\ p$
shows $\forall\, b{\in}set\ (bp\text{-}lazy\text{-}insert\text{-}bucket\ M\ p\ bs).$
$\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ b).\ bp\text{-}marker\ b \leq snd\ q$
⟨*proof*⟩

lemma *bp-bucketize-entries-markers-le*:
fixes *beta* :: *real*
assumes $\forall\, q{\in}set\ xs.\ snd\ q \leq beta$
shows $\forall\, b{\in}set\ (bp\text{-}bucketize\text{-}entries\ M\ xs).\ bp\text{-}marker\ b \leq beta$
⟨*proof*⟩

lemma *bp-bucketize-entries-markers-ge*:
fixes *alpha* :: *real*
assumes $\forall\, q{\in}set\ xs.\ alpha \leq snd\ q$
shows $\forall\, b{\in}set\ (bp\text{-}bucketize\text{-}entries\ M\ xs).\ alpha \leq bp\text{-}marker\ b$
⟨*proof*⟩

lemma *bp-bucketize-entries-entry-in-source*:
assumes *M-pos*: $0 < M$
and *b*: $b \in set\ (bp\text{-}bucketize\text{-}entries\ M\ xs)$
and *q*: $q \in set\ (bp\text{-}bucket\text{-}entries\ b)$
shows $q \in set\ xs$
⟨*proof*⟩

lemma *bp-bucketize-entries-nonempty*:
assumes $0 < M$ **and** $xs \neq []$
shows $bp\text{-}bucketize\text{-}entries\ M\ xs \neq []$
⟨*proof*⟩

lemma *bp-local-insert-bucket-nonempty*:
assumes $0 < M$
shows $bp\text{-}local\text{-}insert\text{-}bucket\ M\ p\ bs \neq []$
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-append*:
assumes *left*: $bp\text{-}bucket\text{-}boundaries\text{-}ok\ xs$
and *right*: $bp\text{-}bucket\text{-}boundaries\text{-}ok\ ys$
and *cross*: $ys \neq [] \Longrightarrow$
$\forall\, b{\in}set\ xs.\ \forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\ b).$
$snd\ p \leq bp\text{-}marker\ (hd\ ys)$
shows $bp\text{-}bucket\text{-}boundaries\text{-}ok\ (xs\ @\ ys)$
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-bucketize-append-Cons*:
assumes *M-pos*: $0 < M$
and *tail*: $bp\text{-}bucket\text{-}boundaries\text{-}ok\ (c\ \#\ bs)$
and *upper*: $\forall\, q{\in}set\ xs.\ snd\ q \leq bp\text{-}marker\ c$
shows $bp\text{-}bucket\text{-}boundaries\text{-}ok\ (bp\text{-}bucketize\text{-}entries\ M\ xs\ @\ c\ \#\ bs)$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-nonempty*:
assumes *M-pos*: $0 < M$
shows $bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b \neq []$
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-nonempty*:
assumes *M-pos*: $0 < M$
shows $bp\text{-}lazy\text{-}insert\text{-}bucket\ M\ p\ bs \neq []$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-markers-sorted*:
assumes *M-pos*: $0 < M$
shows $sorted\text{-}wrt\ (\lambda b\ c.\ bp\text{-}marker\ b \leq bp\text{-}marker\ c)$
$(bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b)$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-markers-ge*:
fixes *alpha* :: *real*
assumes *M-pos*: $0 < M$
and *lower*: $\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ b).\ bp\text{-}marker\ b \leq snd\ q$
and *p-ge*: $bp\text{-}marker\ b \leq snd\ p$
and *alpha-le*: $alpha \leq bp\text{-}marker\ b$
shows $\forall\, c{\in}set\ (bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b).$
$alpha \leq bp\text{-}marker\ c$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-markers-le*:
fixes *beta* :: *real*
assumes *M-pos*: $0 < M$
and *marker-le*: $bp\text{-}marker\ b \leq beta$
and *upper*: $\forall\, q{\in}set\ (p \#\ bp\text{-}bucket\text{-}entries\ b).\ snd\ q \leq beta$
shows $\forall\, c{\in}set\ (bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b).$
$bp\text{-}marker\ c \leq beta$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-entries-le*:
fixes *beta* :: *real*
assumes *M-pos*: $0 < M$
and *upper*: $\forall\, q{\in}set\ (p \#\ bp\text{-}bucket\text{-}entries\ b).\ snd\ q \leq beta$
shows $\forall\, c{\in}set\ (bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b).$
$\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ c).\ snd\ q \leq beta$
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-boundaries-ok*:
assumes *M-pos*: $0 < M$
shows $bp\text{-}bucket\text{-}boundaries\text{-}ok\ (bp\text{-}lazy\text{-}bucket\text{-}insert\text{-}entries\ M\ p\ b)$
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-markers-ge-hd*:
assumes *M-pos*: $0 < M$
and *sorted*: $sorted\text{-}wrt\ (\lambda b\ c.\ bp\text{-}marker\ b \leq bp\text{-}marker\ c)\ bs$
and *lower*: $\forall\, b{\in}set\ bs.\ \forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ b).$
$bp\text{-}marker\ b \leq snd\ q$
and *nonempty*: $bs \neq []$
and *p-ge*: $bp\text{-}marker\ (hd\ bs) \leq snd\ p$
shows $\forall\, b{\in}set\ (bp\text{-}lazy\text{-}insert\text{-}bucket\ M\ p\ bs).$
$bp\text{-}marker\ (hd\ bs) \leq bp\text{-}marker\ b$
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-markers-sorted*:
assumes *M-pos*: $0 < M$
and *sorted*: $sorted\text{-}wrt\ (\lambda b\ c.\ bp\text{-}marker\ b \leq bp\text{-}marker\ c)\ bs$
and *lower*: $\forall\, b{\in}set\ bs.\ \forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\ b).$
$bp\text{-}marker\ b \leq snd\ q$

and *boundaries*: *bp-bucket-boundaries-ok bs*
shows *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c)
(*bp-lazy-insert-bucket M p bs*)
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-boundaries-ok*:
assumes *M-pos*: $0 < M$
and *sorted*: *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *bs*
and *lower*: $\forall\, b{\in}$*set bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*).
bp-marker b $\leq$ *snd q*
and *boundaries*: *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*bp-lazy-insert-bucket M p bs*)
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-preserves-bucket-shape*:
assumes *M-pos*: $0 < M$
and *sizes*: $\forall\, b{\in}$*set bs*. *length* (*bp-bucket-entries b*) $\leq 2 * M$
and *sorted*: *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *bs*
and *lower*: $\forall\, b{\in}$*set bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*).
bp-marker b $\leq$ *snd q*
and *boundaries*: *bp-bucket-boundaries-ok bs*
shows ($\forall\, b{\in}$*set* (*bp-lazy-insert-bucket M p bs*).
length (*bp-bucket-entries b*) $\leq 2 * M$) $\wedge$
sorted-wrt ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c)
(*bp-lazy-insert-bucket M p bs*) $\wedge$
($\forall\, b{\in}$*set* (*bp-lazy-insert-bucket M p bs*).
$\forall\, q{\in}$*set* (*bp-bucket-entries b*). *bp-marker b* $\leq$ *snd q*) $\wedge$
bp-bucket-boundaries-ok (*bp-lazy-insert-bucket M p bs*)
⟨*proof*⟩

lemma *bp-local-insert-bucket-markers-ge-hd*:
assumes *M-pos*: $0 < M$
and *sorted*: *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *bs*
and *lower*: $\forall\, b{\in}$*set bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*).
bp-marker b $\leq$ *snd q*
and *nonempty*: *bs* $\neq$ []
and *p-ge*: *bp-marker* (*hd bs*) $\leq$ *snd p*
shows $\forall\, b{\in}$*set* (*bp-local-insert-bucket M p bs*).
bp-marker (*hd bs*) $\leq$ *bp-marker b*
⟨*proof*⟩

lemma *bp-local-insert-bucket-markers-sorted*:
assumes *M-pos*: $0 < M$
and *sorted*: *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *bs*
and *lower*: $\forall\, b{\in}$*set bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*).
bp-marker b $\leq$ *snd q*
and *boundaries*: *bp-bucket-boundaries-ok bs*
shows *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c)
(*bp-local-insert-bucket M p bs*)

⟨*proof*⟩

lemma *bp-local-insert-bucket-boundaries-ok*:
assumes *M-pos*: $0 < M$
and *sorted*: *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *bs*
and *lower*: $\forall\, b{\in}$*set bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*).
bp-marker b $\leq$ *snd q*
and *boundaries*: *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*bp-local-insert-bucket M p bs*)
⟨*proof*⟩

lemma *bp-local-insert-bucket-preserves-bucket-shape*:
assumes *M-pos*: $0 < M$
and *sizes*: $\forall\, b{\in}$*set bs*. *length* (*bp-bucket-entries b*) $\leq M$
and *sorted*: *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *bs*
and *lower*: $\forall\, b{\in}$*set bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*).
bp-marker b $\leq$ *snd q*
and *boundaries*: *bp-bucket-boundaries-ok bs*
shows ($\forall\, b{\in}$*set* (*bp-local-insert-bucket M p bs*).
length (*bp-bucket-entries b*) $\leq M$) $\wedge$
sorted-wrt ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c)
(*bp-local-insert-bucket M p bs*) $\wedge$
($\forall\, b{\in}$*set* (*bp-local-insert-bucket M p bs*).
$\forall\, q{\in}$*set* (*bp-bucket-entries b*). *bp-marker b* $\leq$ *snd q*) $\wedge$
bp-bucket-boundaries-ok (*bp-local-insert-bucket M p bs*)
⟨*proof*⟩

The local-insert proof is deliberately split into many list-level lemmas because each invariant component has a different reason for being preserved. Size preservation depends on re-chunking only the affected bucket. Marker ordering depends on inserting into the unique marker interval selected by the value of the new entry. Boundary preservation depends on the fact that every entry of the previous bucket remains below the next bucket marker. The bundled lemma ⟦*0 < ?M*; $\forall\, b{\in}$*set ?bs*. *length* (*bp-bucket-entries b*) $\leq$ *?M*; *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) *?bs*; $\forall\, b{\in}$*set ?bs*. $\forall\, q{\in}$*set* (*bp-bucket-entries b*). *bp-marker b* $\leq$ *snd q*; *bp-bucket-boundaries-ok ?bs*⟧ $\Longrightarrow$ ($\forall\, b{\in}$*set* (*bp-local-insert-bucket ?M ?p ?bs*). *length* (*bp-bucket-entries b*) $\leq$ *?M*) $\wedge$ *sorted-wrt* ($\lambda b\ c.$ *bp-marker* $b \leq$ *bp-marker* c) (*bp-local-insert-bucket ?M ?p ?bs*) $\wedge$ ($\forall\, b{\in}$*set* (*bp-local-insert-bucket ?M ?p ?bs*). $\forall\, q{\in}$*set* (*bp-bucket-entries b*). *bp-marker b* $\leq$ *snd q*) $\wedge$ *bp-bucket-boundaries-ok* (*bp-local-insert-bucket ?M ?p ?bs*) is the high-level summary used by the state-level Insert proof.

lemma *length-bp-bucket-entries-flat-bucketize-entries* [*simp*]:
assumes $0 < M$
shows *length* (*bp-bucket-entries-flat* (*bp-bucketize-entries M xs*)) =
length xs
⟨*proof*⟩

lemma *length-bp-bucketize-sorted-entries-aux-le-ratio*:
assumes *M-pos*: *0 < M*
and *fuel*: *length xs ≤ fuel*
shows *length* (*bp-bucketize-sorted-entries-aux fuel M xs*) ≤
Suc (*length xs div M*)
⟨*proof*⟩

lemma *length-bp-bucketize-sorted-entries-le-ratio*:
assumes *0 < M*
shows *length* (*bp-bucketize-sorted-entries M xs*) ≤
Suc (*length xs div M*)
⟨*proof*⟩

lemma *length-bp-bucketize-entries-le-ratio*:
assumes *0 < M*
shows *length* (*bp-bucketize-entries M xs*) ≤
Suc (*length xs div M*)
⟨*proof*⟩

lemma *length-bp-bucketize-entries-singleton* [*simp*]:
assumes *0 < M*
shows *length* (*bp-bucketize-entries M* [*p*]) = *1*
⟨*proof*⟩

lemma *length-bp-bucketize-sorted-entries-aux-le-length*:
assumes *M-pos*: *0 < M*
and *fuel*: *length xs ≤ fuel*
shows *length* (*bp-bucketize-sorted-entries-aux fuel M xs*) ≤
length xs
⟨*proof*⟩

lemma *length-bp-bucketize-entries-le-length*:
assumes *0 < M*
shows *length* (*bp-bucketize-entries M xs*) ≤ *length xs*
⟨*proof*⟩

lemma *length-bp-bucketize-entries-le-three*:
assumes *M-pos*: *0 < M*
and *len*: *length xs ≤ Suc* (*2 ∗ M*)
shows *length* (*bp-bucketize-entries M xs*) ≤ *3*
⟨*proof*⟩

lemma *length-bp-lazy-bucket-insert-entries-le-three*:
assumes *M-pos*: *0 < M*
and *size*: *length* (*bp-bucket-entries b*) ≤ *2 ∗ M*
shows *length* (*bp-lazy-bucket-insert-entries M p b*) ≤ *3*
⟨*proof*⟩

lemma *length-bp-lazy-insert-bucket-le*:

assumes *M-pos*: $0 < M$
and *sizes*: $\forall\, b{\in}set\ bs.\ length\ (bp\text{-}bucket\text{-}entries\ b) \leq 2 * M$
shows $length\ (bp\text{-}lazy\text{-}insert\text{-}bucket\ M\ p\ bs) \leq length\ bs + 2$
⟨*proof*⟩

lemma *length-bp-bucket-entries-flat-local-insert-bucket* [*simp*]:
assumes $0 < M$
shows $length\ (bp\text{-}bucket\text{-}entries\text{-}flat\ (bp\text{-}local\text{-}insert\text{-}bucket\ M\ p\ bs)) =$
$Suc\ (length\ (bp\text{-}bucket\text{-}entries\text{-}flat\ bs))$
⟨*proof*⟩

lemma *bp-local-insert-bucket-distinct-keys*:
assumes *M-pos*: $0 < M$
and *distinct*: $distinct\ (map\ fst\ (bp\text{-}bucket\text{-}entries\text{-}flat\ bs))$
and *fresh*: $fst\ p \notin bp\text{-}entry\text{-}keys\ (bp\text{-}bucket\text{-}entries\text{-}flat\ bs)$
shows *distinct*
$(map\ fst\ (bp\text{-}bucket\text{-}entries\text{-}flat\ (bp\text{-}local\text{-}insert\text{-}bucket\ M\ p\ bs)))$
⟨*proof*⟩

lemma *bp-local-insert-bucket-values-consistent*:
assumes *M-pos*: $0 < M$
and *vals*: $\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\text{-}flat\ bs).\ f\ (fst\ q) = snd\ q$
and *fresh*: $fst\ p \notin bp\text{-}entry\text{-}keys\ (bp\text{-}bucket\text{-}entries\text{-}flat\ bs)$
shows $\forall\, q{\in}set\ (bp\text{-}bucket\text{-}entries\text{-}flat\ (bp\text{-}local\text{-}insert\text{-}bucket\ M\ p\ bs)).$
$(f(fst\ p := snd\ p))\ (fst\ q) = snd\ q$
⟨*proof*⟩

lemma *bp-empty-invariant*:
assumes $0 < M$
shows $bp\text{-}invariant\ (bp\text{-}empty\ M\ B)$
⟨*proof*⟩

lemma *bp-empty-boundaries-state-ok* [*simp*]:
$bp\text{-}bucket\text{-}boundaries\text{-}state\text{-}ok\ (bp\text{-}empty\ M\ B)$
⟨*proof*⟩

lemma *bp-empty-ordered-invariant*:
assumes $0 < M$
shows $bp\text{-}ordered\text{-}invariant\ (bp\text{-}empty\ M\ B)$
⟨*proof*⟩

lemma *bp-empty-view* [*simp*]:
$bp\text{-}view\ (bp\text{-}empty\ M\ B) = (\!|keys\text{-}of = \{\},\ value\text{-}of = (\lambda\text{-}.\ B)|\!)$
⟨*proof*⟩

lemma *bp-empty-partition-upper-bound* [*simp*]:
$partition\text{-}upper\text{-}bound\ (bp\text{-}view\ (bp\text{-}empty\ M\ B0))\ B$
⟨*proof*⟩

lemma *set-bp-insert-bucket*:
set (*bp-insert-bucket p bs*) = *insert* (*bp-singleton-bucket p*) (*set bs*)
⟨*proof*⟩

lemma *set-bp-bucket-entries-flat-insert-bucket*:
set (*bp-bucket-entries-flat* (*bp-insert-bucket p bs*)) =
insert p (*set* (*bp-bucket-entries-flat bs*))
⟨*proof*⟩

lemma *length-bp-bucket-entries-flat-insert-bucket* [*simp*]:
length (*bp-bucket-entries-flat* (*bp-insert-bucket p bs*)) =
Suc (*length* (*bp-bucket-entries-flat bs*))
⟨*proof*⟩

lemma *bp-entry-keys-insert-bucket* [*simp*]:
bp-entry-keys (*bp-bucket-entries-flat* (*bp-insert-bucket p bs*)) =
insert (*fst p*) (*bp-entry-keys* (*bp-bucket-entries-flat bs*))
⟨*proof*⟩

lemma *bp-insert-bucket-sizes-ok*:
assumes ∀ *b*∈*set bs*. *length* (*bp-bucket-entries b*) ≤ *M*
and *0* < *M*
shows ∀ *b*∈*set* (*bp-insert-bucket p bs*).
length (*bp-bucket-entries b*) ≤ *M*
⟨*proof*⟩

lemma *bp-insert-bucket-markers-sorted*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*bp-insert-bucket p bs*)
⟨*proof*⟩

lemma *bp-insert-bucket-markers-lower-bound*:
assumes ∀ *b*∈*set bs*. ∀ *p*∈*set* (*bp-bucket-entries b*).
bp-marker b ≤ *snd p*
shows ∀ *b*∈*set* (*bp-insert-bucket p bs*). ∀ *p*∈*set* (*bp-bucket-entries b*).
bp-marker b ≤ *snd p*
⟨*proof*⟩

lemma *bp-insert-bucket-values-consistent*:
assumes ∀ *q*∈*set* (*bp-bucket-entries-flat bs*). *f* (*fst q*) = *snd q*
and *fst p* ∉ *bp-entry-keys* (*bp-bucket-entries-flat bs*)
shows ∀ *q*∈*set* (*bp-bucket-entries-flat* (*bp-insert-bucket p bs*)).
(*f*(*fst p* := *snd p*)) (*fst q*) = *snd q*
⟨*proof*⟩

lemma *bp-insert-bucket-distinct-keys*:
assumes *distinct*: *distinct* (*map fst* (*bp-bucket-entries-flat bs*))
and *fresh*: *fst p* ∉ *bp-entry-keys* (*bp-bucket-entries-flat bs*)

shows *distinct* (*map fst* (*bp-bucket-entries-flat* (*bp-insert-bucket p bs*)))
⟨*proof*⟩

lemma *distinct-map-fst-filter-neq*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*filter* (λ*p*. *fst p* ≠ *x*) *xs*))
⟨*proof*⟩

lemma *distinct-map-fst-filter-notin*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*filter* (λ*p*. *fst p* ∉ *S*) *xs*))
⟨*proof*⟩

lemma *bp-bucket-entries-flat-delete-key*:
bp-bucket-entries-flat (*map* (*bp-delete-key-from-bucket x*) *bs*) =
filter (λ*p*. *fst p* ≠ *x*) (*bp-bucket-entries-flat bs*)
⟨*proof*⟩

lemma *bp-bucket-entries-flat-delete-keys*:
bp-bucket-entries-flat (*map* (*bp-delete-keys-from-bucket S*) *bs*) =
filter (λ*p*. *fst p* ∉ *S*) (*bp-bucket-entries-flat bs*)
⟨*proof*⟩

lemma *bp-entries-delete-key*:
bp-entries (*bp-delete-key x P*) =
filter (λ*p*. *fst p* ≠ *x*) (*bp-entries P*)
⟨*proof*⟩

lemma *bp-entries-delete-keys*:
bp-entries (*bp-delete-keys S P*) =
filter (λ*p*. *fst p* ∉ *S*) (*bp-entries P*)
⟨*proof*⟩

lemma *length-filter-fst-neq-ge-pred*:
assumes *distinct*: *distinct* (*map fst xs*)
shows *length xs* ≤ *Suc* (*length* (*filter* (λ*p*. *fst p* ≠ *x*) *xs*))
⟨*proof*⟩

lemma *length-bp-entries-delete-key-ge-pred*:
assumes *bp-distinct-keys P*
shows *length* (*bp-entries P*) ≤
Suc (*length* (*bp-entries* (*bp-delete-key x P*)))
⟨*proof*⟩

lemma *bp-entry-keys-delete-key* [*simp*]:
bp-entry-keys (*bp-entries* (*bp-delete-key x P*)) =
bp-entry-keys (*bp-entries P*) − {*x*}
⟨*proof*⟩

lemma *bp-entry-keys-delete-keys* [*simp*]:
bp-entry-keys (*bp-entries* (*bp-delete-keys S P*)) =
bp-entry-keys (*bp-entries P*) − *S*
⟨*proof*⟩

lemma *bp-values-delete-key* [*simp*]:
bp-values (*bp-delete-key x P*) = *bp-values P*
⟨*proof*⟩

lemma *bp-values-delete-keys* [*simp*]:
bp-values (*bp-delete-keys S P*) = *bp-values P*
⟨*proof*⟩

lemma *bp-delete-key-markers-sorted-list*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*map* (*bp-delete-key-from-bucket x*) *bs*)
⟨*proof*⟩

lemma *bp-delete-keys-markers-sorted-list*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*map* (*bp-delete-keys-from-bucket S*) *bs*)
⟨*proof*⟩

lemma *bp-delete-key-markers-sorted*:
assumes *bp-bucket-markers-sorted P*
shows *bp-bucket-markers-sorted* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-markers-sorted*:
assumes *bp-bucket-markers-sorted P*
shows *bp-bucket-markers-sorted* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-key-bucket-sizes-ok*:
assumes *bp-bucket-sizes-ok P*
shows *bp-bucket-sizes-ok* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-bucket-sizes-ok*:
assumes *bp-bucket-sizes-ok P*
shows *bp-bucket-sizes-ok* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-key-lazy-bucket-sizes-ok*:
assumes *bp-lazy-bucket-sizes-ok P*
shows *bp-lazy-bucket-sizes-ok* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-lazy-bucket-sizes-ok*:
assumes *bp-lazy-bucket-sizes-ok P*
shows *bp-lazy-bucket-sizes-ok* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-key-markers-lower-bound*:
assumes *bp-bucket-markers-lower-bound P*
shows *bp-bucket-markers-lower-bound* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-markers-lower-bound*:
assumes *bp-bucket-markers-lower-bound P*
shows *bp-bucket-markers-lower-bound* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-key-values-consistent*:
assumes *bp-values-consistent P*
shows *bp-values-consistent* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-values-consistent*:
assumes *bp-values-consistent P*
shows *bp-values-consistent* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-key-distinct-keys*:
assumes *bp-distinct-keys P*
shows *bp-distinct-keys* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-distinct-keys*:
assumes *bp-distinct-keys P*
shows *bp-distinct-keys* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-delete-key-from-bucket*:
assumes *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*map* (*bp-delete-key-from-bucket x*) *bs*)
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-delete-keys-from-bucket*:
assumes *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*map* (*bp-delete-keys-from-bucket S*) *bs*)
⟨*proof*⟩

lemma *bp-delete-key-boundaries-state-ok*:
assumes *bp-bucket-boundaries-state-ok P*
shows *bp-bucket-boundaries-state-ok* (*bp-delete-key x P*)

⟨*proof*⟩

lemma *bp-delete-keys-boundaries-state-ok*:
assumes *bp-bucket-boundaries-state-ok P*
shows *bp-bucket-boundaries-state-ok* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-key-invariant*:
assumes *bp-invariant P*
shows *bp-invariant* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-key-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-key-lazy-invariant*:
assumes *bp-lazy-invariant P*
shows *bp-lazy-invariant* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-key-lazy-ordered-invariant*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant* (*bp-delete-key x P*)
⟨*proof*⟩

lemma *bp-delete-keys-invariant*:
assumes *bp-invariant P*
shows *bp-invariant* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-keys-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-keys-lazy-invariant*:
assumes *bp-lazy-invariant P*
shows *bp-lazy-invariant* (*bp-delete-keys S P*)
⟨*proof*⟩

lemma *bp-delete-keys-view*:
bp-view (*bp-delete-keys S P*) =
(| *keys-of* = *keys-of* (*bp-view P*) − *S*,
value-of = *value-of* (*bp-view P*) |)
⟨*proof*⟩

lemma *bp-entry-keys-insert-bucket-state* [*simp*]:

bp-entry-keys (*bp-entries*
(*P*(|*bp-buckets* := *bp-insert-bucket p* (*bp-buckets P*)|))) =
insert (*fst p*) (*bp-entry-keys* (*bp-entries P*))
⟨*proof*⟩

lemma *bp-entry-keys-insert-bucket-state-values* [*simp*]:
bp-entry-keys (*bp-entries*
(*P*(|*bp-buckets* := *bp-insert-bucket p* (*bp-buckets P*),
bp-values := *f*|))) =
insert (*fst p*) (*bp-entry-keys* (*bp-entries P*))
⟨*proof*⟩

lemma *bp-insert-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant* (*bp-insert x b P*)
⟨*proof*⟩

lemma *bp-insert-keys* [*simp*]:
bp-entry-keys (*bp-entries* (*bp-insert x b P*)) =
insert x (*bp-entry-keys* (*bp-entries P*))
⟨*proof*⟩

lemma *bp-insert-values* [*simp*]:
bp-values (*bp-insert x b P*) =
(*bp-values P*)
(*x* := *if x* ∈ *bp-entry-keys* (*bp-entries P*)
then min (*bp-values P x*) *b else b*)
⟨*proof*⟩

theorem *bp-insert-refines-min-update*:
bp-view (*bp-insert x b P*) = *min-update* (*bp-view P*) *x b*
⟨*proof*⟩

theorem *bp-insert-refines-insert-spec*:
insert-spec (*bp-view P*) *x b* (*bp-view* (*bp-insert x b P*))
⟨*proof*⟩

theorem *bp-insert-preserves-upper-bound*:
assumes *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *b-lt*: *b* < *B*
shows *partition-upper-bound* (*bp-view* (*bp-insert x b P*)) *B*
⟨*proof*⟩

lemma *bp-local-insert-state-keys*:
assumes *M-pos*: *0* < *bp-block-size P*
shows *bp-entry-keys* (*bp-entries* (*bp-local-insert-state x b P*)) =
insert x (*bp-entry-keys* (*bp-entries P*))
⟨*proof*⟩

lemma *bp-local-insert-state-values* [*simp*]:
bp-values (*bp-local-insert-state x b P*) =
(*bp-values P*)
(*x* := *if x* ∈ *bp-entry-keys* (*bp-entries P*)
then min (*bp-values P x*) *b else b*)
⟨*proof*⟩

theorem *bp-local-insert-state-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-local-insert-state x b P*) = *min-update* (*bp-view P*) *x b*
⟨*proof*⟩

theorem *bp-local-insert-state-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-local-insert-state x b P*))
⟨*proof*⟩

theorem *bp-local-insert-state-preserves-upper-bound*:
assumes *M-pos*: *0* < *bp-block-size P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *b-lt*: *b* < *B*
shows *partition-upper-bound* (*bp-view* (*bp-local-insert-state x b P*)) *B*
⟨*proof*⟩

lemma *bp-lazy-insert-state-keys*:
assumes *M-pos*: *0* < *bp-block-size P*
shows *bp-entry-keys* (*bp-entries* (*bp-lazy-insert-state x b P*)) =
insert x (*bp-entry-keys* (*bp-entries P*))
⟨*proof*⟩

lemma *bp-lazy-insert-state-values* [*simp*]:
bp-values (*bp-lazy-insert-state x b P*) =
(*bp-values P*)
(*x* := *if x* ∈ *bp-entry-keys* (*bp-entries P*)
then min (*bp-values P x*) *b else b*)
⟨*proof*⟩

theorem *bp-lazy-insert-state-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-lazy-insert-state x b P*) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

theorem *bp-lazy-insert-state-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-lazy-insert-state x b P*))
⟨*proof*⟩

theorem *bp-lazy-insert-state-preserves-upper-bound*:
assumes *M-pos*: $0 < bp\text{-}block\text{-}size\ P$
and *upper*: $partition\text{-}upper\text{-}bound\ (bp\text{-}view\ P)\ B$
and *b-lt*: $b < B$
shows $partition\text{-}upper\text{-}bound\ (bp\text{-}view\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P))\ B$
⟨*proof*⟩

lemma *bp-lazy-insert-state-lazy-bucket-sizes-ok*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}lazy\text{-}bucket\text{-}sizes\text{-}ok\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-markers-lower-bound*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}bucket\text{-}markers\text{-}lower\text{-}bound\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-values-consistent*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}values\text{-}consistent\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-distinct-keys*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}distinct\text{-}keys\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-markers-sorted*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}bucket\text{-}markers\text{-}sorted\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-boundaries-state-ok*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}bucket\text{-}boundaries\text{-}state\text{-}ok\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-lazy-invariant*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}lazy\text{-}invariant\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *bp-lazy-insert-state-lazy-ordered-invariant*:
assumes *lazy*: $bp\text{-}lazy\text{-}ordered\text{-}invariant\ P$
shows $bp\text{-}lazy\text{-}ordered\text{-}invariant\ (bp\text{-}lazy\text{-}insert\text{-}state\ x\ b\ P)$
⟨*proof*⟩

lemma *length-bp-lazy-insert-state-entries*:

assumes *M-pos*: *0* < *bp-block-size P*
shows *length* (*bp-entries* (*bp-lazy-insert-state x b P*)) =
Suc (*length* (*bp-entries* (*bp-delete-key x P*)))
⟨*proof*⟩

lemma *length-bp-lazy-insert-state-entries-ge*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *length* (*bp-entries P*) ≤
length (*bp-entries* (*bp-lazy-insert-state x b P*))
⟨*proof*⟩

lemma *length-bp-lazy-insert-state-buckets-le*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *length* (*bp-buckets* (*bp-lazy-insert-state x b P*)) ≤
length (*bp-buckets P*) + *2*
⟨*proof*⟩

lemma *bp-local-insert-state-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-invariant* (*bp-local-insert-state x b P*)
⟨*proof*⟩

lemma *bp-local-insert-state-boundaries-state-ok*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-bucket-boundaries-state-ok* (*bp-local-insert-state x b P*)
⟨*proof*⟩

lemma *bp-local-insert-state-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bp-local-insert-state x b P*)
⟨*proof*⟩

lemma *bp-batch-prepend-invariant*:
assumes *bp-invariant P*
shows *bp-invariant* (*bp-batch-prepend xs P*)
⟨*proof*⟩

theorem *bp-batch-prepend-refines-batch-min-update*:
assumes *bp-invariant P*
shows *bp-view* (*bp-batch-prepend xs P*) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

theorem *bp-batch-prepend-preserves-upper-bound*:
assumes *inv*: *bp-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound* (*bp-view* (*bp-batch-prepend xs P*)) *B*
⟨*proof*⟩

definition *bp-rebucketed-insert* ::
'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition **where**
bp-rebucketed-insert x b P = bp-rebucket (bp-insert x b P)

definition *bp-rebucketed-batch-prepend* ::
('k × real) list ⇒ 'k bucketed-partition ⇒
'k bucketed-partition **where**
bp-rebucketed-batch-prepend xs P = bp-rebucket (bp-batch-prepend xs P)

lemma *bp-rebucketed-insert-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant (bp-rebucketed-insert x b P)*
⟨*proof*⟩

lemma *bp-rebucketed-insert-ordered-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-ordered-invariant (bp-rebucketed-insert x b P)*
⟨*proof*⟩

theorem *bp-rebucketed-insert-refines-min-update*:
assumes *inv*: *bp-invariant P*
shows *bp-view (bp-rebucketed-insert x b P) = min-update (bp-view P) x b*
⟨*proof*⟩

theorem *bp-rebucketed-insert-refines-insert-spec*:
assumes *inv*: *bp-invariant P*
shows *insert-spec (bp-view P) x b (bp-view (bp-rebucketed-insert x b P))*
⟨*proof*⟩

theorem *bp-rebucketed-insert-preserves-upper-bound*:
assumes *inv*: *bp-invariant P*
and *upper*: *partition-upper-bound (bp-view P) B*
and *b-lt*: *b < B*
shows *partition-upper-bound (bp-view (bp-rebucketed-insert x b P)) B*
⟨*proof*⟩

lemma *bp-rebucketed-batch-prepend-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant (bp-rebucketed-batch-prepend xs P)*
⟨*proof*⟩

lemma *bp-rebucketed-batch-prepend-ordered-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-ordered-invariant (bp-rebucketed-batch-prepend xs P)*
⟨*proof*⟩

theorem *bp-rebucketed-batch-prepend-refines-batch-min-update*:
assumes *inv*: *bp-invariant P*

shows *bp-view* (*bp-rebucketed-batch-prepend xs P*) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

theorem *bp-rebucketed-batch-prepend-preserves-upper-bound*:
assumes *inv*: *bp-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view* (*bp-rebucketed-batch-prepend xs P*)) *B*
⟨*proof*⟩

lemma *bp-min-value-le-entry*:
assumes (*x*, *b*) ∈ *set xs*
shows *bp-min-value B xs* ≤ *b*
⟨*proof*⟩

lemma *bp-entries-Cons*:
assumes *bp-buckets P* = *b* # *bs*
shows *bp-entries P* = *bp-bucket-entries b* @ *bp-bucket-entries-flat bs*
⟨*proof*⟩

lemma *bp-tail-entry-ge-head-marker*:
assumes *sorted*: *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) (*c* # *bs*)
and *lower*: ⋀*d p*. ⟦*d* ∈ *set* (*c* # *bs*); *p* ∈ *set* (*bp-bucket-entries d*)⟧
⟹ *bp-marker d* ≤ *snd p*
and *p*: *p* ∈ *set* (*bp-bucket-entries-flat* (*c* # *bs*))
shows *bp-marker c* ≤ *snd p*
⟨*proof*⟩

lemma *bp-first-bucket-pull-invariant*:
assumes *inv*: *bp-invariant P*
and *pull*: *bp-first-bucket-pull M B P* = (*S*, *beta*, *P*′)
shows *bp-invariant P*′
⟨*proof*⟩

theorem *bp-first-bucket-pull-refines-pull-separates*:
assumes *inv*: *bp-invariant P*
and *buckets*: *bp-buckets P* = *b* # *c* # *bs*
and *len-b*: *length* (*bp-bucket-entries b*) ≤ *M*
and *below*: *bp-bucket-below-bound b* (*bp-marker c*)
and *tail-nonempty*: *bp-bucket-entries-flat* (*c* # *bs*) ≠ []
and *pull*: *bp-first-bucket-pull M B P* = (*S*, *beta*, *P*′)
shows *pull-separates* (*bp-view P*) *M B S beta* (*bp-view P*′)
⟨*proof*⟩

lemma *bp-conservative-pull-invariant*:
assumes *inv*: *bp-invariant P*
and *pull*: *bp-conservative-pull M B P* = (*S*, *beta*, *P*′)

shows *bp-invariant* P'
⟨*proof*⟩

theorem *bp-conservative-pull-refines-pull-separates*:
assumes *inv*: *bp-invariant* P
and *upper*: $\bigwedge u.\ u \in$ *keys-of* (*bp-view* P) $\Longrightarrow$ *value-of* (*bp-view* P) $u < B$
and *pull*: *bp-conservative-pull* $M\ B\ P = (S,\ beta,\ P')$
shows *pull-separates* (*bp-view* P) $M\ B\ S\ beta$ (*bp-view* P')
⟨*proof*⟩

lemma *bp-pull-invariant*:
assumes *inv*: *bp-invariant* P
and *pull*: *bp-pull* $M\ B\ P = (S,\ beta,\ P')$
shows *bp-invariant* P'
⟨*proof*⟩

lemma *bp-first-bucket-pull-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant* P
and *pull*: *bp-first-bucket-pull* $M\ B\ P = (S,\ beta,\ P')$
shows *bp-ordered-invariant* P'
⟨*proof*⟩

lemma *bp-conservative-pull-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant* P
and *pull*: *bp-conservative-pull* $M\ B\ P = (S,\ beta,\ P')$
shows *bp-ordered-invariant* P'
⟨*proof*⟩

lemma *bp-pull-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant* P
and *pull*: *bp-pull* $M\ B\ P = (S,\ beta,\ P')$
shows *bp-ordered-invariant* P'
⟨*proof*⟩

theorem *bp-pull-refines-pull-separates*:
assumes *inv*: *bp-invariant* P
and *upper*: $\bigwedge u.\ u \in$ *keys-of* (*bp-view* P) $\Longrightarrow$ *value-of* (*bp-view* P) $u < B$
and *pull*: *bp-pull* $M\ B\ P = (S,\ beta,\ P')$
shows *pull-separates* (*bp-view* P) $M\ B\ S\ beta$ (*bp-view* P')
⟨*proof*⟩

theorem *bp-pull-preserves-upper-bound*:
assumes *inv*: *bp-invariant* P
and *pull-upper*: *partition-upper-bound* (*bp-view* P) *Bpull*
and *upper*: *partition-upper-bound* (*bp-view* P) B
and *pull*: *bp-pull* M *Bpull* $P = (S,\ beta,\ P')$
shows *partition-upper-bound* (*bp-view* P') B

⟨*proof*⟩

The preceding block completes the functional refinement layer. Insert and BatchPrepend update the view by minimum-value update, while Pull satisfies *pull-separates*: it returns at most the requested block, removes those keys, and establishes the returned boundary between pulled and remaining entries. The implementation attempts a cheap first-bucket pull when the ordered boundary proves it is safe; otherwise it uses the conservative pull specification. Both branches preserve *bp-invariant* and the abstract upper-bound condition.

14 Cost Budgets for the Bucketed Structure

The cost layer counts primitive functional steps by pairing each operation result with a natural number. The budget definitions isolate the paper-critical logarithmic search term as a function of N / M, where N is represented by an entry-list length and M by the block size. The potential terms record the credits that pay for lazy bucket splitting and for restoring the bucket-count ratio after a rebuild.

definition *bp-steps-of* :: $'a \times nat \Rightarrow nat$ **where**
  bp-steps-of r = *snd r*

definition *bp-result-of* :: $'a \times nat \Rightarrow 'a$ **where**
  bp-result-of r = *fst r*

lemma *bp-steps-of-pair* [*simp*]:
  bp-steps-of (*x*, *c*) = *c*
  ⟨*proof*⟩

lemma *bp-result-of-pair* [*simp*]:
  bp-result-of (*x*, *c*) = *x*
  ⟨*proof*⟩

lemma *bp-result-of-add-cost-case* [*simp*]:
  bp-result-of (*case r of* (*x*, *c'*) ⇒ (*x*, *c* + *c'*)) = *bp-result-of r*
  ⟨*proof*⟩

lemma *bp-steps-of-add-cost-case* [*simp*]:
  bp-steps-of (*case r of* (*x*, *c'*) ⇒ (*x*, *c* + *c'*)) =
    c + *bp-steps-of r*
  ⟨*proof*⟩

definition *bp-ratio-log-budget* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  bp-ratio-log-budget N M = *Suc* (*floor-log* (*Suc* (*N div M*)))

definition *bp-insert-search-budget* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  bp-insert-search-budget N M = *Suc* (*bp-ratio-log-budget N M*)

definition *bp-lazy-insert-amortized-budget* ::
'k bucketed-partition ⇒ *nat* **where**
bp-lazy-insert-amortized-budget P =
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) + 7

definition *bp-batch-prepend-log-budget* :: (*'k* × *real*) *list* ⇒ *nat* ⇒ *nat* **where**
bp-batch-prepend-log-budget xs M =
length xs ∗ *bp-ratio-log-budget* (*length xs*) *M*

definition *bp-batch-prepend-per-item-budget* ::
(*'k* × *real*) *list* ⇒ *nat* ⇒ *nat* **where**
bp-batch-prepend-per-item-budget xs M =
Suc (*bp-ratio-log-budget* (*length xs*) *M*)

definition *bp-batch-prepend-amortized-budget* ::
(*'k* × *real*) *list* ⇒ *nat* ⇒ *nat* **where**
bp-batch-prepend-amortized-budget xs M =
length xs + *bp-batch-prepend-log-budget xs M*

definition *bp-pull-amortized-budget* :: *nat* ⇒ *nat* **where**
bp-pull-amortized-budget M = 2 ∗ *M*

definition *bp-local-split-budget* :: *nat* ⇒ *nat* **where**
bp-local-split-budget M = *Suc* (2 ∗ *Suc M*)

definition *bp-lazy-split-budget* :: *nat* ⇒ *nat* **where**
bp-lazy-split-budget M = 5 + 4 ∗ *M*

definition *bp-bucket-overflow* :: *nat* ⇒ *'k bp-bucket* ⇒ *nat* **where**
bp-bucket-overflow M b = *length* (*bp-bucket-entries b*) − *M*

definition *bp-bucket-overflow-sum* ::
nat ⇒ *'k bp-bucket list* ⇒ *nat* **where**
bp-bucket-overflow-sum M bs =
sum-list (*map* (*bp-bucket-overflow M*) *bs*)

definition *bp-overflow-potential* :: *'k bucketed-partition* ⇒ *nat* **where**
bp-overflow-potential P =
sum-list (*map* (*bp-bucket-overflow* (*bp-block-size P*)) (*bp-buckets P*))

definition *bp-bucket-count-slack* :: *'k bucketed-partition* ⇒ *nat* **where**
bp-bucket-count-slack P =
length (*bp-buckets P*) −
Suc (*length* (*bp-entries P*) *div bp-block-size P*)

definition *bp-split-potential* :: *'k bucketed-partition* ⇒ *nat* **where**
bp-split-potential P =
4 ∗ *bp-overflow-potential P* + *bp-bucket-count-slack P*

definition *bp-amortized-step-bound* ::
nat ⇒ *'k bucketed-partition* ⇒ *'k bucketed-partition* ⇒ *nat* ⇒ *bool* **where**
bp-amortized-step-bound c P P' t ⟷
c + *bp-split-potential P'* ≤ *t* + *bp-split-potential P*

definition *bp-pull-state-of* ::
'k set × *real* × *'k bucketed-partition* ⇒ *'k bucketed-partition* **where**
bp-pull-state-of R = (*case R of* (-, -, *P'*) ⇒ *P'*)

definition *bp-pull-amortized-step-bound* ::
nat ⇒ *'k bucketed-partition* ⇒
(*'k set* × *real* × *'k bucketed-partition*) ⇒ *nat* ⇒ *bool* **where**
bp-pull-amortized-step-bound c P R t ⟷
c + *bp-split-potential* (*bp-pull-state-of R*) ≤
t + *bp-split-potential P*

definition *bp-bucket-count-ratio-ok* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-bucket-count-ratio-ok P ⟷
0 < *bp-block-size P* ∧
length (*bp-buckets P*) ≤
Suc (*length* (*bp-entries P*) *div bp-block-size P*)

definition *bp-regular-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-regular-invariant P ⟷
bp-ordered-invariant P ∧ *bp-bucket-count-ratio-ok P*

definition *bp-bucket-count-loaded* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-bucket-count-loaded P ⟷
bp-block-size P ∗ (*length* (*bp-buckets P*) − *1*) ≤
length (*bp-entries P*)

definition *bp-bucket-search-steps* :: *'k bucketed-partition* ⇒ *nat* **where**
bp-bucket-search-steps P = *Suc* (*floor-log* (*length* (*bp-buckets P*)))

fun *bp-halving-search-steps* :: *nat* ⇒ *nat* **where**
[*simp del*]: *bp-halving-search-steps n* =
(*if n* < *2 then 1 else Suc* (*bp-halving-search-steps* (*n div 2*)))

definition *bp-local-insert-search-charge* :: *'k bucketed-partition* ⇒ *nat* **where**
bp-local-insert-search-charge P = *Suc* (*bp-bucket-search-steps P*)

definition *c-bp-bucket-directory-search* ::
'k bp-bucket list ⇒ *real* ⇒ *unit* × *nat* **where**
c-bp-bucket-directory-search bs b =
((), *bp-halving-search-steps* (*length bs*))

fun *c-bp-bucketize-sorted-entries-aux* ::
nat ⇒ *nat* ⇒ (*'k* × *real*) *list* ⇒ *'k bp-bucket list* × *nat* **where**
c-bp-bucketize-sorted-entries-aux 0 M xs = ([], *0*)

```
| c-bp-bucketize-sorted-entries-aux (Suc fuel) M xs =
    (if M = 0 ∨ xs = []
     then ([], 0)
     else (let (bs, c) =
             c-bp-bucketize-sorted-entries-aux fuel M (drop M xs)
           in (bp-make-bucket (take M xs) # bs,
               Suc (length (take M xs) + c))))

definition c-bp-bucketize-sorted-entries ::
  nat ⇒ ('k × real) list ⇒ 'k bp-bucket list × nat where
  c-bp-bucketize-sorted-entries M xs =
    c-bp-bucketize-sorted-entries-aux (length xs) M xs

definition c-bp-bucketize-entries ::
  nat ⇒ ('k × real) list ⇒ 'k bp-bucket list × nat where
  c-bp-bucketize-entries M xs =
    c-bp-bucketize-sorted-entries M (sort-key snd xs)

definition c-bp-rebucket-build ::
  'k bucketed-partition ⇒ 'k bucketed-partition × nat where
  c-bp-rebucket-build P =
    (case c-bp-bucketize-entries (bp-block-size P) (bp-entries P) of
      (bs, c) ⇒ (P(|bp-buckets := bs|), c))
```

The definitions above introduce the amortized accounting used throughout the cost proof. *bp-ratio-log-budget* is the formal ratio-log term: it applies *floor-log* to *Suc* (*N div M*), avoiding division by zero corner cases at the statement level. The actual bucket-directory search is modeled by *bp-halving-search-steps*, and *bp-bucket-search-steps* records the resulting cost for a state.

The potential *bp-split-potential* has two components. The weighted overflow component *bp-overflow-potential* counts entries above the strict block size, multiplied by four in the combined potential. The *bp-bucket-count-slack* component measures how far the bucket count is above the target ratio. A regular state, captured by *bp-regular-invariant*, has no such debt: it is ordered and satisfies *bp-bucket-count-ratio-ok*. The amortized predicates *bp-amortized-step-bound* and *bp-pull-amortized-step-bound* express the usual inequality *actual* + *Phi*(*after*) <= *budget* + *Phi*(*before*).

```
definition c-bp-local-insert-search ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-local-insert-search x b P =
    (bp-local-insert-state x b P, bp-local-insert-search-charge P)

definition c-bp-local-insert-split ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-local-insert-split x b P =
    (bp-local-insert-state x b P,
```

```
    bp-local-insert-search-charge P +
      bp-local-split-budget (bp-block-size P))

definition c-bp-lazy-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-lazy-insert x b P =
    (bp-lazy-insert-state x b P,
     bp-local-insert-search-charge P +
       bp-lazy-split-budget (bp-block-size P))

definition bp-lazy-bucket-insert-charge ::
  nat ⇒ 'k bp-bucket ⇒ nat where
  bp-lazy-bucket-insert-charge M b =
    (if length (bp-bucket-entries b) < 2 * M
     then 1 else bp-lazy-split-budget M)

fun bp-lazy-insert-bucket-charge ::
  nat ⇒ 'k × real ⇒ 'k bp-bucket list ⇒ nat where
  bp-lazy-insert-bucket-charge M p [] = 1
| bp-lazy-insert-bucket-charge M p [b] =
    (if snd p < bp-marker b then 1
     else bp-lazy-bucket-insert-charge M b)
| bp-lazy-insert-bucket-charge M p (b # c # bs) =
    (if snd p < bp-marker b then 1
     else if snd p < bp-marker c
       then bp-lazy-bucket-insert-charge M b
       else bp-lazy-insert-bucket-charge M p (c # bs))

definition bp-lazy-insert-charge ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ nat where
  bp-lazy-insert-charge x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in bp-lazy-insert-bucket-charge (bp-block-size P0) (x, b')
          (bp-buckets P0))

definition c-bp-lazy-insert-precise ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-lazy-insert-precise x b P =
    (bp-lazy-insert-state x b P,
     bp-local-insert-search-charge P + bp-lazy-insert-charge x b P)

fun bp-lazy-batch-prepend-state ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition where
  bp-lazy-batch-prepend-state [] P = P
```

```
| bp-lazy-batch-prepend-state ((x, b) # xs) P =
    bp-lazy-batch-prepend-state xs (bp-lazy-insert-state x b P)

fun c-bp-lazy-batch-prepend-precise ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-lazy-batch-prepend-precise [] P = (P, 0)
| c-bp-lazy-batch-prepend-precise ((x, b) # xs) P =
    (case c-bp-lazy-insert-precise x b P of
      (P1, c1) ⇒
        (case c-bp-lazy-batch-prepend-precise xs P1 of
          (P2, c2) ⇒ (P2, c1 + c2)))

fun bp-lazy-batch-ratio-ok ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒ bool where
  bp-lazy-batch-ratio-ok [] P ⟷ True
| bp-lazy-batch-ratio-ok ((x, b) # xs) P ⟷
    bp-bucket-count-ratio-ok P ∧
    bp-lazy-batch-ratio-ok xs
      (bp-result-of (c-bp-lazy-insert-precise x b P))

fun bp-lazy-batch-insert-budget-sum ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒ nat where
  bp-lazy-batch-insert-budget-sum [] P = 0
| bp-lazy-batch-insert-budget-sum ((x, b) # xs) P =
    bp-lazy-insert-amortized-budget P +
    bp-lazy-batch-insert-budget-sum xs
      (bp-result-of (c-bp-lazy-insert-precise x b P))

lemma c-bp-lazy-insert-precise-eq [simp]:
  c-bp-lazy-insert-precise x b P =
    (bp-lazy-insert-state x b P,
     bp-local-insert-search-charge P + bp-lazy-insert-charge x b P)
  ⟨proof⟩

definition bp-regularized-local-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-regularized-local-insert x b P =
    bp-rebucket (bp-local-insert-state x b P)

definition c-bp-regularized-local-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-regularized-local-insert x b P =
    (bp-regularized-local-insert x b P, bp-local-insert-search-charge P)
```

This wrapper counts the concrete rebuilding pass explicitly. It is an accounting bridge for the later lazy-split analysis, not the final insert operation bound: rebuilding every bucket has a linear term.

definition *c-bp-regularized-local-insert-build* ::
'k ⇒ real ⇒ 'k bucketed-partition ⇒
'k bucketed-partition × nat **where**
c-bp-regularized-local-insert-build x b P =
(case c-bp-rebucket-build (bp-local-insert-state x b P) of
(P', c) ⇒ (P', bp-local-insert-search-charge P + c))

definition *c-bp-rebucketed-batch-prepend-bulk* ::
('k × real) list ⇒ 'k bucketed-partition ⇒
'k bucketed-partition × nat **where**
c-bp-rebucketed-batch-prepend-bulk xs P =
(bp-rebucketed-batch-prepend xs P,
bp-batch-prepend-log-budget xs (bp-block-size P))

definition *c-bp-bucketed-batch-prepend-direct* ::
('k × real) list ⇒ 'k bucketed-partition ⇒
'k bucketed-partition × nat **where**
c-bp-bucketed-batch-prepend-direct xs P =
(bp-bucketed-batch-prepend-state xs P,
bp-batch-prepend-amortized-budget xs (bp-block-size P))

definition *c-bp-bucketed-batch-prepend-direct-actual* ::
('k × real) list ⇒ 'k bucketed-partition ⇒
'k bucketed-partition × nat **where**
c-bp-bucketed-batch-prepend-direct-actual xs P =
(bp-bucketed-batch-prepend-state xs P,
bp-batch-prepend-log-budget xs (bp-block-size P))

definition *c-bp-first-bucket-pull* ::
nat ⇒ real ⇒ 'k bucketed-partition ⇒
('k set × real × 'k bucketed-partition) × nat **where**
c-bp-first-bucket-pull M B P = (bp-first-bucket-pull M B P, M)

definition *c-bp-first-bucket-pull-scan* ::
nat ⇒ real ⇒ 'k bucketed-partition ⇒
('k set × real × 'k bucketed-partition) × nat **where**
c-bp-first-bucket-pull-scan M B P =
(case bp-first-bucket-pull M B P of
(S, beta, P') ⇒ ((S, beta, P'), card S))

lemma *bp-ratio-log-budget-pos* [*simp*]:
0 < bp-ratio-log-budget N M
⟨*proof*⟩

lemma *bp-insert-search-budget-pos* [*simp*]:
0 < bp-insert-search-budget N M
⟨*proof*⟩

lemma *bp-lazy-insert-amortized-budget-ratio*:

bp-lazy-insert-amortized-budget P =
bp-ratio-log-budget (length (bp-entries P)) (bp-block-size P) + 8
⟨*proof*⟩

lemma *bp-batch-prepend-amortized-budget-alt*:
bp-batch-prepend-amortized-budget xs M =
*bp-batch-prepend-per-item-budget xs M * length xs*
⟨*proof*⟩

lemma *bp-batch-prepend-log-budget-le-amortized-budget*:
bp-batch-prepend-log-budget xs M ≤
bp-batch-prepend-amortized-budget xs M
⟨*proof*⟩

lemma *bp-halving-search-steps-eq-floor-log*:
bp-halving-search-steps n = Suc (floor-log n)
⟨*proof*⟩

lemma *c-bp-bucket-directory-search-result* [*simp*]:
bp-result-of (c-bp-bucket-directory-search bs b) = ()
⟨*proof*⟩

lemma *c-bp-bucket-directory-search-steps* [*simp*]:
bp-steps-of (c-bp-bucket-directory-search bs b) =
bp-halving-search-steps (length bs)
⟨*proof*⟩

lemma *c-bp-bucket-directory-search-steps-floor-log*:
bp-steps-of (c-bp-bucket-directory-search bs b) =
Suc (floor-log (length bs))
⟨*proof*⟩

lemma *c-bp-bucket-directory-search-state-steps*:
bp-steps-of (c-bp-bucket-directory-search (bp-buckets P) b) =
bp-bucket-search-steps P
⟨*proof*⟩

lemma *c-bp-bucket-directory-search-ratio-bound*:
assumes *count-le*: *length bs ≤ Suc (N div M)*
shows *bp-steps-of (c-bp-bucket-directory-search bs b) ≤*
bp-ratio-log-budget N M
⟨*proof*⟩

lemma *bp-local-insert-search-charge-search-steps*:
bp-local-insert-search-charge P =
Suc (bp-steps-of
(c-bp-bucket-directory-search (bp-buckets P) b))
⟨*proof*⟩

lemma *c-bp-bucketize-sorted-entries-aux-empty* [*simp*]:
c-bp-bucketize-sorted-entries-aux fuel M [] = ([], *0*)
⟨*proof*⟩

lemma *c-bp-bucketize-sorted-entries-aux-result* [*simp*]:
bp-result-of (*c-bp-bucketize-sorted-entries-aux fuel M xs*) =
bp-bucketize-sorted-entries-aux fuel M xs
⟨*proof*⟩

lemma *c-bp-bucketize-sorted-entries-result* [*simp*]:
bp-result-of (*c-bp-bucketize-sorted-entries M xs*) =
bp-bucketize-sorted-entries M xs
⟨*proof*⟩

lemma *c-bp-bucketize-entries-result* [*simp*]:
bp-result-of (*c-bp-bucketize-entries M xs*) = *bp-bucketize-entries M xs*
⟨*proof*⟩

lemma *c-bp-bucketize-sorted-entries-aux-steps-le*:
assumes *M-pos*: *0* < *M*
and *fuel*: *length xs* ≤ *fuel*
shows *bp-steps-of* (*c-bp-bucketize-sorted-entries-aux fuel M xs*) ≤
length xs + *Suc* (*length xs div M*)
⟨*proof*⟩

lemma *c-bp-bucketize-sorted-entries-steps-le*:
assumes *0* < *M*
shows *bp-steps-of* (*c-bp-bucketize-sorted-entries M xs*) ≤
length xs + *Suc* (*length xs div M*)
⟨*proof*⟩

lemma *c-bp-bucketize-entries-steps-le*:
assumes *0* < *M*
shows *bp-steps-of* (*c-bp-bucketize-entries M xs*) ≤
length xs + *Suc* (*length xs div M*)
⟨*proof*⟩

lemma *c-bp-bucketize-entries-steps-le-linear-length*:
assumes *0* < *M*
shows *bp-steps-of* (*c-bp-bucketize-entries M xs*) ≤
Suc (*2* ∗ *length xs*)
⟨*proof*⟩

lemma *c-bp-bucketize-entries-steps-le-local-split-budget*:
assumes *M-pos*: *0* < *M*
and *len*: *length xs* ≤ *Suc M*
shows *bp-steps-of* (*c-bp-bucketize-entries M xs*) ≤
bp-local-split-budget M
⟨*proof*⟩

lemma *bp-local-split-budget-le-lazy-split-budget*:
bp-local-split-budget M ≤ *bp-lazy-split-budget M*
⟨*proof*⟩

lemma *c-bp-bucketize-entries-steps-le-lazy-split-budget*:
assumes *M-pos*: *0* < *M*
and *len*: *length xs* ≤ *Suc* (*2* ∗ *M*)
shows *bp-steps-of* (*c-bp-bucketize-entries M xs*) ≤
bp-lazy-split-budget M
⟨*proof*⟩

lemma *bp-lazy-split-budget-paid-by-bucket-overflow*:
assumes *large*: *2* ∗ *M* ≤ *length* (*bp-bucket-entries b*)
shows *bp-lazy-split-budget M* ≤
5 + *4* ∗ *bp-bucket-overflow M b*
⟨*proof*⟩

lemma *bp-bucket-overflow-sum-simps* [*simp*]:
bp-bucket-overflow-sum M [] = *0*
bp-bucket-overflow-sum M (*b* # *bs*) =
bp-bucket-overflow M b + *bp-bucket-overflow-sum M bs*
bp-bucket-overflow-sum M (*bs* @ *cs*) =
bp-bucket-overflow-sum M bs + *bp-bucket-overflow-sum M cs*
⟨*proof*⟩

lemma *bp-overflow-potential-alt*:
bp-overflow-potential P =
bp-bucket-overflow-sum (*bp-block-size P*) (*bp-buckets P*)
⟨*proof*⟩

lemma *bp-bucket-overflow-sum-zero-if-sizes-ok*:
assumes *sizes*: ∀ *b*∈*set bs*. *length* (*bp-bucket-entries b*) ≤ *M*
shows *bp-bucket-overflow-sum M bs* = *0*
⟨*proof*⟩

lemma *bp-bucket-overflow-cons-credit*:
fixes *b* :: *'k bp-bucket*
shows *1* + *4* ∗ *bp-bucket-overflow M*
(*b*(|*bp-bucket-entries* := *p* # *bp-bucket-entries b*|)) ≤
5 + *4* ∗ *bp-bucket-overflow M b*
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-entries-weighted-overflow-credit*:
assumes *M-pos*: *0* < *M*
and *size*: *length* (*bp-bucket-entries b*) ≤ *2* ∗ *M*
shows *bp-lazy-bucket-insert-charge M b* +
4 ∗ *bp-bucket-overflow-sum M*
(*bp-lazy-bucket-insert-entries M p b*) ≤

$5 + 4 * \textit{bp-bucket-overflow } M\ b$
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-weighted-overflow-credit*:
assumes *M-pos*: $0 < M$
and *sizes*: $\forall\, b{\in}\textit{set } bs.\ \textit{length } (\textit{bp-bucket-entries } b) \leq 2 * M$
shows $\textit{bp-lazy-insert-bucket-charge } M\ p\ bs +$
$4 * \textit{bp-bucket-overflow-sum } M\ (\textit{bp-lazy-insert-bucket } M\ p\ bs) \leq$
$5 + 4 * \textit{bp-bucket-overflow-sum } M\ bs$
⟨*proof*⟩

lemma *bp-delete-key-from-bucket-overflow-le*:
$\textit{bp-bucket-overflow } M\ (\textit{bp-delete-key-from-bucket } x\ b) \leq$
$\textit{bp-bucket-overflow } M\ b$
⟨*proof*⟩

lemma *bp-bucket-overflow-sum-map-delete-key-le*:
$\textit{bp-bucket-overflow-sum } M\ (\textit{map } (\textit{bp-delete-key-from-bucket } x)\ bs) \leq$
$\textit{bp-bucket-overflow-sum } M\ bs$
⟨*proof*⟩

lemma *bp-delete-key-overflow-potential-le*:
$\textit{bp-overflow-potential } (\textit{bp-delete-key } x\ P) \leq \textit{bp-overflow-potential } P$
⟨*proof*⟩

lemma *bp-delete-keys-from-bucket-overflow-le*:
$\textit{bp-bucket-overflow } M\ (\textit{bp-delete-keys-from-bucket } S\ b) \leq$
$\textit{bp-bucket-overflow } M\ b$
⟨*proof*⟩

lemma *bp-bucket-overflow-sum-map-delete-keys-le*:
$\textit{bp-bucket-overflow-sum } M\ (\textit{map } (\textit{bp-delete-keys-from-bucket } S)\ bs) \leq$
$\textit{bp-bucket-overflow-sum } M\ bs$
⟨*proof*⟩

lemma *bp-delete-keys-overflow-potential-le*:
$\textit{bp-overflow-potential } (\textit{bp-delete-keys } S\ P) \leq \textit{bp-overflow-potential } P$
⟨*proof*⟩

lemma *length-filter-fst-notin-le-card*:
assumes *distinct*: *distinct* (*map fst xs*)
and *finite*: *finite S*
shows $\textit{length } xs \leq \textit{length } (\textit{filter } (\lambda p.\ \textit{fst } p \notin S)\ xs) + \textit{card } S$
⟨*proof*⟩

lemma *div-add-right-le-div-plus*:
fixes *n k M* :: *nat*
assumes *M-pos*: $0 < M$
shows $(n + k)\ \textit{div } M \leq n\ \textit{div } M + k$

⟨*proof*⟩

lemma *div-bound-after-delete*:
  assumes *M-pos*: $0 < M$
    and *len*: $n \leq n' + k$
  shows $Suc\ (n\ div\ M) \leq Suc\ (n'\ div\ M) + k$
⟨*proof*⟩

lemma *nat-diff-le-diff-add-right*:
  fixes $a\ c\ d\ k :: nat$
  assumes $c \leq d + k$
  shows $a - d \leq (a - c) + k$
⟨*proof*⟩

lemma *nat-diff-add-right-le*:
  fixes $a\ b\ c\ d\ k :: nat$
  assumes *a-le*: $a \leq k + b$
    and *d-le-c*: $d \leq c$
  shows $a - c \leq k + (b - d)$
⟨*proof*⟩

lemma *bp-delete-keys-entries-length-le*:
  assumes *distinct*: *bp-distinct-keys P*
    and *finite*: *finite S*
  shows *length* (*bp-entries P*) ≤
    length (*bp-entries* (*bp-delete-keys S P*)) + *card S*
  ⟨*proof*⟩

lemma *bp-delete-keys-bucket-count-slack-le*:
  assumes *M-pos*: $0 <$ *bp-block-size P*
    and *distinct*: *bp-distinct-keys P*
    and *finite*: *finite S*
  shows *bp-bucket-count-slack* (*bp-delete-keys S P*) ≤
    bp-bucket-count-slack P + *card S*
⟨*proof*⟩

lemma *bp-delete-keys-split-potential-le*:
  assumes *M-pos*: $0 <$ *bp-block-size P*
    and *distinct*: *bp-distinct-keys P*
    and *finite*: *finite S*
  shows *bp-split-potential* (*bp-delete-keys S P*) ≤
    bp-split-potential P + *card S*
⟨*proof*⟩

lemma *bp-lazy-insert-state-weighted-overflow-credit-after-delete*:
  assumes *lazy*: *bp-lazy-ordered-invariant P*
  shows *bp-lazy-insert-charge x b P* +
      $4 *$ *bp-overflow-potential* (*bp-lazy-insert-state x b P*) ≤
    $5 + 4 *$ *bp-overflow-potential* (*bp-delete-key x P*)

⟨*proof*⟩

lemma *bp-lazy-insert-state-weighted-overflow-credit*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-insert-charge x b P* +
4 ∗ *bp-overflow-potential* (*bp-lazy-insert-state x b P*) ≤
5 + *4* ∗ *bp-overflow-potential P*
⟨*proof*⟩

lemma *nat-diff-le-diff-add*:
fixes *a b c d k* :: *nat*
assumes *a-le*: *a* ≤ *b* + *k*
and *c-le*: *c* ≤ *d*
shows *a* − *d* ≤ (*b* − *c*) + *k*
⟨*proof*⟩

lemma *bp-lazy-insert-state-bucket-count-slack-le*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-bucket-count-slack* (*bp-lazy-insert-state x b P*) ≤
bp-bucket-count-slack P + *2*
⟨*proof*⟩

lemma *bp-lazy-insert-state-split-potential-credit*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-insert-charge x b P* +
bp-split-potential (*bp-lazy-insert-state x b P*) ≤
7 + *bp-split-potential P*
⟨*proof*⟩

The lazy Insert credit proof is the core amortized argument. A local insert either adds an entry to a bucket below the lazy threshold or splits the bucket into strict chunks. The weighted overflow lemma pays for the expensive case: any actual lazy-insert charge plus four times the post-state overflow is at most a constant plus four times the pre-state overflow. The bucket-count slack can increase by only a small constant. Combining those facts yields *bp-lazy-ordered-invariant ?P* ⟹ *bp-lazy-insert-charge ?x ?b ?P* + *bp-split-potential* (*bp-lazy-insert-state ?x ?b ?P*) ≤ *7* + *bp-split-potential ?P*, which is the precise credit inequality used by the costed Insert operation.

lemma *c-bp-rebucket-build-result* [*simp*]:
bp-result-of (*c-bp-rebucket-build P*) = *bp-rebucket P*
⟨*proof*⟩

lemma *c-bp-rebucket-build-steps-le*:
assumes *inv*: *bp-invariant P*
shows *bp-steps-of* (*c-bp-rebucket-build P*) ≤
length (*bp-entries P*) +
Suc (*length* (*bp-entries P*) *div bp-block-size P*)
⟨*proof*⟩

lemma *bp-bucket-overflow-zero*:
assumes *length* (*bp-bucket-entries b*) ≤ *M*
shows *bp-bucket-overflow M b* = *0*
⟨*proof*⟩

lemma *bp-overflow-potential-zero-if-sizes-ok*:
assumes *bp-bucket-sizes-ok P*
shows *bp-overflow-potential P* = *0*
⟨*proof*⟩

lemma *bp-invariant-overflow-potential-zero*:
assumes *bp-invariant P*
shows *bp-overflow-potential P* = *0*
⟨*proof*⟩

lemma *bp-bucket-count-slack-zero-if-ratio-ok*:
assumes *bp-bucket-count-ratio-ok P*
shows *bp-bucket-count-slack P* = *0*
⟨*proof*⟩

lemma *bp-split-potential-zero-if-regular*:
assumes *inv*: *bp-invariant P*
and *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-split-potential P* = *0*
⟨*proof*⟩

lemma *bp-bucket-search-steps-ratio-bound*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-bucket-search-steps P* ≤
bp-ratio-log-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
⟨*proof*⟩

lemma *bp-bucket-count-ratio-okI-loaded*:
assumes *M-pos*: *0* < *bp-block-size P*
and *loaded*: *bp-bucket-count-loaded P*
shows *bp-bucket-count-ratio-ok P*
⟨*proof*⟩

lemma *bp-empty-bucket-count-ratio-ok*:
assumes *0* < *M*
shows *bp-bucket-count-ratio-ok* (*bp-empty M B*)
⟨*proof*⟩

lemma *bp-regular-invariant-ordered-invariant*:
assumes *bp-regular-invariant P*
shows *bp-ordered-invariant P*
⟨*proof*⟩

lemma *bp-regular-invariant-ratio-ok*:
assumes *bp-regular-invariant P*
shows *bp-bucket-count-ratio-ok P*
⟨*proof*⟩

lemma *bp-empty-regular-invariant*:
assumes *0 < M*
shows *bp-regular-invariant* (*bp-empty M B*)
⟨*proof*⟩

lemma *bp-rebucket-bucket-count-ratio-ok*:
assumes *inv*: *bp-invariant P*
shows *bp-bucket-count-ratio-ok* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-bucket-count-ratio-ok-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-bucket-count-ratio-ok* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-regular-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-rebucket P*) = *0*
⟨*proof*⟩

lemma *bp-rebucket-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bp-rebucket-regular-split-potential-zero-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-split-potential* (*bp-rebucket P*) = *0*
⟨*proof*⟩

lemma *bp-rebucket-regular-invariant-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-regular-invariant* (*bp-rebucket P*)
⟨*proof*⟩

lemma *c-bp-rebucket-build-steps-le-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-steps-of* (*c-bp-rebucket-build P*) ≤
length (*bp-entries P*) +
Suc (*length* (*bp-entries P*) *div bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-rebucket-build-regular-invariant-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*

shows *bp-regular-invariant* (*bp-result-of* (*c-bp-rebucket-build P*))
⟨*proof*⟩

lemma *c-bp-rebucket-build-split-potential-zero-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-split-potential* (*bp-result-of* (*c-bp-rebucket-build P*)) = *0*
⟨*proof*⟩

lemma *c-bp-rebucket-build-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-result-of* (*c-bp-rebucket-build P*))
⟨*proof*⟩

lemma *c-bp-rebucket-build-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-result-of* (*c-bp-rebucket-build P*)) = *0*
⟨*proof*⟩

lemma *bp-rebucketed-insert-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-rebucketed-insert x b P*)
⟨*proof*⟩

lemma *bp-rebucketed-batch-prepend-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-rebucketed-batch-prepend xs P*)
⟨*proof*⟩

lemma *bp-rebucketed-insert-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-rebucketed-insert x b P*) = *0*
⟨*proof*⟩

lemma *bp-rebucketed-batch-prepend-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-rebucketed-batch-prepend xs P*) = *0*
⟨*proof*⟩

lemma *bp-regularized-local-insert-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-invariant* (*bp-regularized-local-insert x b P*)
⟨*proof*⟩

lemma *bp-regularized-local-insert-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bp-regularized-local-insert x b P*)
⟨*proof*⟩

lemma *bp-regularized-local-insert-regular-invariant*:

assumes *ord*: *bp-ordered-invariant P*
shows *bp-regular-invariant* (*bp-regularized-local-insert x b P*)
⟨*proof*⟩

lemma *bp-regularized-local-insert-refines-min-update*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-view* (*bp-regularized-local-insert x b P*) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *bp-regularized-local-insert-refines-insert-spec*:
assumes *ord*: *bp-ordered-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-regularized-local-insert x b P*))
⟨*proof*⟩

lemma *bp-regularized-local-insert-split-potential-zero*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-split-potential* (*bp-regularized-local-insert x b P*) = *0*
⟨*proof*⟩

lemma *bp-local-insert-search-charge-ratio-bound*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-local-insert-search-charge P* ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-local-insert-search-result* [*simp*]:
bp-result-of (*c-bp-local-insert-search x b P*) =
bp-local-insert-state x b P
⟨*proof*⟩

lemma *c-bp-local-insert-search-steps* [*simp*]:
bp-steps-of (*c-bp-local-insert-search x b P*) =
bp-local-insert-search-charge P
⟨*proof*⟩

lemma *c-bp-local-insert-search-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-local-insert-search x b P*)) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-local-insert-search-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-invariant* (*bp-result-of* (*c-bp-local-insert-search x b P*))
⟨*proof*⟩

lemma *c-bp-local-insert-search-ordered-invariant*:

assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-local-insert-search x b P*))
⟨*proof*⟩

lemma *c-bp-local-insert-search-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-local-insert-search x b P*)))
⟨*proof*⟩

lemma *c-bp-local-insert-search-ratio-bound*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-local-insert-search x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-local-insert-search-partition-insert-cost-bound*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-local-insert-search x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-local-insert-split-result* [*simp*]:
bp-result-of (*c-bp-local-insert-split x b P*) =
bp-local-insert-state x b P
⟨*proof*⟩

lemma *c-bp-local-insert-split-steps* [*simp*]:
bp-steps-of (*c-bp-local-insert-split x b P*) =
bp-local-insert-search-charge P +
bp-local-split-budget (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-local-insert-split-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-local-insert-split x b P*)) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-local-insert-split-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-invariant* (*bp-result-of* (*c-bp-local-insert-split x b P*))
⟨*proof*⟩

lemma *c-bp-local-insert-split-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant*

(*bp-result-of* (*c-bp-local-insert-split x b P*))
⟨*proof*⟩

lemma *c-bp-local-insert-split-refines-insert-spec*:
assumes *0 < bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-local-insert-split x b P*)))
⟨*proof*⟩

lemma *c-bp-local-insert-split-steps-le*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-local-insert-split x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +
bp-local-split-budget (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-local-insert-split-steps-le-lazy-budget*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-local-insert-split x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +
bp-lazy-split-budget (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-local-insert-split-amortized-if-split-credit*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
and *paid*:
bp-local-split-budget (*bp-block-size P*) +
bp-split-potential
(*bp-result-of* (*c-bp-local-insert-split x b P*)) ≤
bp-split-potential P
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-local-insert-split x b P*)) *P*
(*bp-result-of* (*c-bp-local-insert-split x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-lazy-insert-result* [*simp*]:
bp-result-of (*c-bp-lazy-insert x b P*) =
bp-lazy-insert-state x b P
⟨*proof*⟩

lemma *c-bp-lazy-insert-steps* [*simp*]:
bp-steps-of (*c-bp-lazy-insert x b P*) =
bp-local-insert-search-charge P +
bp-lazy-split-budget (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-lazy-insert-refines-min-update*:
assumes *0 < bp-block-size P*

shows *bp-view* (*bp-result-of* (*c-bp-lazy-insert x b P*)) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-lazy-insert-lazy-ordered-invariant*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-lazy-insert x b P*))
⟨*proof*⟩

lemma *c-bp-lazy-insert-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-lazy-insert x b P*)))
⟨*proof*⟩

lemma *c-bp-lazy-insert-steps-le-lazy-budget*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-lazy-insert x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +
bp-lazy-split-budget (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-lazy-insert-amortized-if-split-credit*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
and *paid*:
bp-lazy-split-budget (*bp-block-size P*) +
bp-split-potential (*bp-result-of* (*c-bp-lazy-insert x b P*)) ≤
bp-split-potential P
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-lazy-insert x b P*)) *P*
(*bp-result-of* (*c-bp-lazy-insert x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*))
⟨*proof*⟩

lemma *bp-lazy-bucket-insert-charge-le*:
bp-lazy-bucket-insert-charge M b ≤ *bp-lazy-split-budget M*
⟨*proof*⟩

lemma *bp-lazy-insert-bucket-charge-le*:
bp-lazy-insert-bucket-charge M p bs ≤ *bp-lazy-split-budget M*
⟨*proof*⟩

lemma *bp-lazy-insert-charge-le*:
bp-lazy-insert-charge x b P ≤ *bp-lazy-split-budget* (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-result* [*simp*]:
bp-result-of (*c-bp-lazy-insert-precise x b P*) =

bp-lazy-insert-state x b P
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-steps* [*simp*]:
bp-steps-of (*c-bp-lazy-insert-precise x b P*) =
bp-local-insert-search-charge P + *bp-lazy-insert-charge x b P*
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-lazy-insert-precise x b P*)) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-lazy-ordered-invariant*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-lazy-insert-precise x b P*)))
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-steps-le-lazy-budget*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-lazy-insert-precise x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +
bp-lazy-split-budget (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-steps-le-coarse*:
bp-steps-of (*c-bp-lazy-insert-precise x b P*) ≤
bp-steps-of (*c-bp-lazy-insert x b P*)
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-amortized-if-credit*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
and *paid*:
bp-lazy-insert-charge x b P +
bp-split-potential
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*)) ≤
extra + *bp-split-potential P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-lazy-insert-precise x b P*)) *P*
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*) +

extra)
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-amortized-bound*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
and *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-lazy-insert-precise x b P*)) *P*
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*) + 7)
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-amortized-ratio-budget*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
and *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-lazy-insert-precise x b P*)) *P*
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
(*bp-lazy-insert-amortized-budget P*)
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-regular-ratio-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-steps-of* (*c-bp-lazy-insert-precise x b P*) ≤
bp-lazy-insert-amortized-budget P
⟨*proof*⟩

lemma *c-bp-lazy-insert-precise-regular-partition-insert-cost-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-lazy-insert-precise x b P*))
(*bp-lazy-insert-amortized-budget P*)
⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-result* [*simp*]:
bp-result-of (*c-bp-lazy-batch-prepend-precise xs P*) =
bp-lazy-batch-prepend-state xs P
⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-steps-Nil* [*simp*]:
bp-steps-of (*c-bp-lazy-batch-prepend-precise* [] *P*) = *0*
⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-steps-Cons* [*simp*]:
bp-steps-of (*c-bp-lazy-batch-prepend-precise* ((*x*, *b*) # *xs*) *P*) =
bp-steps-of (*c-bp-lazy-insert-precise x b P*) +
bp-steps-of
(*c-bp-lazy-batch-prepend-precise xs*
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*)))

⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-lazy-ordered-invariant*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-lazy-batch-prepend-precise xs P*))
⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-refines-batch-min-update*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-view* (*bp-result-of* (*c-bp-lazy-batch-prepend-precise xs P*)) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-amortized-bound*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
and *ratios*: *bp-lazy-batch-ratio-ok xs P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-lazy-batch-prepend-precise xs P*)) *P*
(*bp-result-of* (*c-bp-lazy-batch-prepend-precise xs P*))
(*bp-lazy-batch-insert-budget-sum xs P*)
⟨*proof*⟩

lemma *c-bp-lazy-batch-prepend-precise-regular-steps-le-budget-sum*:
assumes *reg*: *bp-regular-invariant P*
and *ratios*: *bp-lazy-batch-ratio-ok xs P*
shows *bp-steps-of* (*c-bp-lazy-batch-prepend-precise xs P*) ≤
bp-lazy-batch-insert-budget-sum xs P
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-result* [*simp*]:
bp-result-of (*c-bp-regularized-local-insert x b P*) =
bp-regularized-local-insert x b P
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-steps* [*simp*]:
bp-steps-of (*c-bp-regularized-local-insert x b P*) =
bp-local-insert-search-charge P
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-regular-invariant*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-regular-invariant*
(*bp-result-of* (*c-bp-regularized-local-insert x b P*))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-refines-min-update*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-view* (*bp-result-of* (*c-bp-regularized-local-insert x b P*)) =

min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-refines-insert-spec*:
assumes *reg*: *bp-regular-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-regularized-local-insert x b P*)))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-ratio-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-steps-of* (*c-bp-regularized-local-insert x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-split-potential-zero*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-split-potential*
(*bp-result-of* (*c-bp-regularized-local-insert x b P*)) = *0*
⟨*proof*⟩

lemma *bp-regular-invariant-split-potential-zero*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-split-potential P* = *0*
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-amortized-ratio-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-regularized-local-insert x b P*)) *P*
(*bp-result-of* (*c-bp-regularized-local-insert x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-partition-insert-cost-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-regularized-local-insert x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-result* [*simp*]:
bp-result-of (*c-bp-regularized-local-insert-build x b P*) =
bp-regularized-local-insert x b P
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-steps*:
bp-steps-of (*c-bp-regularized-local-insert-build x b P*) =
bp-local-insert-search-charge P +

bp-steps-of (*c-bp-rebucket-build* (*bp-local-insert-state x b P*))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-regular-invariant*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-regular-invariant*
(*bp-result-of* (*c-bp-regularized-local-insert-build x b P*))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-refines-min-update*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-view* (*bp-result-of*
(*c-bp-regularized-local-insert-build x b P*)) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-refines-insert-spec*:
assumes *reg*: *bp-regular-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of*
(*c-bp-regularized-local-insert-build x b P*)))
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-split-potential-zero*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-split-potential*
(*bp-result-of* (*c-bp-regularized-local-insert-build x b P*)) = *0*
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-rebuild-steps-le*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-steps-of* (*c-bp-regularized-local-insert-build x b P*) ≤
bp-local-insert-search-charge P +
length (*bp-entries* (*bp-local-insert-state x b P*)) +
Suc (*length* (*bp-entries* (*bp-local-insert-state x b P*)) *div*
bp-block-size P)
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-steps-le*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-steps-of* (*c-bp-regularized-local-insert-build x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +
length (*bp-entries* (*bp-local-insert-state x b P*)) +
Suc (*length* (*bp-entries* (*bp-local-insert-state x b P*)) *div*
bp-block-size P)
⟨*proof*⟩

lemma *c-bp-regularized-local-insert-build-amortized-rebuild-bound*:
assumes *reg*: *bp-regular-invariant P*

shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-regularized-local-insert-build x b P*)) *P*
(*bp-result-of* (*c-bp-regularized-local-insert-build x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*) +
length (*bp-entries* (*bp-local-insert-state x b P*)) +
Suc (*length* (*bp-entries* (*bp-local-insert-state x b P*)) *div*
bp-block-size P))
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-result* [*simp*]:
bp-result-of (*c-bp-rebucketed-batch-prepend-bulk xs P*) =
bp-rebucketed-batch-prepend xs P
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-steps* [*simp*]:
bp-steps-of (*c-bp-rebucketed-batch-prepend-bulk xs P*) =
bp-batch-prepend-log-budget xs (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-result* [*simp*]:
bp-result-of (*c-bp-bucketed-batch-prepend-direct xs P*) =
bp-bucketed-batch-prepend-state xs P
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-steps* [*simp*]:
bp-steps-of (*c-bp-bucketed-batch-prepend-direct xs P*) =
bp-batch-prepend-amortized-budget xs (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-result* [*simp*]:
bp-result-of (*c-bp-bucketed-batch-prepend-direct-actual xs P*) =
bp-bucketed-batch-prepend-state xs P
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-steps* [*simp*]:
bp-steps-of (*c-bp-bucketed-batch-prepend-direct-actual xs P*) =
bp-batch-prepend-log-budget xs (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-paper-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
(*bp-batch-prepend-per-item-budget xs* (*bp-block-size P*)) *xs*
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-empty* [*simp*]:
bp-bucketed-batch-prepend-state [] *P* = *P*
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-entries-nonempty*:
assumes *M-pos*: $0 < bp\text{-}block\text{-}size\ P$
and *xs*: $xs \neq []$
shows $bp\text{-}entries\ (bp\text{-}bucketed\text{-}batch\text{-}prepend\text{-}state\ xs\ P) =$
$sort\text{-}key\ snd\ xs\ @\ bp\text{-}entries\ P$
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-keys-nonempty*:
assumes *M-pos*: $0 < bp\text{-}block\text{-}size\ P$
and *xs*: $xs \neq []$
shows $keys\text{-}of\ (bp\text{-}view\ (bp\text{-}bucketed\text{-}batch\text{-}prepend\text{-}state\ xs\ P)) =$
$bp\text{-}entry\text{-}keys\ xs \cup keys\text{-}of\ (bp\text{-}view\ P)$
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-values*:
assumes *distinct*: $distinct\ (map\ fst\ xs)$
and *disjoint*: $bp\text{-}entry\text{-}keys\ xs \cap keys\text{-}of\ (bp\text{-}view\ P) = \{\}$
shows $value\text{-}of\ (bp\text{-}view\ (bp\text{-}bucketed\text{-}batch\text{-}prepend\text{-}state\ xs\ P)) =$
$value\text{-}of\ (batch\text{-}min\text{-}update\ (bp\text{-}view\ P)\ xs)$
⟨*proof*⟩

theorem *bp-bucketed-batch-prepend-state-refines-batch-min-update*:
assumes *inv*: $bp\text{-}invariant\ P$
and *distinct*: $distinct\ (map\ fst\ xs)$
and *disjoint*: $bp\text{-}entry\text{-}keys\ xs \cap keys\text{-}of\ (bp\text{-}view\ P) = \{\}$
shows $bp\text{-}view\ (bp\text{-}bucketed\text{-}batch\text{-}prepend\text{-}state\ xs\ P) =$
$batch\text{-}min\text{-}update\ (bp\text{-}view\ P)\ xs$
⟨*proof*⟩

lemma *bp-rebase-first-bucket-marker-markers-lower-bound*:
assumes *lower*:
$\forall\, b{\in}set\ bs.\ \forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\ b).\ bp\text{-}marker\ b \leq snd\ p$
and *beta-le*: $\forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\text{-}flat\ bs).\ beta \leq snd\ p$
shows $\forall\, b{\in}set\ (bp\text{-}rebase\text{-}first\text{-}bucket\text{-}marker\ beta\ bs).$
$\forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\ b).\ bp\text{-}marker\ b \leq snd\ p$
⟨*proof*⟩

lemma *bp-rebase-first-bucket-marker-markers-sorted*:
assumes *sorted*: $sorted\text{-}wrt\ (\lambda b\ c.\ bp\text{-}marker\ b \leq bp\text{-}marker\ c)\ bs$
and *boundaries*: $bp\text{-}bucket\text{-}boundaries\text{-}ok\ bs$
and *head-nonempty*:
$\bigwedge b\ bs'.\ bp\text{-}drop\text{-}empty\text{-}prefix\ bs = b\ \#\ bs' \Longrightarrow$
$bp\text{-}bucket\text{-}entries\ b \neq []$
and *beta-le*:
$\forall\, p{\in}set\ (bp\text{-}bucket\text{-}entries\text{-}flat\ (bp\text{-}drop\text{-}empty\text{-}prefix\ bs)).$
$beta \leq snd\ p$
shows $sorted\text{-}wrt\ (\lambda b\ c.\ bp\text{-}marker\ b \leq bp\text{-}marker\ c)$
$(bp\text{-}rebase\text{-}first\text{-}bucket\text{-}marker\ beta\ (bp\text{-}drop\text{-}empty\text{-}prefix\ bs))$
⟨*proof*⟩

lemma *bp-batch-max-value-le-old-entry*:
assumes *xs*: $xs = p \# ps$
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
and *inv*: *bp-invariant P*
and *r*: $r \in set\ (bp\text{-}entries\ P)$
shows *bp-batch-max-value* (*snd p*) *ps* $\leq$ *snd r*
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-invariant* (*bp-bucketed-batch-prepend-state xs P*)
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-boundaries-state-ok*:
assumes *ord*: *bp-ordered-invariant P*
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-bucket-boundaries-state-ok*
(*bp-bucketed-batch-prepend-state xs P*)
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant* (*bp-bucketed-batch-prepend-state xs P*)
⟨*proof*⟩

Bucketed BatchPrepend is not implemented as repeated Insert. The incoming batch is first bucketized on its own, then placed before the existing buckets. To preserve the boundary condition, the old prefix of empty buckets is dropped and the first surviving old bucket is rebased to the maximum value of the batch. The admissibility assumption is the abstract promise that every old entry is above the batch range. Together with distinctness and disjointness of keys, this yields both the view refinement ⟦*bp-invariant ?P*; *distinct* (*map fst ?xs*); *bp-entry-keys ?xs* ∩ *keys-of* (*bp-view ?P*) = {}⟧ ⟹ *bp-view* (*bp-bucketed-batch-prepend-state ?xs ?P*) = *batch-min-update* (*bp-view ?P*) *?xs* and the ordered invariant theorem above.

lemma *c-bp-bucketed-batch-prepend-direct-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-invariant*

(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-refines-batch-min-update*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
shows *bp-view* (*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*)) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-preserves-upper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view* (*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*))) *B*
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
(*bp-batch-prepend-per-item-budget xs* (*bp-block-size P*)) *xs*
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-buckets-length-nonempty*:
assumes *M-pos*: *0* < *bp-block-size P*
and *xs*: *xs* ≠ []
shows *length* (*bp-buckets* (*bp-bucketed-batch-prepend-state xs P*)) ≤
length xs + *length* (*bp-buckets P*)
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-bucket-count-slack-le*:
assumes *inv*: *bp-invariant P*
shows *bp-bucket-count-slack* (*bp-bucketed-batch-prepend-state xs P*) ≤
length xs + *bp-bucket-count-slack P*
⟨*proof*⟩

lemma *bp-bucketed-batch-prepend-state-split-potential-le*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-split-potential* (*bp-bucketed-batch-prepend-state xs P*) ≤
length xs + *bp-split-potential P*
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-refines-batch-min-update*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
shows *bp-view*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*)) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-invariant*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*))
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*))
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-preserves-upper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*))) *B*
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-amortized-paper-bound*:
assumes *ord*: *bp-ordered-invariant P*

and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*)) *P*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*))
(*bp-batch-prepend-amortized-budget xs* (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-bucketed-batch-prepend-direct-actual-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-bucketed-batch-prepend-direct-actual xs P*))
(*bp-ratio-log-budget* (*length xs*) (*bp-block-size P*)) *xs*
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-regular-invariant*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-regular-invariant*
(*bp-result-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-refines-batch-min-update*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-view* (*bp-result-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*)) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-preserves-upper-bound*:
assumes *reg*: *bp-regular-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view* (*bp-result-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))) *B*
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-split-potential-zero*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-split-potential*
(*bp-result-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*)) = *0*
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-amortized-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*)) *P*
(*bp-result-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))
(*bp-batch-prepend-log-budget xs* (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-steps-le-amortized-budget*:
bp-steps-of (*c-bp-rebucketed-batch-prepend-bulk xs P*) ≤
bp-batch-prepend-amortized-budget xs (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-amortized-paper-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*)) *P*
(*bp-result-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))
(*bp-batch-prepend-amortized-budget xs* (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))
(*bp-ratio-log-budget* (*length xs*) (*bp-block-size P*)) *xs*
⟨*proof*⟩

lemma *c-bp-rebucketed-batch-prepend-bulk-paper-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))
(*bp-batch-prepend-per-item-budget xs* (*bp-block-size P*)) *xs*
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-result* [*simp*]:
bp-result-of (*c-bp-first-bucket-pull M B P*) =
bp-first-bucket-pull M B P
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-steps* [*simp*]:
bp-steps-of (*c-bp-first-bucket-pull M B P*) = *M*
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-M-bound*:
bp-steps-of (*c-bp-first-bucket-pull M B P*) ≤ *M*
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-result* [*simp*]:
bp-result-of (*c-bp-first-bucket-pull-scan M B P*) =
bp-first-bucket-pull M B P
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-steps*:
assumes *bp-first-bucket-pull M B P* = (*S*, *beta*, *P'*)
shows *bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*) = *card S*
⟨*proof*⟩

lemma *bp-bucket-keys-card-le-length*:

card (*bp-bucket-keys b*) ≤ *length* (*bp-bucket-entries b*)
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-M-bound*:
assumes *can*: *bp-can-first-bucket-pull M P*
shows *bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*) ≤ *M*
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-partition-pull-cost-bound*:
assumes *pull*: *bp-first-bucket-pull M B P* = (*S*, *beta*, *P'*)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *S*
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-partition-pull-cost-bound-from-result*:
assumes *pull*:
bp-result-of (*c-bp-first-bucket-pull-scan M B P*) = (*S*, *beta*, *P'*)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *S*
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant*
(*bp-pull-state-of* (*bp-result-of* (*c-bp-first-bucket-pull-scan M B P*)))
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-amortized-M-bound*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
shows *bp-pull-amortized-step-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *P*
(*bp-result-of* (*c-bp-first-bucket-pull-scan M B P*)) (*2* ∗ *M*)
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-amortized-paper-bound*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
shows *bp-pull-amortized-step-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *P*
(*bp-result-of* (*c-bp-first-bucket-pull-scan M B P*))
(*bp-pull-amortized-budget M*)
⟨*proof*⟩

lemma *c-bp-first-bucket-pull-scan-refines-pull-separates*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
and *pull*:
bp-result-of (*c-bp-first-bucket-pull-scan M B P*) = (*S*, *beta*, *P'*)

shows *pull-separates* (*bp-view P*) *M B S beta* (*bp-view P′*)
⟨*proof*⟩

Pull is charged by the size of the set it returns. The first-bucket policy is sound when *bp-can-first-bucket-pull* exposes two adjacent nonempty ranges and the first bucket lies entirely below the next marker. In that case the returned boundary is the next marker, the deleted set is the key set of the first bucket, and *pull-separates* follows from the bucket boundary invariant. The scan-cost lemmas count exactly the returned keys, so the primitive cost is at most M; the amortized Pull bound additionally allows potential to change after deleting those keys.

14.1 Paper-Facing Costed Operations

The following aliases are the operations exported to the rest of the BMSSP development. They choose the lazy precise Insert, the direct bucketed BatchPrepend, and the first-bucket Pull scan. The aliases keep the public layer small: clients do not need to know which internal costed variant proved the bound, only that the selected operation refines the abstract interface and has the paper budget.

definition *c-bp-paper-insert* ::
  ′k ⇒ *real* ⇒ *′k bucketed-partition* ⇒
    ′k bucketed-partition × *nat* **where**
  c-bp-paper-insert x b P = *c-bp-lazy-insert-precise x b P*

definition *c-bp-paper-batch-prepend* ::
  (*′k* × *real*) *list* ⇒ *′k bucketed-partition* ⇒
    ′k bucketed-partition × *nat* **where**
  c-bp-paper-batch-prepend xs P =
    c-bp-bucketed-batch-prepend-direct-actual xs P

definition *c-bp-paper-pull* ::
  nat ⇒ *real* ⇒ *′k bucketed-partition* ⇒
    (*′k set* × *real* × *′k bucketed-partition*) × *nat* **where**
  c-bp-paper-pull M B P = *c-bp-first-bucket-pull-scan M B P*

definition *bp-insert-paper-budget* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  bp-insert-paper-budget N M =
    9 + *floor-log* (*Suc* (*N div M*))

definition *bp-batch-prepend-paper-budget* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  bp-batch-prepend-paper-budget N M =
    2 + *floor-log* (*Suc* (*N div M*))

definition *bp-pull-paper-budget* :: *nat* ⇒ *nat* **where**
  bp-pull-paper-budget M = *2* ∗ *M*

lemma *bp-lazy-insert-amortized-budget-paper-form*:
bp-lazy-insert-amortized-budget P =
bp-insert-paper-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
⟨*proof*⟩

lemma *bp-batch-prepend-amortized-budget-paper-form*:
bp-batch-prepend-amortized-budget xs M =
length xs ∗ *bp-batch-prepend-paper-budget* (*length xs*) *M*
⟨*proof*⟩

lemma *bp-ratio-log-budget-le-insert-paper-budget*:
bp-ratio-log-budget N M ≤ *bp-insert-paper-budget N M*
⟨*proof*⟩

lemma *bp-ratio-log-budget-le-batch-prepend-paper-budget*:
bp-ratio-log-budget N M ≤ *bp-batch-prepend-paper-budget N M*
⟨*proof*⟩

lemma *bp-insert-paper-budget-bigo-ratio-log*:
(λ*N*. *real* (*bp-insert-paper-budget N M*)) ∈
O(λ*N*. *real* (*bp-ratio-log-budget N M*))
⟨*proof*⟩

lemma *bp-batch-prepend-paper-budget-bigo-ratio-log*:
(λ*N*. *real* (*bp-batch-prepend-paper-budget N M*)) ∈
O(λ*N*. *real* (*bp-ratio-log-budget N M*))
⟨*proof*⟩

lemma *bp-pull-paper-budget-bigo-linear*:
(λ*M*. *real* (*bp-pull-paper-budget M*)) ∈ *O*(λ*M*. *real M*)
⟨*proof*⟩

The named paper budgets are intentionally simple. The Insert budget *bp-insert-paper-budget* is a constant plus the ratio-log term over stored entries and block size. The BatchPrepend budget *bp-batch-prepend-paper-budget* is the corresponding per-item budget; the abstract batch predicate then multiplies it by the batch length. Pull uses *bp-pull-paper-budget*, linear in the requested block size. The Big-O lemmas above record exactly the envelopes consumed by the headline bridge.

lemma *c-bp-paper-insert-result* [*simp*]:
bp-result-of (*c-bp-paper-insert x b P*) =
bp-lazy-insert-state x b P
⟨*proof*⟩

lemma *c-bp-paper-insert-steps* [*simp*]:
bp-steps-of (*c-bp-paper-insert x b P*) =
bp-local-insert-search-charge P + *bp-lazy-insert-charge x b P*
⟨*proof*⟩

lemma *c-bp-paper-insert-refines-min-update*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-view* (*bp-result-of* (*c-bp-paper-insert x b P*)) =
min-update (*bp-view P*) *x b*
⟨*proof*⟩

lemma *c-bp-paper-insert-refines-insert-spec*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-paper-insert x b P*)))
⟨*proof*⟩

lemma *c-bp-paper-insert-lazy-ordered-invariant*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-paper-insert x b P*))
⟨*proof*⟩

lemma *c-bp-paper-insert-amortized-paper-bound*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
and *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-insert x b P*)) *P*
(*bp-result-of* (*c-bp-paper-insert x b P*))
(*bp-lazy-insert-amortized-budget P*)
⟨*proof*⟩

lemma *c-bp-paper-insert-regular-partition-insert-cost-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-paper-insert x b P*))
(*bp-lazy-insert-amortized-budget P*)
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-result* [*simp*]:
bp-result-of (*c-bp-paper-batch-prepend xs P*) =
bp-bucketed-batch-prepend-state xs P
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-steps* [*simp*]:
bp-steps-of (*c-bp-paper-batch-prepend xs P*) =
bp-batch-prepend-log-budget xs (*bp-block-size P*)
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-refines-batch-min-update*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
shows *bp-view* (*bp-result-of* (*c-bp-paper-batch-prepend xs P*)) =

batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-preserves-upper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view* (*bp-result-of* (*c-bp-paper-batch-prepend xs P*))) *B*
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-amortized-paper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-batch-prepend xs P*)) *P*
(*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
(*bp-batch-prepend-amortized-budget xs* (*bp-block-size P*))
⟨*proof*⟩

lemma *c-bp-paper-batch-prepend-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-paper-batch-prepend xs P*))
(*bp-ratio-log-budget* (*length xs*) (*bp-block-size P*)) *xs*
⟨*proof*⟩

lemma *c-bp-paper-pull-result* [*simp*]:
bp-result-of (*c-bp-paper-pull M B P*) =
bp-first-bucket-pull M B P
⟨*proof*⟩

lemma *c-bp-paper-pull-M-bound*:
assumes *can*: *bp-can-first-bucket-pull M P*
shows *bp-steps-of* (*c-bp-paper-pull M B P*) ≤ *M*
⟨*proof*⟩

lemma *c-bp-paper-pull-partition-pull-cost-bound*:
assumes *pull*:
bp-result-of (*c-bp-paper-pull M B P*) = (*S*, *beta*, *P'*)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-paper-pull M B P*)) *S*
⟨*proof*⟩

lemma *c-bp-paper-pull-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant*
(*bp-pull-state-of* (*bp-result-of* (*c-bp-paper-pull M B P*)))
⟨*proof*⟩

lemma *c-bp-paper-pull-amortized-paper-bound*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
shows *bp-pull-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-pull M B P*)) *P*
(*bp-result-of* (*c-bp-paper-pull M B P*))
(*bp-pull-amortized-budget M*)
⟨*proof*⟩

lemma *c-bp-paper-pull-refines-pull-separates*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
and *pull*:
bp-result-of (*c-bp-paper-pull M B P*) = (*S*, *beta*, *P'*)
shows *pull-separates* (*bp-view P*) *M B S beta* (*bp-view P'*)
⟨*proof*⟩

corollary *bp-realises-partition-insert-cost-bound*:
assumes *reg*: *bp-regular-invariant P*
shows ∃ *t*. *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-paper-insert x b P*)) *t* ∧
t = *bp-insert-paper-budget* (*length* (*bp-entries P*)) (*bp-block-size P*)
⟨*proof*⟩

corollary *bp-realises-partition-batch-cost-bound*:
shows ∃ *t*. *partition-batch-cost-bound*
(*bp-steps-of* (*c-bp-paper-batch-prepend xs P*)) *t xs* ∧
t = *bp-batch-prepend-paper-budget* (*length xs*) (*bp-block-size P*)
⟨*proof*⟩

corollary *bp-realises-partition-pull-cost-bound*:
assumes *pull*:
bp-result-of (*c-bp-paper-pull M B P*) = (*S*, *beta*, *P'*)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-paper-pull M B P*)) *S*
⟨*proof*⟩

The realization corollaries are the interface-facing summary. They convert the concrete step counters of *c-bp-paper-insert*, *c-bp-paper-batch-prepend*, and *c-bp-paper-pull* into the abstract predicates *partition-insert-cost-bound*, *partition-batch-cost-bound*, and *partition-pull-cost-bound*. These are the facts imported by the headline bridge; the operation-specific theorems below keep the sharper amortized statements visible for readers who want to inspect the data-structure proof itself.

theorem *bp-insert-cost-bound*:
  assumes *lazy*: *bp-lazy-ordered-invariant P*
    and *ratio*: *bp-bucket-count-ratio-ok P*
  shows *bp-amortized-step-bound*
    (*bp-steps-of* (*c-bp-paper-insert x b P*)) *P*
    (*bp-result-of* (*c-bp-paper-insert x b P*))
    (*9* + *floor-log*
      (*Suc* (*length* (*bp-entries P*) *div bp-block-size P*)))
  ⟨*proof*⟩

theorem *bp-batch-prepend-cost-bound*:
  assumes *ord*: *bp-ordered-invariant P*
    and *distinct*: *distinct* (*map fst xs*)
    and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
    and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
  shows *bp-amortized-step-bound*
    (*bp-steps-of* (*c-bp-paper-batch-prepend xs P*)) *P*
    (*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
    (*length xs* ∗
      (*2* + *floor-log* (*Suc* (*length xs div bp-block-size P*))))
  ⟨*proof*⟩

theorem *bp-pull-cost-bound*:
  assumes *inv*: *bp-invariant P*
    and *can*: *bp-can-first-bucket-pull M P*
  shows *bp-pull-amortized-step-bound*
    (*bp-steps-of* (*c-bp-paper-pull M B P*)) *P*
    (*bp-result-of* (*c-bp-paper-pull M B P*))
    (*2* ∗ *M*)
  ⟨*proof*⟩

The three theorems above are the exported paper-tight cost bounds for the bucketed structure. ⟦*bp-lazy-ordered-invariant ?P*; *bp-bucket-count-ratio-ok ?P*⟧ ⟹ *bp-amortized-step-bound* (*bp-steps-of* (*c-bp-paper-insert ?x ?b ?P*)) *?P* (*bp-result-of* (*c-bp-paper-insert ?x ?b ?P*)) (*9* + *floor-log* (*Suc* (*length* (*bp-entries ?P*) *div bp-block-size ?P*))) combines a ratio-log bucket search with the lazy split potential, giving an amortized Insert budget of a constant plus *floor-log* of *Suc* (*N div M*). This is the formal $\log(N/M)$ term required by the BMSSP parameter schedule. The batch theorem ⟦*bp-ordered-invariant ?P*; *distinct* (*map fst ?xs*); *bp-entry-keys ?xs* ∩ *keys-of* (*bp-view*

?P) = {}; *batch-prepend-admissible* (*bp-view ?P*) *?xs*⟧ ⟹ *bp-amortized-step-bound* (*bp-steps-of* (*c-bp-paper-batch-prepend ?xs ?P*)) *?P* (*bp-result-of* (*c-bp-paper-batch-prepend ?xs ?P*)) (*length ?xs* ∗ (*2* + *floor-log* (*Suc* (*length ?xs div bp-block-size ?P*)))) charges the same ratio-log scale per incoming item, matching the paper's batched prepend cost. Finally, ⟦*bp-invariant ?P*; *bp-can-first-bucket-pull ?M ?P*⟧ ⟹ *bp-pull-amortized-step-bound* (*bp-steps-of* (*c-bp-paper-pull ?M ?B ?P*)) *?P* (*bp-result-of* (*c-bp-paper-pull ?M ?B ?P*)) (*2* ∗ *?M*) gives the amortized linear Pull budget *2* ∗ *M*. These statements are intentionally stronger and more concrete than the abstract cost-interface corollaries: they expose the primitive-step and potential accounting that justifies plugging the structure into the top-level runtime proof.

end
theory *BMSSP-Bucketed-Partition*
  imports *BMSSP-Bucketed-Partition-Internal*
begin

15 Bucketed Partition

This theory is the public import point for the bucketed partition implementation. The definitions and proof-engineering lemmas live in *BMSSP-Bucketed-Partition-Internal*; importing this wrapper exports the same constants and facts to the rest of the BMSSP development while keeping the documented surface short.

end
theory *BMSSP-Code-Export*
  imports *BMSSP-Bucketed-Partition*
begin

16 Executable Bucketed BMSSP Smoke Test

This theory gives the development an executable surface. All preceding imported theory proves the paper-faithful bucketed operation costs and the concrete partition API. Those facts are essential, but they do not by themselves show a human reader that the formalized program can be run on an ordinary graph. The purpose of this file is to close that gap with a small concrete instantiation and a proof by evaluation.

The executable graph representation is deliberately plain: vertices are natural numbers, edges are triples of natural numbers, and distances are association lists sorted by vertex at the end of the run. The work-list is the paper-style bucketed partition structure imported from the bucketed theory, not the baseline sorted-list partition model. The loop below uses Pull to obtain the next bucket of unsettled vertices, relaxes outgoing edges from that batch, and feeds discovered or improved labels through BatchPrepend

for an empty partition and Insert otherwise. This shape is chosen to exercise all three bucketed operations in the worked example.

The fuel argument is an executable guard, not part of the mathematical correctness theorem. For the small smoke graph it is chosen large enough to allow all relaxations needed by the hand calculation. The checked statement at the bottom is the important artifact: *example-bmssp-correct* is proved by Isabelle's evaluator, so the bundled Poly/ML runtime executes the exported equations during the build and compares the result with the literal expected distance map.

```
type-synonym nat-edge = nat × nat × nat
type-synonym nat-graph = nat-edge list
type-synonym nat-dist = (nat × nat) list

definition bmssp-block-size :: nat where
  bmssp-block-size = 1

definition bmssp-infinity :: nat where
  bmssp-infinity = 1000000

definition bmssp-bound :: real where
  bmssp-bound = real bmssp-infinity

fun bmssp-edge-vertices :: nat-edge ⇒ nat list where
  bmssp-edge-vertices (u, v, w) = [u, v]

definition bmssp-vertices :: nat-graph ⇒ nat ⇒ nat list where
  bmssp-vertices G s = sort (remdups (s # concat (map bmssp-edge-vertices G)))

fun bmssp-lookup-dist :: nat-dist ⇒ nat ⇒ nat option where
  bmssp-lookup-dist [] x = None
| bmssp-lookup-dist ((y, d) # ds) x =
    (if x = y then Some d else bmssp-lookup-dist ds x)

fun bmssp-set-dist :: nat ⇒ nat ⇒ nat-dist ⇒ nat-dist where
  bmssp-set-dist x d [] = [(x, d)]
| bmssp-set-dist x d ((y, e) # ds) =
    (if x = y then (x, d) # ds else (y, e) # bmssp-set-dist x d ds)

definition bmssp-normalize-dist :: nat-dist ⇒ nat-dist where
  bmssp-normalize-dist ds = sort-key fst ds

definition bmssp-partition-key :: nat ⇒ nat ⇒ real where
  bmssp-partition-key x d = real d + real x / real (Suc x)

lemma bmssp-partition-key-fraction-nonneg:
  0 ≤ real x / real (Suc x)
  ⟨proof⟩
```

lemma *bmssp-partition-key-fraction-lt-one*:
real x / real (Suc x) < 1
⟨*proof*⟩

lemma *bmssp-partition-key-ge-distance*:
real d ≤ bmssp-partition-key x d
⟨*proof*⟩

lemma *bmssp-partition-key-lt-suc-distance*:
bmssp-partition-key x d < real (Suc d)
⟨*proof*⟩

lemma *bmssp-partition-key-floor*:
nat (floor (bmssp-partition-key x d)) = d
⟨*proof*⟩

lemma *bmssp-partition-key-strict-mono-distance*:
assumes *d < e*
shows *bmssp-partition-key x d < bmssp-partition-key y e*
⟨*proof*⟩

fun *bmssp-insert-updates* ::
(nat × real) list ⇒ nat bucketed-partition ⇒
nat bucketed-partition **where**
bmssp-insert-updates [] P = P
| *bmssp-insert-updates ((x, b) # xs) P =*
bmssp-insert-updates xs
(bp-result-of (c-bp-regularized-local-insert x b P))

lemma *bmssp-insert-updates-regular-invariant*:
assumes *bp-regular-invariant P*
shows *bp-regular-invariant (bmssp-insert-updates xs P)*
⟨*proof*⟩

lemma *bmssp-insert-updates-ordered-invariant*:
assumes *bp-regular-invariant P*
shows *bp-ordered-invariant (bmssp-insert-updates xs P)*
⟨*proof*⟩

lemma *bmssp-insert-updates-refines-batch-min-update*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-view (bmssp-insert-updates xs P) =*
batch-min-update (bp-view P) xs
⟨*proof*⟩

lemma *bmssp-insert-updates-ordered-invariant-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-ordered-invariant (bmssp-insert-updates xs P)*
⟨*proof*⟩

lemma *bmssp-insert-updates-refines-batch-min-update-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-view* (*bmssp-insert-updates xs P*) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

definition *bmssp-trim-empty-prefix* ::
nat bucketed-partition ⇒ *nat bucketed-partition* **where**
bmssp-trim-empty-prefix P =
P(|*bp-buckets* := *bp-drop-empty-prefix* (*bp-buckets P*)|)

lemma *bp-bucket-entries-flat-drop-empty-prefix* [*simp*]:
bp-bucket-entries-flat (*bp-drop-empty-prefix bs*) =
bp-bucket-entries-flat bs
⟨*proof*⟩

lemma *bp-drop-empty-prefix-set-subset*:
set (*bp-drop-empty-prefix bs*) ⊆ *set bs*
⟨*proof*⟩

lemma *bp-drop-empty-prefix-length-le* [*simp*]:
length (*bp-drop-empty-prefix bs*) ≤ *length bs*
⟨*proof*⟩

lemma *bp-drop-empty-prefix-sorted-wrt*:
assumes *sorted-wrt R bs*
shows *sorted-wrt R* (*bp-drop-empty-prefix bs*)
⟨*proof*⟩

lemma *bp-bucket-boundaries-ok-drop-empty-prefix*:
assumes *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*bp-drop-empty-prefix bs*)
⟨*proof*⟩

lemma *bmssp-trim-empty-prefix-entries* [*simp*]:
bp-entries (*bmssp-trim-empty-prefix P*) = *bp-entries P*
⟨*proof*⟩

lemma *bmssp-trim-empty-prefix-view* [*simp*]:
bp-view (*bmssp-trim-empty-prefix P*) = *bp-view P*
⟨*proof*⟩

lemma *bmssp-trim-empty-prefix-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bmssp-trim-empty-prefix P*)
⟨*proof*⟩

lemma *bmssp-trim-empty-prefix-regular-invariant*:

assumes *bp-regular-invariant P*
shows *bp-regular-invariant* (*bmssp-trim-empty-prefix P*)
⟨*proof*⟩

definition *bmssp-apply-updates* ::
(*nat* × *real*) *list* ⇒ *nat bucketed-partition* ⇒
nat bucketed-partition **where**
bmssp-apply-updates xs P =
(*let P0* = *bmssp-trim-empty-prefix P in*
if bp-entries P0 = []
then bp-result-of (*c-bp-paper-batch-prepend xs P0*)
else bmssp-insert-updates xs P0)

lemma *bmssp-apply-updates-refines-batch-min-update*:
assumes *reg*: *bp-regular-invariant P*
and *distinct*: *distinct* (*map fst xs*)
shows *bp-view* (*bmssp-apply-updates xs P*) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *bmssp-apply-updates-ordered-invariant*:
assumes *reg*: *bp-regular-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *admissible*:
batch-prepend-admissible (*bp-view* (*bmssp-trim-empty-prefix P*)) *xs*
shows *bp-ordered-invariant* (*bmssp-apply-updates xs P*)
⟨*proof*⟩

lemma *bmssp-apply-updates-refines-batch-min-update-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
shows *bp-view* (*bmssp-apply-updates xs P*) =
batch-min-update (*bp-view P*) *xs*
⟨*proof*⟩

lemma *bmssp-apply-updates-ordered-invariant-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
shows *bp-ordered-invariant* (*bmssp-apply-updates xs P*)
⟨*proof*⟩

lemma *bmssp-apply-updates-preserves-upper-bound*:
assumes *reg*: *bp-regular-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *distinct*: *distinct* (*map fst xs*)
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound* (*bp-view* (*bmssp-apply-updates xs P*)) *B*
⟨*proof*⟩

lemma *bmssp-apply-updates-preserves-upper-bound-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *distinct*: *distinct* (*map fst xs*)
and *values-lt*: $\bigwedge x\ b.\ (x,\ b) \in set\ xs \Longrightarrow b < B$
shows *partition-upper-bound* (*bp-view* (*bmssp-apply-updates xs P*)) *B*
⟨*proof*⟩

lemma *bmssp-pull-refines-pull-separates*:
assumes *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull M B P* = (*S*, *beta*, *P'*)
shows *pull-separates* (*bp-view P*) *M B S beta* (*bp-view P'*)
⟨*proof*⟩

lemma *bmssp-pull-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *pull*: *bp-pull M B P* = (*S*, *beta*, *P'*)
shows *bp-ordered-invariant P'*
⟨*proof*⟩

lemma *bmssp-pull-preserves-upper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull M B P* = (*S*, *beta*, *P'*)
shows *partition-upper-bound* (*bp-view P'*) *B*
⟨*proof*⟩

lemma *bmssp-pull-step-bridge*:
assumes *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull M B P* = (*S*, *beta*, *P'*)
shows *pull-separates* (*bp-view P*) *M B S beta* (*bp-view P'*)
and *bp-ordered-invariant P'*
and *partition-upper-bound* (*bp-view P'*) *B*
⟨*proof*⟩

lemma *bmssp-vertices-distinct* [*simp*]:
distinct (*bmssp-vertices G s*)
⟨*proof*⟩

lemma *finite-keys-of-bp-view* [*simp*]:
finite (*keys-of* (*bp-view P*))
⟨*proof*⟩

lemma *bmssp-pulled-length-le-block-size*:
assumes *distinct-vertices*: *distinct vertices*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D'*
shows *length* (*filter* ($\lambda x.\ x \in S \wedge x \notin set\ settled$) *vertices*)

```
    ≤ bmssp-block-size
⟨proof⟩

lemma bmssp-pulled-length-le-one:
  assumes distinct-vertices: distinct vertices
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
  shows length (filter (λx. x ∈ S ∧ x ∉ set settled) vertices)
    ≤ 1
  ⟨proof⟩

fun bmssp-relax-edges ::
  nat-graph ⇒ nat list ⇒ nat ⇒ nat ⇒ nat-dist ⇒
    (nat × real) list × nat-dist where
  bmssp-relax-edges [] settled u du ds = ([], ds)
| bmssp-relax-edges ((a, b, w) # es) settled u du ds =
    (case bmssp-relax-edges es settled u du ds of
      (updates, ds1) ⇒
        (if a = u ∧ b ∉ set settled
         then
           (let nd = du + w in
            case bmssp-lookup-dist ds1 b of
              None ⇒
                ((b, bmssp-partition-key b nd) # updates,
                 bmssp-set-dist b nd ds1)
            | Some old ⇒
                (if nd < old
                 then ((b, bmssp-partition-key b nd) # updates,
                       bmssp-set-dist b nd ds1)
                 else (updates, ds1)))
         else (updates, ds1)))

fun bmssp-relax-vertices ::
  nat-graph ⇒ nat list ⇒ nat list ⇒ nat-dist ⇒
    (nat × real) list × nat-dist where
  bmssp-relax-vertices G settled [] ds = ([], ds)
| bmssp-relax-vertices G settled (u # us) ds =
    (case bmssp-lookup-dist ds u of
       None ⇒ bmssp-relax-vertices G settled us ds
     | Some du ⇒
         (case bmssp-relax-edges G settled u du ds of
            (updates-u, ds1) ⇒
              (case bmssp-relax-vertices G settled us ds1 of
                 (updates-us, ds2) ⇒ (updates-u @ updates-us, ds2))))

fun bmssp-loop ::
  nat ⇒ nat-graph ⇒ nat list ⇒ nat list ⇒
    nat-dist ⇒ nat bucketed-partition ⇒ nat-dist where
  bmssp-loop 0 G vertices settled ds P = bmssp-normalize-dist ds
| bmssp-loop (Suc fuel) G vertices settled ds P =
```

```
    (case bp-pull bmssp-block-size bmssp-bound P of
      (S, beta, P1) ⇒
        (let pulled = filter (λx. x ∈ S ∧ x ∉ set settled) vertices in
         if pulled = []
         then bmssp-normalize-dist ds
         else
           (let settled' = pulled @ settled;
                relaxed = bmssp-relax-vertices G settled' pulled ds;
                updates = fst relaxed;
                ds' = snd relaxed;
                P2 = bmssp-apply-updates updates P1
            in bmssp-loop fuel G vertices settled' ds' P2)))

definition bmssp-distances :: nat-graph ⇒ nat ⇒ nat-dist where
  bmssp-distances G s =
    (let vertices = bmssp-vertices G s;
         P0 = bp-empty bmssp-block-size bmssp-bound;
         P1 = bp-result-of
           (c-bp-regularized-local-insert s (bmssp-partition-key s 0) P0);
         fuel = Suc (length vertices * Suc (length G))
     in bmssp-loop fuel G vertices [] [(s, 0)] P1)
```

The entry point *bmssp-distances* initializes the bucketed partition with the source at distance zero and then iterates *bmssp-loop*. The helper *bmssp-apply-updates* makes the example exercise both bulk and singleton update paths: an empty work-list receives a batch prepend, while a non-empty work-list receives local inserts. The bucketed Pull operation is invoked in every loop iteration through *bp-pull*.

```
definition example-graph :: nat-graph where
  example-graph =
    [(0, 1, 3), (0, 4, 6), (0, 2, 10),
     (1, 2, 2), (1, 4, 4), (1, 3, 9),
     (2, 3, 3), (2, 4, 1), (4, 3, 5)]
```

From source 0: d(0) = 0 d(1) = 3 via 0 -> 1 d(2) = 5 via 0 -> 1 -> 2 d(3) = 8 via 0 -> 1 -> 2 -> 3 d(4) = 6 via 0 -> 4, tied by 0 -> 1 -> 2 -> 4

```
definition example-expected-dist :: nat-dist where
  example-expected-dist = [(0, 0), (1, 3), (2, 5), (3, 8), (4, 6)]
```

The graph is small enough to audit by hand but nontrivial enough to check the executable plumbing. Vertex 0 starts the run. Vertex 1 is reached directly with cost 3; vertex 2 improves from the direct edge of cost 10 to cost 5 via 1; vertex 3 is then reached at cost 8 via 2; and vertex 4 has final cost 6, tied between the direct edge and the path through 1 and 2. The constant *example-expected-dist* records that hand calculation as a literal sorted association list.

```
lemma example-bmssp-correct:
  bmssp-distances example-graph 0 = example-expected-dist
```

⟨*proof*⟩

lemma *bmssp-counterexample-fixed*:
bmssp-distances [(*0*::*nat*, *1*::*nat*, *1*::*nat*), (*0*, *2*, *1*), (*1*, *3*, *2*)] *0*
= [(*0*, *0*), (*1*, *1*), (*2*, *1*), (*3*, *3*)]
⟨*proof*⟩

lemma *bmssp-line-graph-fixed*:
bmssp-distances [(*0*::*nat*, *1*::*nat*, *1*::*nat*), (*1*, *2*, *1*), (*2*, *3*, *1*)] *0*
= [(*0*, *0*), (*1*, *1*), (*2*, *2*), (*3*, *3*)]
⟨*proof*⟩

lemma *bmssp-empty-prefix-progress-fixed*:
bmssp-distances
[(*0*::*nat*, *1*::*nat*, *1*::*nat*), (*0*, *2*, *2*), (*0*, *3*, *3*), (*2*, *4*, *1*)] *0*
= [(*0*, *0*), (*1*, *1*), (*2*, *2*), (*3*, *3*), (*4*, *3*)]
⟨*proof*⟩

The theorem *bmssp-distances example-graph 0 = example-expected-dist* is the end-to-end smoke test. The regression lemmas above cover equal-distance frontier updates, a multi-hop line graph, and progress after a pull leaves an empty bucket in front of pending vertices. The proof method *eval* runs the generated equations inside Isabelle and proves that the result equals the expected literal distance map. The following **value** command is intentionally kept in the theory so the build log prints the computed list for human inspection, and the final **export-code** declaration emits the same executable entry point as SML in the generated directory.

value *bmssp-distances example-graph 0*

export-code *bmssp-distances example-graph example-expected-dist*
in *SML* **module-name** *BMSSP* **file-prefix** *generated/BMSSP*

end
theory *BMSSP-Executable-Base-Case*
imports *BMSSP-Base-Case BMSSP-Code-Export HOL−Library.Multiset*
begin

17 Executable Base-Case Ordering

The semantic base-case theory orders a finite set by *finite-weighted-digraph.dist*. The functions below provide an executable finite counterpart: enumerate simple walks over explicit vertex and edge lists, compute their weights by a list minimum, and sort the target vertices by the resulting finite distance scan.

fun *exec-walk* :: *'v list* ⇒ (*'v* × *'v*) *list* ⇒ *'v list* ⇒ *bool* **where**
exec-walk vs es [] ⟷ *False*
| *exec-walk vs es* [*x*] ⟷ *x* ∈ *set vs*

```
| exec-walk vs es (x # y # xs) ⟷
    x ∈ set vs ∧ (x, y) ∈ set es ∧ exec-walk vs es (y # xs)

fun exec-walk-weight :: ('v ⇒ 'v ⇒ real) ⇒ 'v list ⇒ real where
  exec-walk-weight W [] = 0
| exec-walk-weight W [x] = 0
| exec-walk-weight W (x # y # xs) = W x y + exec-walk-weight W (y # xs)

definition exec-simple-walks-betw ::
  'v list ⇒ ('v × 'v) list ⇒ 'v ⇒ 'v ⇒ 'v list list where
  exec-simple-walks-betw vs es a b =
    filter (λp. p ≠ [] ∧ hd p = a ∧ last p = b ∧
      exec-walk vs es p ∧ distinct p)
      (concat (map (λn. List.n-lists n vs) [1..<Suc (length vs)]))

definition exec-walk-weights ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ real list where
  exec-walk-weights vs es W a b =
    map (exec-walk-weight W) (exec-simple-walks-betw vs es a b)

definition exec-dist ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ real where
  exec-dist vs es W a b =
    (case exec-walk-weights vs es W a b of [] ⇒ 0 | ws ⇒ min-list ws)

definition exec-closest-vertices ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ 'v list where
  exec-closest-vertices vs es W src xs =
    sort-key (exec-dist vs es W src) (remdups xs)

definition exec-reachable ::
  'v list ⇒ ('v × 'v) list ⇒ 'v ⇒ 'v ⇒ bool where
  exec-reachable vs es a b ⟷ exec-simple-walks-betw vs es a b ≠ []

definition exec-shortest-through ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ 'v ⇒ bool where
  exec-shortest-through vs es W src x v ⟷
    (∃ p∈set (exec-simple-walks-betw vs es src v).
      exec-walk-weight W p = exec-dist vs es W src v ∧ x ∈ set p)

definition exec-bound-tree-vertices ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ bound ⇒ 'v list where
  exec-bound-tree-vertices vs es W src x B =
    filter (λv. exec-reachable vs es src v ∧
      exec-shortest-through vs es W src x v ∧
      below-bound (exec-dist vs es W src v) B)
      (remdups vs)

definition exec-base-case-order ::
```

```
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    'v ⇒ bound ⇒ 'v list where
  exec-base-case-order W src vs es x B =
     exec-closest-vertices vs es W src (exec-bound-tree-vertices vs es W src x B)

definition exec-base-case-vertices ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    nat ⇒ 'v ⇒ bound ⇒ 'v set where
  exec-base-case-vertices W src vs es k x B =
     (let xs = exec-base-case-order W src vs es x B in
       if length xs ≤ k then set xs
       else set (filter (λv. exec-dist vs es W src v < exec-dist vs es W src (xs ! k))
              (take (Suc k) xs)))

definition exec-base-case-bound ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    nat ⇒ 'v ⇒ bound ⇒ bound where
  exec-base-case-bound W src vs es k x B =
     (let xs = exec-base-case-order W src vs es x B in
       if length xs ≤ k then B else Fin (exec-dist vs es W src (xs ! k)))

definition exec-base-case-result ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    nat ⇒ 'v ⇒ bound ⇒ bound × 'v set where
  exec-base-case-result W src vs es k x B =
     (exec-base-case-bound W src vs es k x B,
      exec-base-case-vertices W src vs es k x B)

context finite-weighted-digraph
begin

lemma exec-walk-iff-walk:
  assumes set vs = V set es = E
  shows exec-walk vs es p ⟷ walk p
  ⟨proof⟩

lemma exec-walk-weight-eq-walk-weight [simp]:
  exec-walk-weight w p = walk-weight p
  ⟨proof⟩

lemma set-exec-simple-walks-betw:
  assumes vs: set vs = V
    and es: set es = E
  shows set (exec-simple-walks-betw vs es a b) = {p. simple-walk-betw a p b}
⟨proof⟩

lemma set-exec-walk-weights:
  assumes vs: set vs = V
    and es: set es = E
```

shows *set* (*exec-walk-weights vs es w a b*) = *simple-walk-weights a b*
⟨*proof*⟩

lemma *exec-walk-weights-nonempty*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *reach*: *reachable a b*
shows *exec-walk-weights vs es w a b* ≠ []
⟨*proof*⟩

lemma *exec-dist-eq-dist*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *reach*: *reachable a b*
shows *exec-dist vs es w a b* = *dist a b*
⟨*proof*⟩

lemma *exec-closest-vertices-properties*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *xs*: *set xs* = *A*
and *reach-A*: *A* ⊆ {*v*. *reachable s v*}
shows *set* (*exec-closest-vertices vs es w s xs*) = *A*
and *distinct* (*exec-closest-vertices vs es w s xs*)
and *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*)
(*exec-closest-vertices vs es w s xs*)
⟨*proof*⟩

lemma *closest-vertices-executable*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *xs*: *set xs* = *A*
and *reach-A*: *A* ⊆ {*v*. *reachable s v*}
shows *set* (*exec-closest-vertices vs es w s xs*) = *A*
and *distinct* (*exec-closest-vertices vs es w s xs*)
and *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*)
(*exec-closest-vertices vs es w s xs*)
⟨*proof*⟩

lemma *closest-vertices-eq-exec-if-inj*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *xs*: *set xs* = *A*
and *reach-A*: *A* ⊆ {*v*. *reachable s v*}
and *inj*: *inj-on* (*dist s*) *A*
shows *closest-vertices A* = *exec-closest-vertices vs es w s xs*
⟨*proof*⟩

lemma *exec-reachable-iff-reachable*:

assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-reachable vs es a b* ⟷ *reachable a b*
⟨*proof*⟩

lemma *exec-shortest-through-iff-through-single*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-shortest-through vs es w s x v* ⟷ *through* {*x*} *v*
⟨*proof*⟩

lemma *set-exec-bound-tree-vertices*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *set* (*exec-bound-tree-vertices vs es w s x B*) = *bound-tree* {*x*} *B*
⟨*proof*⟩

definition *base-case-order-impl* ::
'v list ⇒ (*'v* × *'v*) *list* ⇒ *'v* ⇒ *bound* ⇒ *'v list* **where**
base-case-order-impl vs es x B =
exec-closest-vertices vs es w s (*exec-bound-tree-vertices vs es w s x B*)

definition *base-case-vertices-impl* ::
'v list ⇒ (*'v* × *'v*) *list* ⇒ *nat* ⇒ *'v* ⇒ *bound* ⇒ *'v set* **where**
base-case-vertices-impl vs es k x B =
(*let xs* = *base-case-order-impl vs es x B in*
if length xs ≤ *k then set xs*
else set (*filter* (λ*v*. *exec-dist vs es w s v* < *exec-dist vs es w s* (*xs* ! *k*))
(*take* (*Suc k*) *xs*)))

definition *base-case-bound-impl* ::
'v list ⇒ (*'v* × *'v*) *list* ⇒ *nat* ⇒ *'v* ⇒ *bound* ⇒ *bound* **where**
base-case-bound-impl vs es k x B =
(*let xs* = *base-case-order-impl vs es x B in*
if length xs ≤ *k then B else Fin* (*exec-dist vs es w s* (*xs* ! *k*)))

definition *base-case-result-impl* ::
'v list ⇒ (*'v* × *'v*) *list* ⇒ *nat* ⇒ *'v* ⇒ *bound* ⇒ *bound* × *'v set* **where**
base-case-result-impl vs es k x B =
(*base-case-bound-impl vs es k x B*, *base-case-vertices-impl vs es k x B*)

declare *base-case-order-impl-def* [*code*]
declare *base-case-vertices-impl-def* [*code*]
declare *base-case-bound-impl-def* [*code*]
declare *base-case-result-impl-def* [*code*]

lemma *base-case-order-impl-set*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*

shows *set* (*base-case-order-impl vs es x B*) = *bound-tree* {*x*} *B*
⟨*proof*⟩

lemma *base-case-order-impl-eq-if-inj*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *inj*: *inj-on* (*dist s*) (*bound-tree* {*x*} *B*)
shows *base-case-order x B* = *base-case-order-impl vs es x B*
⟨*proof*⟩

lemma *base-case-order-impl-distinct*:
distinct (*base-case-order-impl vs es x B*)
⟨*proof*⟩

lemma *base-case-order-impl-sorted*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) (*base-case-order-impl vs es x B*)
⟨*proof*⟩

lemma *base-case-order-impl-length-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *length* (*base-case-order-impl vs es x B*) = *length* (*base-case-order x B*)
⟨*proof*⟩

lemma *dist-map-eq-if-same-sorted-set*:
assumes *sorted-x*: *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) *xs*
and *sorted-y*: *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) *ys*
and *distinct-x*: *distinct xs*
and *distinct-y*: *distinct ys*
and *set-eq*: *set xs* = *set ys*
shows *map* (*dist s*) *xs* = *map* (*dist s*) *ys*
⟨*proof*⟩

lemma *base-case-order-impl-kth-dist-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *k*: *k* < *length* (*base-case-order x B*)
shows *dist s* ((*base-case-order x B*) ! *k*) =
dist s ((*base-case-order-impl vs es x B*) ! *k*)
⟨*proof*⟩

lemma *exec-dist-eq-dist-on-base-case-order-impl*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *v*: *v* ∈ *set* (*base-case-order-impl vs es x B*)
shows *exec-dist vs es w s v* = *dist s v*
⟨*proof*⟩

lemma *base-case-bound-impl-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *base-case-bound-impl vs es k x B* = *base-case-bound k x B*
⟨*proof*⟩

lemma *base-case-vertices-impl-success*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *len*: *length* (*base-case-order-impl vs es x B*) ≤ *k*
shows *base-case-vertices-impl vs es k x B* = *bound-tree* {*x*} *B*
⟨*proof*⟩

lemma *base-case-vertices-impl-partial*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *len*: *k* < *length* (*base-case-order-impl vs es x B*)
shows *base-case-vertices-impl vs es k x B* =
bound-tree {*x*} (*Fin* (*exec-dist vs es w s* ((*base-case-order-impl vs es x B*) ! *k*)))
⟨*proof*⟩

lemma *base-case-vertices-impl-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *base-case-vertices-impl vs es k x B* = *base-case-vertices k x B*
⟨*proof*⟩

lemma *base-case-result-impl-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *base-case-result-impl vs es k x B* = *base-case-result k x B*
⟨*proof*⟩

lemma *base-case-result-impl-bmssp-post*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *S*: *S* = {*x*}
shows *case base-case-result-impl vs es k x B of* (*B*′, *U*) ⇒
bmssp-post d S B (λ*v*. *if v* ∈ *U then dist s v else d v*) *B*′ *U*
⟨*proof*⟩

lemma *base-case-order-impl-eq-exec-base-case-order*:
base-case-order-impl vs es x B = *exec-base-case-order w s vs es x B*
⟨*proof*⟩

lemma *base-case-vertices-impl-eq-exec-base-case-vertices*:
base-case-vertices-impl vs es k x B = *exec-base-case-vertices w s vs es k x B*
⟨*proof*⟩

lemma *base-case-bound-impl-eq-exec-base-case-bound*:
base-case-bound-impl vs es k x B = *exec-base-case-bound w s vs es k x B*
⟨*proof*⟩

lemma *base-case-result-impl-eq-exec-base-case-result*:
base-case-result-impl vs es k x B = *exec-base-case-result w s vs es k x B*
⟨*proof*⟩

lemma *exec-base-case-bound-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-base-case-bound w s vs es k x B* = *base-case-bound k x B*
⟨*proof*⟩

lemma *exec-base-case-vertices-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-base-case-vertices w s vs es k x B* = *base-case-vertices k x B*
⟨*proof*⟩

lemma *exec-base-case-result-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-base-case-result w s vs es k x B* = *base-case-result k x B*
⟨*proof*⟩

lemmas *base-case-bound-exec-code* [*code*] = *exec-base-case-bound-eq*[*symmetric*]
lemmas *base-case-vertices-exec-code* [*code*] = *exec-base-case-vertices-eq*[*symmetric*]
lemmas *base-case-result-exec-code* [*code*] = *exec-base-case-result-eq*[*symmetric*]

end

declare *finite-weighted-digraph.base-case-bound-exec-code* [*code*]
declare *finite-weighted-digraph.base-case-vertices-exec-code* [*code*]
declare *finite-weighted-digraph.base-case-result-exec-code* [*code*]

definition *example-edges* :: (*nat* × *nat*) *list* **where**
example-edges = *map* (λ(*u*, *v*, *c*). (*u*, *v*)) *example-graph*

definition *example-vertices* :: *nat list* **where**
example-vertices = *bmssp-vertices example-graph 0*

definition *example-V* :: *nat set* **where**
example-V = *set example-vertices*

definition *example-E* :: (*nat* × *nat*) *set* **where**
example-E = *set example-edges*

definition *example-weight* :: *nat* ⇒ *nat* ⇒ *real* **where**
example-weight u v =
(*case map-of* (*map* (λ(*a*, *b*, *c*). ((*a*, *b*), *real c*)) *example-graph*) (*u*, *v*) *of*
None ⇒ *0*
| *Some c* ⇒ *c*)

lemma *example-vertices-set* [*simp*]: *set example-vertices* = *example-V*
⟨*proof*⟩

lemma *example-edges-set* [*simp*]: *set example-edges* = *example-E*
⟨*proof*⟩

lemma *example-finite-weighted-digraph*:
finite-weighted-digraph example-V example-E example-weight 0
⟨*proof*⟩

interpretation *example-fw*: *finite-weighted-digraph example-V example-E example-weight 0*
⟨*proof*⟩

lemma *example-exec-simple-walk-0*:
[*0*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 0*)
⟨*proof*⟩

lemma *example-exec-simple-walk-1*:
[*0*, *1*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 1*)
⟨*proof*⟩

lemma *example-exec-simple-walk-2*:
[*0*, *1*, *2*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 2*)
⟨*proof*⟩

lemma *example-exec-simple-walk-3*:
[*0*, *1*, *2*, *3*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 3*)
⟨*proof*⟩

lemma *example-exec-simple-walk-4*:
[*0*, *4*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 4*)
⟨*proof*⟩

lemma *example-exec-reachable-if-in-V*:
assumes *v* ∈ *example-V*
shows *exec-reachable example-vertices example-edges 0 v*
⟨*proof*⟩

lemma *example-dist-inj-on-V*:
inj-on (*finite-weighted-digraph.dist example-V example-E example-weight 0*) *example-V*
⟨*proof*⟩

lemma *example-base-case-order-code* [*code-unfold*]:
finite-weighted-digraph.base-case-order example-V example-E example-weight 0 x B =
exec-base-case-order example-weight 0 example-vertices example-edges x B
⟨*proof*⟩

lemma *example-base-case-vertices-code* [*code-unfold*]:
finite-weighted-digraph.base-case-vertices example-V example-E example-weight 0 k x B =
exec-base-case-vertices example-weight 0 example-vertices example-edges k x B
⟨*proof*⟩

lemma *example-base-case-result-code* [*code-unfold*]:
finite-weighted-digraph.base-case-result example-V example-E example-weight 0 k x B =
exec-base-case-result example-weight 0 example-vertices example-edges k x B
⟨*proof*⟩

value *exec-closest-vertices*
(*bmssp-vertices example-graph 0*) *example-edges example-weight 0*
(*bmssp-vertices example-graph 0*)

value *exec-base-case-order*
example-weight 0 (*bmssp-vertices example-graph 0*) *example-edges 0* (*Fin 100*)

value *exec-base-case-result*
example-weight 0 (*bmssp-vertices example-graph 0*) *example-edges 3 0* (*Fin 100*)
::
bound × *nat set*

value *finite-weighted-digraph.base-case-vertices*
example-V example-E example-weight 0 3 0 (*Fin 100*) :: *nat set*

value *finite-weighted-digraph.base-case-order*
example-V example-E example-weight 0 0 (*Fin 100*)

value *finite-weighted-digraph.base-case-result*
example-V example-E example-weight 0 3 0 (*Fin 100*) :: *bound* × *nat set*

end
theory *BMSSP-Unique-Shortest-Tree*
imports *BMSSP-Shortest-Path-Lemmas*
begin

18 Unique Shortest-Path Tree

After tie-breaking, shortest paths form a tree rooted at the source. This locale makes that assumption explicit and derives the tree notions used by later BMSSP arguments from the graph model.

locale *unique-shortest-digraph* = *finite-weighted-digraph* +
  assumes *unique-shortest-walk*:
    $[\![$*shortest-walk s p v*; *shortest-walk s q v*$]\!] \Longrightarrow p = q$
begin

definition *shortest-path-to* **where**
  shortest-path-to v = (*THE p*. *shortest-walk s p v*)

definition *tree-path* **where**
  tree-path u v $\longleftrightarrow$ *reachable s v* $\wedge$ *u* $\in$ *set* (*shortest-path-to v*)

definition *tree-of* **where**
  tree-of u = $\{v \in V$. *tree-path u v*$\}$

definition *tree-set* **where**
  tree-set S = $\{v \in V$. $\exists\, u{\in}S$. *tree-path u v*$\}$

definition *tree-antichain* **where**
  tree-antichain S $\longleftrightarrow$ ($\forall\, u{\in}S$. $\forall\, v{\in}S$. *tree-path u v* $\longrightarrow$ *u* = *v*)

lemma *tree-antichain-subset*:
  assumes *anti*: *tree-antichain T*
    and *subset*: *S* $\subseteq$ *T*
  shows *tree-antichain S*
  ⟨*proof*⟩

lemma *tree-antichain-singleton* [*simp*]:
  tree-antichain $\{x\}$
  ⟨*proof*⟩

lemma *tree-pathD*:
  assumes *tree-path u v*
  shows *reachable s v u* $\in$ *set* (*shortest-path-to v*)
  ⟨*proof*⟩

lemma *tree-pathI*:
  assumes *reachable s v u* $\in$ *set* (*shortest-path-to v*)
  shows *tree-path u v*
  ⟨*proof*⟩

lemma *shortest-path-to-ex1*:
  assumes *reachable s v*
  shows $\exists!p$. *shortest-walk s p v*
⟨*proof*⟩

lemma *shortest-path-to-shortest*:
assumes *reachable s v*
shows *shortest-walk s* (*shortest-path-to v*) *v*
⟨*proof*⟩

lemma *shortest-path-to-successively-tight*:
assumes *reachable s v*
shows *successively tight-edge-step* (*shortest-path-to v*)
⟨*proof*⟩

lemma *shortest-path-prefix-eq*:
assumes *reach-v*: *reachable s v*
and *i*: $i < length$ (*shortest-path-to v*)
shows *shortest-path-to* (*shortest-path-to v* ! *i*) =
take (*Suc i*) (*shortest-path-to v*)
⟨*proof*⟩

lemma *tree-path-between-shortest-path-indices*:
assumes *reach-v*: *reachable s v*
and *ij*: $i \leq j$
and *j*: $j < length$ (*shortest-path-to v*)
shows *tree-path* (*shortest-path-to v* ! *i*) (*shortest-path-to v* ! *j*)
⟨*proof*⟩

lemma *tree-prefix-of-shortest-path-suffix*:
assumes *reach-v*: *reachable s v*
and *i*: $i < length$ (*shortest-path-to v*)
shows *set* (*take k* (*drop i* (*shortest-path-to v*))) ⊆
tree-of (*shortest-path-to v* ! *i*)
⟨*proof*⟩

lemma *tree-path-dist-le*:
assumes *tree-path u v*
shows *dist s u* ≤ *dist s v*
⟨*proof*⟩

lemma *tree-path-root-reachable*:
assumes *tree-path u v*
shows *reachable s u*
⟨*proof*⟩

lemma *tree-path-root-in-V*:
assumes *tree-path u v*
shows *u* ∈ *V*
⟨*proof*⟩

lemma *tree-path-trans*:
assumes *uv*: *tree-path u v*

and *xu*: *tree-path x u*
shows *tree-path x v*
⟨*proof*⟩

lemma *tree-path-comparable*:
assumes *ux*: *tree-path u x*
and *vx*: *tree-path v x*
shows *tree-path u v* ∨ *tree-path v u*
⟨*proof*⟩

lemma *tree-antichainD*:
assumes *tree-antichain S*
and $u \in S$ $v \in S$ *tree-path u v*
shows $u = v$
⟨*proof*⟩

lemma *tree-of-disjoint-if-antichain*:
assumes *anti*: *tree-antichain S*
and *uS*: $u \in S$
and *vS*: $v \in S$
and *neq*: $u \neq v$
shows *tree-of u* ∩ *tree-of v* = {}
⟨*proof*⟩

lemma *through-iff-tree-path*:
assumes *reachable s v*
shows *through S v* ⟷ (∃ *u*∈*S*. *tree-path u v*)
⟨*proof*⟩

lemma *bound-tree-eq-tree-set*:
bound-tree S B = {*v* ∈ *tree-set S*. *below-bound* (*dist s v*) *B*}
⟨*proof*⟩

end

end
theory *BMSSP-Find-Pivots-Core*
imports *BMSSP-Unique-Shortest-Tree*
begin

19 FindPivots Core Relaxation Lemmas

The concrete FindPivots procedure performs bounded Bellman-Ford-style relaxations. This theory packages the generic relaxation facts in the form needed for that proof: vertices reached by a bounded tight path from a complete source are complete after the corresponding path relaxations, and remain complete through additional valid relaxations.

context *unique-shortest-digraph*

begin

lemma *finite-E* [*simp*]: *finite E*
⟨*proof*⟩

definition *outgoing-edges* **where**
outgoing-edges F = {(*u*, *v*) ∈ *E*. *u* ∈ *F*}

lemma *finite-outgoing-edges* [*simp*]:
finite (*outgoing-edges F*)
⟨*proof*⟩

definition *edge-list-of* **where**
edge-list-of A = (*SOME xs*. *set xs* = *A* ∧ *distinct xs*)

lemma *edge-list-of-properties*:
assumes *finite A*
shows *set* (*edge-list-of A*) = *A*
and *distinct* (*edge-list-of A*)
⟨*proof*⟩

definition *relax-frontier* **where**
relax-frontier d F = *relax-edges d* (*edge-list-of* (*outgoing-edges F*))

lemma *relax-frontier-le*:
relax-frontier d F x ≤ *d x*
⟨*proof*⟩

lemma *relax-frontier-sound*:
assumes *sound*: *sound-label d*
and *frontier-reaches*: ⋀*u*. *u* ∈ *F* ⟹ *reachable s u*
shows *sound-label* (*relax-frontier d F*)
⟨*proof*⟩

fun *relax-rounds* **where**
relax-rounds d [] = *d*
| *relax-rounds d* (*r* # *rs*) = *relax-rounds* (*relax-edges d r*) *rs*

definition *within-k-edges* **where**
within-k-edges p k ⟷ *p* ≠ [] ∧ *length* (*path-edges p*) ≤ *k*

lemma *relax-rounds-concat*:
relax-rounds d rounds = *relax-edges d* (*concat rounds*)
⟨*proof*⟩

lemma *relax-rounds-sound*:
assumes *sound*: *sound-label d*
and *edges*: ⋀*u v*. (*u*, *v*) ∈ *set* (*concat rounds*) ⟹ (*u*, *v*) ∈ *E*
and *reaches*: ⋀*u v*. (*u*, *v*) ∈ *set* (*concat rounds*) ⟹ *reachable s u*

shows *sound-label* (*relax-rounds d rounds*)
⟨*proof*⟩

lemma *relax-rounds-preserves-complete-sound*:
assumes *sound*: *sound-label d*
and *complete-x*: *d x* = *dist s x*
and *edges*: ⋀*u v*. (*u*, *v*) ∈ *set* (*concat rounds*) ⟹ (*u*, *v*) ∈ *E*
and *reaches*: ⋀*u v*. (*u*, *v*) ∈ *set* (*concat rounds*) ⟹ *reachable s u*
shows *relax-rounds d rounds x* = *dist s x*
⟨*proof*⟩

lemma *path-edges-length*:
length (*path-edges p*) = *length p* − *1*
⟨*proof*⟩

lemma *within-k-edgesD*:
assumes *within-k-edges p k*
shows *p* ≠ [] *length* (*path-edges p*) ≤ *k*
⟨*proof*⟩

lemma *find-pivots-tight-path-prefix-completion*:
assumes *bounded*: *within-k-edges p k*
and *sound*: *sound-label d*
and *complete-source*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *es*: *es* = *path-edges p* @ *extra*
and *extra-edges*: ⋀*u v*. (*u*, *v*) ∈ *set extra* ⟹ (*u*, *v*) ∈ *E*
and *extra-reaches*: ⋀*u v*. (*u*, *v*) ∈ *set extra* ⟹ *reachable s u*
shows *sound-label* (*relax-edges d es*)
and *relax-edges d es* (*last p*) = *dist s* (*last p*)
⟨*proof*⟩

lemma *find-pivots-tight-path-block-completion*:
assumes *bounded*: *within-k-edges p k*
and *sound*: *sound-label d*
and *complete-source*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *es*: *es* = *pre* @ *path-edges p* @ *extra*
and *pre-edges*: ⋀*u v*. (*u*, *v*) ∈ *set pre* ⟹ (*u*, *v*) ∈ *E*
and *pre-reaches*: ⋀*u v*. (*u*, *v*) ∈ *set pre* ⟹ *reachable s u*
and *extra-edges*: ⋀*u v*. (*u*, *v*) ∈ *set extra* ⟹ (*u*, *v*) ∈ *E*
and *extra-reaches*: ⋀*u v*. (*u*, *v*) ∈ *set extra* ⟹ *reachable s u*
shows *sound-label* (*relax-edges d es*)
and *relax-edges d es* (*last p*) = *dist s* (*last p*)
⟨*proof*⟩

lemma *find-pivots-rounds-block-completion*:
assumes *bounded*: *within-k-edges p k*
and *sound*: *sound-label d*

and *complete-source*: $d\ (hd\ p) = dist\ s\ (hd\ p)$
and *tight*: *successively tight-edge-step p*
and *rounds*: *concat rounds* = *pre* @ *path-edges p* @ *extra*
and *pre-edges*: $\bigwedge u\ v.\ (u,\ v) \in set\ pre \Longrightarrow (u,\ v) \in E$
and *pre-reaches*: $\bigwedge u\ v.\ (u,\ v) \in set\ pre \Longrightarrow reachable\ s\ u$
and *extra-edges*: $\bigwedge u\ v.\ (u,\ v) \in set\ extra \Longrightarrow (u,\ v) \in E$
and *extra-reaches*: $\bigwedge u\ v.\ (u,\ v) \in set\ extra \Longrightarrow reachable\ s\ u$
shows *sound-label* (*relax-rounds d rounds*)
and *relax-rounds d rounds* (*last p*) = *dist s* (*last p*)
⟨*proof*⟩

lemma *relax-frontier-tight-successor-complete*:
assumes *sound*: *sound-label d*
and *uF*: $u \in F$
and *frontier-reaches*: $\bigwedge x.\ x \in F \Longrightarrow reachable\ s\ x$
and *complete-u*: $d\ u = dist\ s\ u$
and *tight*: *tight-edge-step u v*
shows *sound-label* (*relax-frontier d F*)
and *relax-frontier d F v* = *dist s v*
⟨*proof*⟩

end

end
theory *BMSSP-Find-Pivots*
imports *BMSSP-Find-Pivots-Core BMSSP-Algorithm-Correctness*
begin

20 Concrete FindPivots Rounds

FindPivots is the first nontrivial subroutine in a BMSSP recursive step. Conceptually, it starts from a current source frontier, performs a bounded number of relaxation rounds, records the vertices reached by those rounds, and chooses a smaller set of pivots whose shortest-path trees are large enough to justify the next recursive partitioning phase.

This theory gives the concrete round-by-round model used by the later correctness and cost proofs. The round state consists of a label function, a frontier, and a seen set. One round relaxes edges out of the frontier, selects the next frontier under the current upper bound, and extends the seen set. The capped variant stops early once the seen set exceeds a cardinality cap. That cap is the formal hook for the paper's parameter schedule: if the local search grows too quickly, the original source set is already large enough to serve as the next pivot set.

The proof obligations in this file fall into three groups. First, the frontier and seen sets stay inside the finite vertex set and have the expected cardinality bounds under an outdegree assumption. Second, relaxation preserves the

BMSSP source precondition and label soundness. Third, if the capped search does not overflow, every short tight path starting from a complete source is completed, and long tight trees expose a pivot. These facts are assembled into the concrete FindPivots postcondition consumed by the abstract algorithm-correctness theory.

context *unique-shortest-digraph*
begin

definition *fp-next* **where**
fp-next d F B =
{*v* ∈ *V*. ∃ *u*∈*F*. (*u*, *v*) ∈ *E* ∧ *d u* + *w u v* ≤ *d v* ∧
below-bound (*d u* + *w u v*) *B*}

fun *fp-iter* **where**
fp-iter 0 d F W B = (*d*, *F*, *W*)
| *fp-iter* (*Suc n*) *d F W B* =
(*let d*′ = *relax-frontier d F*;
F′ = *fp-next d F B*;
W′ = *W* ∪ *F*′
in fp-iter n d′ *F*′ *W*′ *B*)

fun *fp-iter-capped* **where**
fp-iter-capped 0 cap d F W B = (*d*, *F*, *W*)
| *fp-iter-capped* (*Suc n*) *cap d F W B* =
(*let d*′ = *relax-frontier d F*;
F′ = *fp-next d F B*;
W′ = *W* ∪ *F*′
in if card W′ > *cap then* (*d*′, *F*′, *W*′)
else fp-iter-capped n cap d′ *F*′ *W*′ *B*)

fun *fp-iter-capped-scan-cost* **where**
fp-iter-capped-scan-cost 0 cap d F W B = *0*
| *fp-iter-capped-scan-cost* (*Suc n*) *cap d F W B* =
(*let d*′ = *relax-frontier d F*;
F′ = *fp-next d F B*;
W′ = *W* ∪ *F*′;
round = *card* (*outgoing-edges F*)
in round + (*if card W*′ > *cap then 0*
else fp-iter-capped-scan-cost n cap d′ *F*′ *W*′ *B*))

definition *fp-label* **where**
fp-label st = *fst st*

definition *fp-frontier* **where**
fp-frontier st = *fst* (*snd st*)

definition *fp-seen* **where**
fp-seen st = *snd* (*snd st*)

definition *find-pivots-label* **where**
find-pivots-label k d S B = *fp-label* (*fp-iter k d S S B*)

definition *find-pivots-seen* **where**
find-pivots-seen k d S B = *fp-seen* (*fp-iter k d S S B*)

definition *find-pivots-seen-capped* **where**
find-pivots-seen-capped k cap d S B = *fp-seen* (*fp-iter-capped k cap d S S B*)

definition *find-pivots-label-capped* **where**
find-pivots-label-capped k cap d S B = *fp-label* (*fp-iter-capped k cap d S S B*)

definition *find-pivots-pivots-capped* **where**
find-pivots-pivots-capped k cap d S B =
(*if card* (*find-pivots-seen-capped k cap d S B*) > *cap then S*
else {*u* ∈ *S*. *k* ≤ *card* (*tree-of u* ∩ *find-pivots-seen-capped k cap d S B*)})

definition *find-pivots-pivots* **where**
find-pivots-pivots k d S B =
{*u* ∈ *S*. *k* ≤ *card* (*tree-of u* ∩ *find-pivots-seen k d S B*)}

definition *short-tight-witness* **where**
short-tight-witness k d S B x ⟷
(∃ *p*. *p* ≠ [] ∧ *hd p* ∈ *S* ∧ *last p* = *x* ∧
d (*hd p*) = *dist s* (*hd p*) ∧ *successively tight-edge-step p* ∧
(∀ *y*∈*set p*. *below-bound* (*dist s y*) *B*) ∧
length (*path-edges p*) ≤ *k*)

definition *find-pivots-short-vertices* **where**
find-pivots-short-vertices k d S B =
{*x* ∈ *V*. *short-tight-witness k d S B x*}

definition *out-neighbors* **where**
out-neighbors F = {*v* ∈ *V*. ∃ *u*∈*F*. (*u*, *v*) ∈ *E*}

definition *outdegree-le* **where**
outdegree-le Δ ⟷ (∀ *u*∈*V*. *card* (*out-neighbors* {*u*}) ≤ Δ)

definition *edge-outdegree-le* **where**
edge-outdegree-le Δ ⟷ (∀ *u*∈*V*. *card* (*outgoing-edges* {*u*}) ≤ Δ)

The definitions above mirror the paper's informal description. The set *fp-next* is the next bounded frontier: it contains vertices reached by a currently tight relaxation whose value remains below the call bound. The iterator *fp-iter* performs exactly k such rounds, while *fp-iter-capped* returns early when the accumulated seen set exceeds the cap. The projections *find-pivots-label-capped*, *find-pivots-seen-capped*, and *find-pivots-pivots-capped* are the concrete objects used by later BMSSP steps.

Two degree predicates are kept separate. *outdegree-le* bounds the number of distinct out-neighbours and is convenient for bounding the size of the seen set. *edge-outdegree-le* bounds outgoing edge records and is used for scan-cost accounting. The distinction matters in multigraph-style encodings where several edges may point to the same neighbour.

lemma *finite-tree-of* [*simp*]:
finite (*tree-of u*)
⟨*proof*⟩

lemma *fp-next-reaches*:
assumes *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
and *v*: *v* ∈ *fp-next d F B*
shows *reachable s v*
⟨*proof*⟩

lemma *fp-next-subset-V*:
fp-next d F B ⊆ *V*
⟨*proof*⟩

lemma *fp-next-subset-out-neighbors*:
fp-next d F B ⊆ *out-neighbors F*
⟨*proof*⟩

lemma *out-neighbors-subset-V*:
out-neighbors F ⊆ *V*
⟨*proof*⟩

lemma *finite-out-neighbors* [*simp*]:
finite (*out-neighbors F*)
⟨*proof*⟩

lemma *out-neighbors-UN-singleton*:
out-neighbors F = ($\bigcup$ *u*∈*F*. *out-neighbors* {*u*})
⟨*proof*⟩

lemma *outgoing-edges-UN-singleton*:
outgoing-edges F = ($\bigcup$ *u*∈*F*. *outgoing-edges* {*u*})
⟨*proof*⟩

lemma *card-out-neighbors-le*:
assumes *F-subset*: *F* ⊆ *V*
and *degree*: *outdegree-le* Δ
shows *card* (*out-neighbors F*) ≤ Δ ∗ *card F*
⟨*proof*⟩

lemma *card-fp-next-le*:
assumes *F-subset*: *F* ⊆ *V*
and *degree*: *outdegree-le* Δ

shows *card* (*fp-next d F B*) $\leq \Delta * card\ F$
⟨*proof*⟩

lemma *card-outgoing-edges-le*:
assumes *F-subset*: $F \subseteq V$
and *degree*: *edge-outdegree-le* Δ
shows *card* (*outgoing-edges F*) $\leq \Delta * card\ F$
⟨*proof*⟩

lemma *card-seen-after-fp-step-le*:
assumes *W-subset*: $W \subseteq V$
and *F-subset*: $F \subseteq V$
and *degree*: *outdegree-le* Δ
shows *card* ($W \cup$ *fp-next d F B*) $\leq$ *card W* $+ \Delta *$ *card F*
⟨*proof*⟩

lemma *card-seen-after-fp-step-cap-le*:
assumes *F-subset-W*: $F \subseteq W$
and *W-subset*: $W \subseteq V$
and *degree*: *outdegree-le* Δ
and *W-cap*: *card W* $\leq$ *cap*
shows *card* ($W \cup$ *fp-next d F B*) $\leq$ *Suc* $\Delta *$ *cap*
⟨*proof*⟩

lemma *fp-iter-capped-seen-subset-V*:
assumes *F-subset*: $F \subseteq V$
and *W-subset*: $W \subseteq V$
shows *fp-seen* (*fp-iter-capped n cap d F W B*) $\subseteq V$
⟨*proof*⟩

lemma *fp-iter-capped-frontier-subset-seen*:
assumes $F \subseteq W$
shows *fp-frontier* (*fp-iter-capped n cap d F W B*) $\subseteq$
fp-seen (*fp-iter-capped n cap d F W B*)
⟨*proof*⟩

lemma *fp-iter-capped-seen-initial-subset*:
$W \subseteq$ *fp-seen* (*fp-iter-capped n cap d F W B*)
⟨*proof*⟩

lemma *fp-iter-capped-seen-reaches*:
assumes *F-reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
and *W-reaches*: $\bigwedge u.\ u \in W \Longrightarrow$ *reachable s u*
shows $\forall\, u \in$ *fp-seen* (*fp-iter-capped n cap d F W B*). *reachable s u*
⟨*proof*⟩

lemma *find-pivots-seen-capped-reaches*:
assumes *S-reaches*: $\bigwedge u.\ u \in S \Longrightarrow$ *reachable s u*
and *u*: $u \in$ *find-pivots-seen-capped k cap d S B*

shows *reachable s u*
⟨*proof*⟩

theorem *fp-iter-capped-seen-card-le*:
assumes *F-subset-W*: $F \subseteq W$
and *W-subset*: $W \subseteq V$
and *degree*: *outdegree-le* Δ
and *W-cap*: *card* $W \leq cap$
shows *card* (*fp-seen* (*fp-iter-capped n cap d F W B*)) $\leq$ *Suc* $\Delta * cap$
⟨*proof*⟩

theorem *find-pivots-seen-capped-card-le*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *outdegree-le* Δ
and *S-cap*: *card* $S \leq cap$
shows *card* (*find-pivots-seen-capped k cap d S B*) $\leq$ *Suc* $\Delta * cap$
⟨*proof*⟩

theorem *fp-iter-capped-scan-cost-le*:
assumes *F-subset-W*: $F \subseteq W$
and *W-subset*: $W \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *W-cap*: *card* $W \leq cap$
shows *fp-iter-capped-scan-cost n cap d F W B* $\leq n * \Delta * cap$
⟨*proof*⟩

theorem *find-pivots-capped-scan-cost-le*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card* $S \leq cap$
shows *fp-iter-capped-scan-cost k cap d S S B* $\leq k * \Delta * cap$
⟨*proof*⟩

theorem *fp-iter-capped-scan-cost-le-seen*:
assumes *F-subset-W*: $F \subseteq W$
and *W-subset*: $W \subseteq V$
and *degree*: *edge-outdegree-le* Δ
shows *fp-iter-capped-scan-cost n cap d F W B* $\leq$
$n * \Delta *$ *card* (*fp-seen* (*fp-iter-capped n cap d F W B*))
⟨*proof*⟩

theorem *find-pivots-capped-scan-cost-le-seen*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
shows *fp-iter-capped-scan-cost k cap d S S B* $\leq$
$k * \Delta *$ *card* (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

theorem *fp-iter-capped-scan-cost-le-outgoing-seen*:

assumes *F-subset-W*: $F \subseteq W$
shows *fp-iter-capped-scan-cost n cap d F W B* $\leq$
$n * card$ (*outgoing-edges* (*fp-seen* (*fp-iter-capped n cap d F W B*)))
⟨*proof*⟩

theorem *find-pivots-capped-scan-cost-le-outgoing-seen*:
shows *fp-iter-capped-scan-cost k cap d S S B* $\leq$
$k * card$ (*outgoing-edges* (*find-pivots-seen-capped k cap d S B*))
⟨*proof*⟩

The capped search scans only edges leaving frontiers that have already been absorbed into the final seen set. The theorem *fp-iter-capped-scan-cost ?k ?cap ?d ?S ?S ?B* $\leq$ *?k* $*$ *card* (*outgoing-edges* (*find-pivots-seen-capped ?k ?cap ?d ?S ?B*)) is the compact cost statement used later: *fp-iter-capped-scan-cost* is bounded by the number of rounds times the outgoing-edge volume of *find-pivots-seen-capped*. This is deliberately stated in terms of the final seen set rather than a sum over intermediate frontiers, because the top-level analysis controls the size of that set through the cap.

lemma *fp-iter-capped-eq-fp-iter-if-final-within-cap*:
assumes *final-cap*: *card* (*fp-seen* (*fp-iter-capped n cap d F W B*)) $\leq$ *cap*
shows *fp-iter-capped n cap d F W B* = *fp-iter n d F W B*
⟨*proof*⟩

lemma *fp-iter-seen-subset-V*:
assumes *F-subset*: $F \subseteq V$
and *W-subset*: $W \subseteq V$
shows *fp-seen* (*fp-iter n d F W B*) $\subseteq V$
⟨*proof*⟩

lemma *find-pivots-seen-subset-V*:
assumes $S \subseteq V$
shows *find-pivots-seen k d S B* $\subseteq V$
⟨*proof*⟩

lemma *finite-find-pivots-seen* [*simp*]:
assumes $S \subseteq V$
shows *finite* (*find-pivots-seen k d S B*)
⟨*proof*⟩

lemma *fp-next-tight-successor*:
assumes *sound*: *sound-label d*
and *uF*: $u \in F$
and *complete-u*: *d u* = *dist s u*
and *tight*: *tight-edge-step u v*
and *below-v*: *below-bound* (*dist s v*) *B*
shows $v \in$ *fp-next d F B*
⟨*proof*⟩

lemma *fp-step-tight-successor-complete*:

assumes *sound*: *sound-label d*
and *uF*: $u \in F$
and *frontier-reaches*: $\bigwedge x.\ x \in F \Longrightarrow$ *reachable s x*
and *complete-u*: *d u* = *dist s u*
and *tight*: *tight-edge-step u v*
and *below-v*: *below-bound* (*dist s v*) *B*
shows *sound-label* (*relax-frontier d F*)
and *relax-frontier d F v* = *dist s v*
and $v \in$ *fp-next d F B*
⟨*proof*⟩

lemma *fp-iter-seen-mono*:
assumes $x \in W$
shows $x \in$ *fp-seen* (*fp-iter n d F W B*)
⟨*proof*⟩

lemma *relax-frontier-preserves-complete-sound*:
assumes *sound*: *sound-label d*
and *complete-x*: *d x* = *dist s x*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *relax-frontier d F x* = *dist s x*
⟨*proof*⟩

lemma *relax-frontier-preserves-bmssp-pre-full*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *bmssp-pre-full* (*relax-frontier d F*) *S B*
⟨*proof*⟩

lemma *fp-iter-capped-preserves-bmssp-pre-full*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B0*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *bmssp-pre-full* (*fp-label* (*fp-iter-capped n cap d F W B*)) *S B0*
⟨*proof*⟩

lemma *find-pivots-label-capped-preserves-source-pre*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge u.\ u \in S \Longrightarrow$ *reachable s u*
shows *bmssp-pre-full* (*find-pivots-label-capped k cap d S B*) *S B*
⟨*proof*⟩

lemma *fp-iter-capped-label-le*:
fp-label (*fp-iter-capped n cap d F W B*) $x \leq$ *d x*
⟨*proof*⟩

lemma *find-pivots-label-capped-le*:

find-pivots-label-capped k cap d S B x $\leq$ *d x*
⟨*proof*⟩

lemma *fp-iter-capped-label-sound*:
assumes *sound*: *sound-label d*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *sound-label* (*fp-label* (*fp-iter-capped n cap d F W B*))
⟨*proof*⟩

lemma *fp-iter-capped-preserves-complete-sound*:
assumes *sound*: *sound-label d*
and *complete-x*: *d x* = *dist s x*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *fp-label* (*fp-iter-capped n cap d F W B*) *x* = *dist s x*
⟨*proof*⟩

theorem *find-pivots-capped-overflow-establishes-pre*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *overflow*: *card* (*find-pivots-seen-capped k cap d S B*) > *cap*
shows *bmssp-pre-full*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B*
⟨*proof*⟩

lemma *fp-iter-preserves-complete-sound*:
assumes *sound*: *sound-label d*
and *complete-x*: *d x* = *dist s x*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *fp-label* (*fp-iter n d F W B*) *x* = *dist s x*
⟨*proof*⟩

lemma *fp-iter-one-tight-step-completion*:
assumes *sound*: *sound-label d*
and *uF*: *u* $\in$ *F*
and *frontier-reaches*: $\bigwedge x.\ x \in F \Longrightarrow$ *reachable s x*
and *complete-u*: *d u* = *dist s u*
and *tight*: *tight-edge-step u v*
and *below-v*: *below-bound* (*dist s v*) *B*
shows *fp-label* (*fp-iter* (*Suc n*) *d F W B*) *v* = *dist s v*
and *v* $\in$ *fp-seen* (*fp-iter* (*Suc n*) *d F W B*)
⟨*proof*⟩

lemma *fp-iter-tight-path-completion*:
assumes *sound*: *sound-label d*
and *frontier-reaches*: $\bigwedge x.\ x \in F \Longrightarrow$ *reachable s x*
and *F-seen*: *F* $\subseteq$ *W*
and *nonempty*: *p* $\neq$ []

and *start*: *hd p* ∈ *F*
and *complete-start*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *below*: ⋀*x*. *x* ∈ *set p* ⟹ *below-bound* (*dist s x*) *B*
and *len*: *length* (*path-edges p*) ≤ *n*
shows *fp-label* (*fp-iter n d F W B*) (*last p*) = *dist s* (*last p*) ∧
last p ∈ *fp-seen* (*fp-iter n d F W B*)
⟨*proof*⟩

corollary *find-pivots-short-tight-path-complete*:
assumes *sound*: *sound-label d*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *nonempty*: *p* ≠ []
and *start*: *hd p* ∈ *S*
and *complete-start*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *below*: ⋀*x*. *x* ∈ *set p* ⟹ *below-bound* (*dist s x*) *B*
and *len*: *length* (*path-edges p*) ≤ *k*
shows *find-pivots-label k d S B* (*last p*) = *dist s* (*last p*)
and *last p* ∈ *find-pivots-seen k d S B*
⟨*proof*⟩

lemma *find-pivots-short-paths-complete-on*:
assumes *sound*: *sound-label d*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *witnesses*:
⋀*x*. *x* ∈ *X* ⟹ ∃ *p*. *p* ≠ [] ∧ *hd p* ∈ *S* ∧ *last p* = *x* ∧
d (*hd p*) = *dist s* (*hd p*) ∧ *successively tight-edge-step p* ∧
(∀ *y*∈*set p*. *below-bound* (*dist s y*) *B*) ∧ *length* (*path-edges p*) ≤ *k*
shows *complete-on* (*find-pivots-label k d S B*) *X* ∧
X ⊆ *find-pivots-seen k d S B*
⟨*proof*⟩

The path-completion lemmas are the semantic core of FindPivots. A short tight witness is a shortest-path prefix, already correct at its start, whose edges remain below the current bound. The round induction proves that such a witness is followed by the relaxation process: ⟦*sound-label ?d*; ⋀*x*. *x* ∈ *?S* ⟹ *reachable s x*; *?p* ≠ []; *hd ?p* ∈ *?S*; *?d* (*hd ?p*) = *local.dist s* (*hd ?p*); *successively tight-edge-step ?p*; ⋀*x*. *x* ∈ *set ?p* ⟹ *below-bound* (*local.dist s x*) *?B*; *length* (*path-edges ?p*) ≤ *?k*⟧ ⟹ *find-pivots-label ?k ?d ?S ?B* (*last ?p*) = *local.dist s* (*last ?p*)

⟦*sound-label ?d*; ⋀*x*. *x* ∈ *?S* ⟹ *reachable s x*; *?p* ≠ []; *hd ?p* ∈ *?S*; *?d* (*hd ?p*) = *local.dist s* (*hd ?p*); *successively tight-edge-step ?p*; ⋀*x*. *x* ∈ *set ?p* ⟹ *below-bound* (*local.dist s x*) *?B*; *length* (*path-edges ?p*) ≤ *?k*⟧ ⟹ *last ?p* ∈ *find-pivots-seen ?k ?d ?S ?B* records both the resulting label equality and membership in the seen set. The set-level lemma above packages the same fact for any family of witnesses. The next theorem instantiates that package with *find-pivots-short-vertices*, the set defined exactly by *short-tight-witness*,

yielding completeness via *complete-on*.

theorem *find-pivots-completes-short-vertices*:
  assumes *sound*: *sound-label d*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
  shows *complete-on* (*find-pivots-label k d S B*)
      (*find-pivots-short-vertices k d S B*) $\wedge$
    find-pivots-short-vertices k d S B $\subseteq$ *find-pivots-seen k d S B*
⟨*proof*⟩

lemma *card-set-take-distinct*:
  assumes *distinct xs* $k \leq$ *length xs*
  shows *card* (*set* (*take k xs*)) = *k*
  ⟨*proof*⟩

lemma *successively-take*:
  assumes *successively R xs*
  shows *successively R* (*take n xs*)
  ⟨*proof*⟩

lemma *successively-drop*:
  assumes *successively R xs*
  shows *successively R* (*drop n xs*)
  ⟨*proof*⟩

lemma *below-bound-le-trans*:
  assumes *a* $\leq$ *b below-bound b B*
  shows *below-bound a B*
  ⟨*proof*⟩

lemma *find-pivots-pivots-capped-label-below*:
  assumes *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
    and *xP*: *x* $\in$ *find-pivots-pivots-capped k cap d S B*
  shows *below-bound* (*find-pivots-label-capped k cap d S B x*) *B*
⟨*proof*⟩

lemma *find-pivots-completes-short-tree-suffix*:
  assumes *sound*: *sound-label d*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *reach-v*: *reachable s v*
    and *below-v*: *below-bound* (*dist s v*) *B*
    and *i*: *i* < *length* (*shortest-path-to v*)
    and *source-in-S*: *shortest-path-to v* ! *i* $\in$ *S*
    and *source-complete*: *d* (*shortest-path-to v* ! *i*) = *dist s* (*shortest-path-to v* ! *i*)
    and *suffix-len*: *length* (*path-edges* (*drop i* (*shortest-path-to v*))) $\leq$ *k*
  shows *find-pivots-label k d S B v* = *dist s v*
    and *v* $\in$ *find-pivots-seen k d S B*
⟨*proof*⟩

lemma *find-pivots-pivot-if-k-seen-tree-vertices*:

assumes *uS*: $u \in S$
and *distinct*: *distinct xs*
and *len*: $k \leq$ *length xs*
and *in-tree*: *set* (*take k xs*) $\subseteq$ *tree-of u*
and *seen*: *set* (*take k xs*) $\subseteq$ *find-pivots-seen k d S B*
shows $u \in$ *find-pivots-pivots k d S B*
⟨*proof*⟩

lemma *finite-find-pivots-pivots* [*simp*]:
assumes $S \subseteq V$
shows *finite* (*find-pivots-pivots k d S B*)
⟨*proof*⟩

lemma *find-pivots-pivots-subset*:
find-pivots-pivots k d S B $\subseteq S$
⟨*proof*⟩

theorem *find-pivots-pivots-card-times-le-seen*:
assumes *S-subset*: $S \subseteq V$
and *anti*: *tree-antichain S*
shows $k *$ *card* (*find-pivots-pivots k d S B*) $\leq$
card (*find-pivots-seen k d S B*)
⟨*proof*⟩

theorem *find-pivots-pivots-capped-card-times-le-seen-or-overflow*:
assumes *S-subset*: $S \subseteq V$
and *anti*: *tree-antichain S*
shows $k *$ *card* (*find-pivots-pivots-capped k cap d S B*) $\leq$
card (*find-pivots-seen-capped k cap d S B*) $\vee$
(*find-pivots-pivots-capped k cap d S B* $= S \wedge$
card (*find-pivots-seen-capped k cap d S B*) $>$ *cap*)
⟨*proof*⟩

lemma *find-pivots-long-tight-path-root-pivot*:
assumes *sound*: *sound-label d*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *uS*: $u \in S$
and *complete-u*: *d u* $=$ *dist s u*
and *p-nonempty*: $p \neq []$
and *hd-p*: *hd p* $= u$
and *distinct-p*: *distinct p*
and *len*: $k \leq$ *length p*
and *tight*: *successively tight-edge-step p*
and *below-prefix*:
$\bigwedge z.\ z \in$ *set* (*take k p*) $\Longrightarrow$ *below-bound* (*dist s z*) *B*
and *tree-prefix*: *set* (*take k p*) $\subseteq$ *tree-of u*
shows $u \in$ *find-pivots-pivots k d S B*
⟨*proof*⟩

lemma *find-pivots-long-tree-suffix-root-pivot*:
  assumes *sound*: *sound-label d*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *reach-v*: *reachable s v*
    and *below-v*: *below-bound* (*dist s v*) *B*
    and *i*: $i <$ *length* (*shortest-path-to v*)
    and *source-in-S*: *shortest-path-to v* ! *i* $\in S$
    and *source-complete*: *d* (*shortest-path-to v* ! *i*) = *dist s* (*shortest-path-to v* ! *i*)
    and *long*: $k \leq$ *length* (*drop i* (*shortest-path-to v*))
    and *tree-prefix*:
      set (*take k* (*drop i* (*shortest-path-to v*))) $\subseteq$
        tree-of (*shortest-path-to v* ! *i*)
  shows *shortest-path-to v* ! *i* $\in$ *find-pivots-pivots k d S B*
⟨*proof*⟩

corollary *find-pivots-one-tight-step-complete*:
  assumes *sound*: *sound-label d*
    and *uS*: $u \in S$
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *complete-u*: *d u* = *dist s u*
    and *tight*: *tight-edge-step u v*
    and *below-v*: *below-bound* (*dist s v*) *B*
  shows *find-pivots-label* (*Suc n*) *d S B v* = *dist s v*
    and $v \in$ *find-pivots-seen* (*Suc n*) *d S B*
  ⟨*proof*⟩

theorem *find-pivots-covers-bound-tree-complete-sources*:
  assumes *sound*: *sound-label d*
    and *S-complete*: *complete-on d S*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *v-bound*: $v \in$ *bound-tree S B*
  shows
    (*find-pivots-label k d S B v* = *dist s v* $\wedge$
      $v \in$ *find-pivots-seen k d S B*) $\vee$
     through-complete (*find-pivots-label k d S B*)
       (*find-pivots-pivots k d S B*) *v*
⟨*proof*⟩

theorem *find-pivots-post-complete-sources*:
  assumes *sound*: *sound-label d*
    and *S-subset*: $S \subseteq V$
    and *S-complete*: *complete-on d S*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
  shows *find-pivots-post d S B*
    (*find-pivots-label k d S B*)
    (*find-pivots-pivots k d S B*)
    {$v \in$ *bound-tree S B*. *find-pivots-label k d S B v* = *dist s v*}
⟨*proof*⟩

theorem *find-pivots-establishes-pivot-pre-concrete*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *bmssp-pre-full*
(*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *B*
⟨*proof*⟩

theorem *find-pivots-post-from-pre-concrete*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *find-pivots-post d S B*
(*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*)
{*v* ∈ *bound-tree S B*. *find-pivots-label k d S B v* = *dist s v*}
⟨*proof*⟩

theorem *find-pivots-capped-no-overflow-establishes-pre*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *no-overflow*: ¬ *card* (*find-pivots-seen-capped k cap d S B*) > *cap*
shows *bmssp-pre-full*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B*
⟨*proof*⟩

theorem *find-pivots-capped-establishes-pivot-pre-concrete*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *bmssp-pre-full*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B*
⟨*proof*⟩

The capped variant has two proof branches but one external contract. If the cap overflows, the paper keeps the original source set as the pivot set; the existing BMSSP precondition is therefore preserved. If it does not overflow, *card* (*fp-seen* (*fp-iter-capped ?n ?cap ?d ?F ?W ?B*)) ≤ *?cap* ⟹ *fp-iter-capped ?n ?cap ?d ?F ?W ?B* = *fp-iter ?n ?d ?F ?W ?B* identifies the capped execution with the uncapped one, so the pivot-precondition proof can reuse the short-path completeness argument. The theorem above hides that case split and exposes a single *bmssp-pre-full* fact for the capped labels and *find-pivots-pivots-capped*.

theorem *find-pivots-capped-post-from-pre-concrete*:

assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *find-pivots-post d S B*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*)
{$v \in$ *bound-tree S B*.
find-pivots-label-capped k cap d S B v = *dist s v*}
⟨*proof*⟩

end

end
theory *BMSSP-Pull-Minimum*
imports *BMSSP-Find-Pivots*
begin

21 Pull Minimum

This theory starts replacing the abstract partition-loop contract by concrete range-splitting lemmas. In the BMSSP loop, a partition Pull returns a boundary and a lower batch of candidate sources. The recursive child call is made below that boundary. To justify that call, we need to know that every incomplete vertex below the boundary is still covered by a complete source in the pulled lower part.

The rest of the theory is range algebra. It defines the half-open range tree between two distance boundaries, proves that a monotone boundary list partitions a bounded tree into disjoint child ranges, and assembles completeness of the lower prefix and every child range into the BMSSP postcondition. These lemmas are the semantic counterpart of the operational partition loop and are later used by the concrete one-step assembly theory.

context *unique-shortest-digraph*
begin

definition *split-below* **where**
split-below d S beta = {$x \in S.\ d\ x <$ *beta*}

lemma *split-below-label-below*:
assumes $x \in$ *split-below d S beta*
shows *below-bound* (*d x*) (*Fin beta*)
⟨*proof*⟩

lemma *split-below-tree-antichain*:
assumes *anti*: *tree-antichain S*
shows *tree-antichain* (*split-below d S beta*)
⟨*proof*⟩

lemma *split-below-scaled-card-le*:
  assumes *S-subset*: $S \subseteq V$
    and *S-k-cap*: $k * card\ S \leq cap$
  shows $k * card\ (split\text{-}below\ d\ S\ beta) \leq cap$
⟨*proof*⟩

The set *split-below* is the lower part produced by a Pull boundary: it keeps exactly those sources whose current labels are below *beta*. The first lemmas show that this split respects the finite bound, preserves the tree-antichain property, and cannot increase the scaled cardinality used by the FindPivots cap. These are small facts, but they are the side conditions needed before the lower recursive call can be formed.

definition *complete-tree-cover* **where**
  $complete\text{-}tree\text{-}cover\ d\ S\ B \longleftrightarrow (\forall v{\in}bound\text{-}tree\ S\ B.\ through\text{-}complete\ d\ S\ v)$

definition *range-tree* **where**
  $range\text{-}tree\ S\ a\ B =$
    $\{v \in tree\text{-}set\ S.\ a \leq dist\ s\ v \wedge below\text{-}bound\ (dist\ s\ v)\ B\}$

fun *nondecreasing-from* **where**
  $nondecreasing\text{-}from\ a\ [] \longleftrightarrow True$
| $nondecreasing\text{-}from\ a\ (b\ \#\ bs) \longleftrightarrow a \leq b \wedge nondecreasing\text{-}from\ b\ bs$

fun *bounds-le* **where**
  $bounds\text{-}le\ B\ [] \longleftrightarrow True$
| $bounds\text{-}le\ B\ (b\ \#\ bs) \longleftrightarrow bound\text{-}le\ (Fin\ b)\ B \wedge bounds\text{-}le\ B\ bs$

fun *range-tree-chain* **where**
  $range\text{-}tree\text{-}chain\ S\ a\ []\ B = range\text{-}tree\ S\ a\ B$
| $range\text{-}tree\text{-}chain\ S\ a\ (b\ \#\ bs)\ B =$
    $range\text{-}tree\ S\ a\ (Fin\ b) \cup range\text{-}tree\text{-}chain\ S\ b\ bs\ B$

fun *range-tree-chain-list* **where**
  $range\text{-}tree\text{-}chain\text{-}list\ S\ a\ []\ B = [range\text{-}tree\ S\ a\ B]$
| $range\text{-}tree\text{-}chain\text{-}list\ S\ a\ (b\ \#\ bs)\ B =$
    $range\text{-}tree\ S\ a\ (Fin\ b)\ \#\ range\text{-}tree\text{-}chain\text{-}list\ S\ b\ bs\ B$

The predicates *complete-tree-cover* and *range-tree* describe the two main geometric facts used by the loop. A complete tree cover says every vertex in a bounded tree is certified by a complete source. A range tree is the part of a source tree whose true distances lie between a lower boundary and an upper bound. The functions *range-tree-chain* and *range-tree-chain-list* extend this to a monotone list of child boundaries; the list form is convenient for relating one child postcondition to one child range.

lemma *below-bound-strict-trans*:
  assumes $a < b\ below\text{-}bound\ b\ B$
  shows $below\text{-}bound\ a\ B$
  ⟨*proof*⟩

lemma *complete-verticesD*:
assumes $u \in$ *complete-vertices* d
shows $u \in V$ *reachable* $s\ u\ d\ u =$ *dist* $s\ u$
⟨*proof*⟩

lemma *bound-tree-complete-vertices-eq*:
assumes *cover*:
$\bigwedge u.\ [\![u \in S;$ *reachable* $s\ u;$ *below-bound* (*dist* $s\ u$) $B;$
$d\ u \neq$ *dist* $s\ u]\!] \Longrightarrow$ *through-complete* $d\ S\ u$
shows *bound-tree* $S\ B =$ *bound-tree* $(S \cap$ *complete-vertices* $d)\ B$
⟨*proof*⟩

lemma *bmssp-pre-full-bound-tree-complete-vertices-eq*:
assumes *pre*: *bmssp-pre-full* $d\ S\ B$
shows *bound-tree* $S\ B =$ *bound-tree* $(S \cap$ *complete-vertices* $d)\ B$
⟨*proof*⟩

lemma *bound-tree-complete-vertices-eq-imp-cover*:
assumes *eq*: *bound-tree* $S\ B =$ *bound-tree* $(S \cap$ *complete-vertices* $d)\ B$
shows *complete-tree-cover* $d\ S\ B$
⟨*proof*⟩

lemma *bmssp-pre-full-complete-tree-cover*:
assumes *pre*: *bmssp-pre-full* $d\ S\ B$
shows *complete-tree-cover* $d\ S\ B$
⟨*proof*⟩

lemma *bmssp-pre-full-bound-mono*:
assumes *pre*: *bmssp-pre-full* $d\ S\ B$
and *le*: *bound-le* $B'\ B$
shows *bmssp-pre-full* $d\ S\ B'$
⟨*proof*⟩

theorem *concrete-find-pivots-complete-pivot-tree*:
assumes *sound*: *sound-label* d
and *pre*: *bmssp-pre-full* $d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
shows *bound-tree* (*find-pivots-pivots* $k\ d\ S\ B)\ B =$
bound-tree
(*find-pivots-pivots* $k\ d\ S\ B \cap$
complete-vertices (*find-pivots-label* $k\ d\ S\ B)) B$
⟨*proof*⟩

theorem *concrete-find-pivots-pivot-tree-cover*:
assumes *sound*: *sound-label* d
and *pre*: *bmssp-pre-full* $d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
shows *complete-tree-cover*

(*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *B*
⟨*proof*⟩

theorem *concrete-capped-find-pivots-complete-pivot-tree*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
shows *bound-tree* (*find-pivots-pivots-capped k cap d S B*) *B* =
bound-tree
(*find-pivots-pivots-capped k cap d S B* ∩
complete-vertices (*find-pivots-label-capped k cap d S B*)) *B*
⟨*proof*⟩

theorem *concrete-capped-find-pivots-pivot-tree-cover*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
shows *complete-tree-cover*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B*
⟨*proof*⟩

lemma *through-complete-obtain*:
assumes *through-complete d S v*
obtains *u p* **where**
u ∈ *S*
u ∈ *complete-vertices d*
shortest-walk s p v
u ∈ *set p*
d u = *dist s u*
dist s u ≤ *dist s v*
⟨*proof*⟩

lemma *pull-minimum-incomplete-covered*:
assumes *pre*: *bmssp-pre-full d S B*
and *beta*: *below-bound beta B*
and *vV*: *v* ∈ *V*
and *reach*: *reachable s v*
and *lt*: *dist s v* < *beta*
and *incomplete*: *d v* ≠ *dist s v*
shows *through-complete d* (*split-below d S beta*) *v*
⟨*proof*⟩

lemma *pull-minimum-pre-for-lower-split*:
assumes *pre*: *bmssp-pre-full d S B*
and *beta*: *below-bound beta B*
shows *bmssp-pre-full d* (*split-below d S beta*) (*Fin beta*)
⟨*proof*⟩

lemma *split-below-subset*:
split-below d S beta $\subseteq$ *S*
⟨*proof*⟩

lemma *range-tree-eq-bound-diff*:
range-tree S a B = *bound-tree S B* − *bound-tree S* (*Fin a*)
⟨*proof*⟩

lemma *range-tree-subset-bound-tree*:
range-tree S a B $\subseteq$ *bound-tree S B*
⟨*proof*⟩

lemma *finite-bound-tree* [*simp*]:
finite (*bound-tree S B*)
⟨*proof*⟩

lemma *finite-range-tree* [*simp*]:
finite (*range-tree S a B*)
⟨*proof*⟩

lemma *finite-range-tree-chain* [*simp*]:
finite (*range-tree-chain S a bs B*)
⟨*proof*⟩

lemma *finite-range-tree-chain-list-sets* [*simp*]:
assumes *X* $\in$ *set* (*range-tree-chain-list S a bs B*)
shows *finite X*
⟨*proof*⟩

lemma *Union-range-tree-chain-list*:
$\bigcup$(*set* (*range-tree-chain-list S a bs B*)) = *range-tree-chain S a bs B*
⟨*proof*⟩

lemma *range-tree-chain-lower-bound*:
assumes *mono*: *nondecreasing-from a bs*
and *x*: *x* $\in$ *range-tree-chain S a bs B*
shows *a* $\leq$ *dist s x*
⟨*proof*⟩

lemma *range-tree-disjoint-lower*:
bound-tree S (*Fin a*) $\cap$ *range-tree S a B* = {}
⟨*proof*⟩

lemma *bound-tree-disjoint-range-tree-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows *bound-tree S* (*Fin a*) $\cap$ *range-tree-chain S a bs B* = {}
⟨*proof*⟩

lemma *range-tree-disjoint-range-tree-chain-tail*:
  assumes *a-le-b*: $a \leq b$
    and *mono-tail*: *nondecreasing-from b bs*
  shows *range-tree S a* (*Fin b*) $\cap$ *range-tree-chain S b bs B* = {}
⟨*proof*⟩

lemma *card-range-tree-chain-eq-sum-list*:
  assumes *mono*: *nondecreasing-from a bs*
  shows *card* (*range-tree-chain S a bs B*) =
    sum-list (*map card* (*range-tree-chain-list S a bs B*))
  ⟨*proof*⟩

lemma *sum-list-map-mult-left*:
  fixes c :: *nat*
    and f :: $'a \Rightarrow$ *nat*
  shows *sum-list* (*map* ($\lambda x.$ $c * f\ x$) *xs*) = c * *sum-list* (*map f xs*)
  ⟨*proof*⟩

lemma *sum-list-map-card-mult-left*:
  fixes C :: *nat*
    and *Xs* :: $'v$ *set list*
  shows *sum-list* (*map* ($\lambda X.$ C * *card X*) *Xs*) = C * *sum-list* (*map card Xs*)
  ⟨*proof*⟩

lemma *sum-list-le-if-list-all2*:
  assumes *list-all2* ($\lambda c\ X.$ $c \leq C$ * *card X*) *costs Xs*
  shows *sum-list costs* $\leq$ *sum-list* (*map* ($\lambda X.$ C * *card X*) *Xs*)
  ⟨*proof*⟩

lemma *child-costs-le-range-tree-chain-card*:
  assumes *mono*: *nondecreasing-from a bs*
    and *costs*: *list-all2* ($\lambda c\ X.$ $c \leq C$ * *card X*) *costs*
      (*range-tree-chain-list S a bs B*)
  shows *sum-list costs* $\leq C$ * *card* (*range-tree-chain S a bs B*)
⟨*proof*⟩

lemma *range-tree-adjacent-disjoint*:
  range-tree S a (*Fin b*) $\cap$ *range-tree S b B* = {}
  ⟨*proof*⟩

lemma *range-tree-adjacent-union*:
  assumes $a \leq b$
    and *bound-le* (*Fin b*) *B*
  shows *range-tree S a* (*Fin b*) $\cup$ *range-tree S b B* = *range-tree S a B*
  ⟨*proof*⟩

lemma *bound-tree-split-at*:
  assumes *bound-le* (*Fin a*) *B*
  shows *bound-tree S B* = *bound-tree S* (*Fin a*) $\cup$ *range-tree S a B*

⟨*proof*⟩

lemma *bound-tree-adjacent-union*:
assumes $a \leq b$
and *bound-le* (*Fin b*) *B*
shows *bound-tree S* (*Fin a*) ∪ *range-tree S a* (*Fin b*) ∪ *range-tree S b B* =
bound-tree S B
⟨*proof*⟩

lemma *complete-on-Un*:
assumes *complete-on d A complete-on d B*
shows *complete-on d* (*A* ∪ *B*)
⟨*proof*⟩

lemma *complete-on-Union-list*:
assumes ⋀*X*. *X* ∈ *set Xs* ⟹ *complete-on d X*
shows *complete-on d* (⋃(*set Xs*))
⟨*proof*⟩

lemma *complete-on-bound-tree-split*:
assumes *bound-le* (*Fin a*) *B*
and *lower*: *complete-on d* (*bound-tree S* (*Fin a*))
and *upper*: *complete-on d* (*range-tree S a B*)
shows *complete-on d* (*bound-tree S B*)
⟨*proof*⟩

lemma *bound-tree-range-chain-union*:
assumes *mono*: *nondecreasing-from a bs*
and *bounds*: *bounds-le B* (*a* # *bs*)
shows *bound-tree S* (*Fin a*) ∪ *range-tree-chain S a bs B* = *bound-tree S B*
⟨*proof*⟩

lemma *card-bound-tree-range-chain-union*:
assumes *mono*: *nondecreasing-from a bs*
and *bounds*: *bounds-le B* (*a* # *bs*)
shows *card* (*bound-tree S B*) =
card (*bound-tree S* (*Fin a*)) + *card* (*range-tree-chain S a bs B*)
⟨*proof*⟩

lemma *complete-on-range-chain-union*:
assumes *mono*: *nondecreasing-from a bs*
and *bounds*: *bounds-le B* (*a* # *bs*)
and *lower*: *complete-on d* (*bound-tree S* (*Fin a*))
and *chain*: *complete-on d* (*range-tree-chain S a bs B*)
shows *complete-on d* (*bound-tree S B*)
⟨*proof*⟩

lemma *complete-on-range-tree-chain-list*:
assumes ⋀*X*. *X* ∈ *set* (*range-tree-chain-list S a bs B*) ⟹ *complete-on d X*

shows *complete-on d* (*range-tree-chain S a bs B*)
⟨*proof*⟩

theorem *partition-loop-trace-assembly-post*:
assumes *le*: *bound-le B′ B*
and *mono*: *nondecreasing-from a bs*
and *bounds*: *bounds-le B′* (*a # bs*)
and *lower*: *complete-on d′* (*bound-tree S* (*Fin a*))
and *children*: *list-all2* (λ*U X*. *U* = *X* ∧ *complete-on d′ U*) *Us*
(*range-tree-chain-list S a bs B′*)
and *U-def*: *U* = *bound-tree S* (*Fin a*) ∪ ⋃(*set Us*)
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

The range-chain lemmas culminate in ⟦*bound-le ?B′ ?B*; *nondecreasing-from ?a ?bs*; *bounds-le ?B′* (*?a # ?bs*); *complete-on ?d′* (*bound-tree ?S* (*Fin ?a*)); *list-all2* (λ*U X*. *U* = *X* ∧ *complete-on ?d′ U*) *?Us* (*range-tree-chain-list ?S ?a ?bs ?B′*); *?U* = *bound-tree ?S* (*Fin ?a*) ∪ ⋃ (*set ?Us*)⟧ ⟹ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U*. The hypotheses are exactly the certificate carried by the concrete partition trace: a returned bound below the input bound, nondecreasing child boundaries, child bounds below the returned bound, completeness of the lower prefix, and one completed set for each range in *range-tree-chain-list*. The theorem proves that their union is precisely *bound-tree* for the returned bound and that the final labels are complete there.

lemma *pull-minimum-bound-tree-lower-split*:
assumes *cover*: *complete-tree-cover d S* (*Fin beta*)
and *beta′-le*: *beta′* ≤ *beta*
shows *bound-tree S* (*Fin beta′*) = *bound-tree* (*split-below d S beta*) (*Fin beta′*)
⟨*proof*⟩

lemma *pull-recursive-post-lifts-bound-tree*:
assumes *cover*: *complete-tree-cover d S* (*Fin beta*)
and *post*: *bmssp-post-full d* (*split-below d S beta*) (*Fin beta*) *d′ B′ U*
shows *U* = *bound-tree S B′*
⟨*proof*⟩

theorem *concrete-find-pivots-pull-pre*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *beta*: *below-bound beta B*
shows *bmssp-pre-full*
(*find-pivots-label k d S B*)
(*split-below* (*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *beta*)
(*Fin beta*)
⟨*proof*⟩

theorem *concrete-capped-find-pivots-pull-pre*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *beta*: *below-bound beta B*
shows *bmssp-pre-full*
(*find-pivots-label-capped k cap d S B*)
(*split-below* (*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *beta*)
(*Fin beta*)
⟨*proof*⟩

theorem *concrete-find-pivots-pull-bound-tree-lower*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *beta*: *below-bound beta B*
and *beta′-le*: $beta' \leq beta$
shows *bound-tree* (*find-pivots-pivots k d S B*) (*Fin beta′*) =
bound-tree
(*split-below* (*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *beta*)
(*Fin beta′*)
⟨*proof*⟩

theorem *concrete-capped-find-pivots-pull-bound-tree-lower*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *beta*: *below-bound beta B*
and *beta′-le*: $beta' \leq beta$
shows *bound-tree* (*find-pivots-pivots-capped k cap d S B*) (*Fin beta′*) =
bound-tree
(*split-below* (*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *beta*)
(*Fin beta′*)
⟨*proof*⟩

theorem *concrete-find-pivots-pull-recursive-post-lifts*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *beta*: *below-bound beta B*
and *post*: *bmssp-post-full*
(*find-pivots-label k d S B*)
(*split-below* (*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *beta*)
(*Fin beta*) *d″ B′ U*

shows $U = \textit{bound-tree}\ (\textit{find-pivots-pivots}\ k\ d\ S\ B)\ B'$
⟨*proof*⟩

theorem *concrete-capped-find-pivots-pull-recursive-post-lifts*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \textit{reachable}\ s\ x$
and *beta*: *below-bound beta B*
and *post*: *bmssp-post-full*
(*find-pivots-label-capped k cap d S B*)
(*split-below* (*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *beta*)
(*Fin beta*) *d'' B' U*
shows $U = \textit{bound-tree}\ (\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B)\ B'$
⟨*proof*⟩

The Pull-minimum lemmas connect the range algebra to FindPivots. Once FindPivots establishes a pivot precondition, *split-below* can be used as the source set for a recursive call below a Pull boundary. If that recursive call returns a BMSSP postcondition for the split set, the lifting lemmas identify its output tree with the corresponding pivot tree. Both capped and uncapped FindPivots variants have parallel theorems because later operational relations use the capped executable search.

theorem *concrete-find-pivots-final-tree-assembly*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \textit{reachable}\ s\ x$
and *loop*: *bmssp-post-full*
(*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *B d-loop B' U-loop*
shows *U-loop* ∪
$\{v \in \textit{bound-tree}\ S\ B'.\ \textit{find-pivots-label}\ k\ d\ S\ B\ v = \textit{dist}\ s\ v\} =$
bound-tree S B'
⟨*proof*⟩

theorem *concrete-capped-find-pivots-final-tree-assembly*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \textit{reachable}\ s\ x$
and *loop*: *bmssp-post-full*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d-loop B' U-loop*
shows *U-loop* ∪
$\{v \in \textit{bound-tree}\ S\ B'.\ \textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B\ v = \textit{dist}\ s\ v\} =$
bound-tree S B'
⟨*proof*⟩

theorem *concrete-find-pivots-final-assembly-post*:
assumes *sound*: *sound-label d*

and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *loop*: *bmssp-post-full*
(*find-pivots-label k d S B*)
(*find-pivots-pivots k d S B*) *B d-final B′ U-loop*
and *compW*: *complete-on d-final*
$\{v \in$ *bound-tree S B′*. *find-pivots-label k d S B v* $=$ *dist s v*$\}$
defines $U \equiv$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B′*. *find-pivots-label k d S B v* $=$ *dist s v*$\}$
shows *bmssp-post-full d S B d-final B′ U*
⟨*proof*⟩

theorem *concrete-capped-find-pivots-final-assembly-post*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *loop*: *bmssp-post-full*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d-final B′ U-loop*
and *compW*: *complete-on d-final*
$\{v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$
defines $U \equiv$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$
shows *bmssp-post-full d S B d-final B′ U*
⟨*proof*⟩

The final assembly theorems are the facts used by *concrete-capped-bmssp-step*. The loop solves the pivot tree, while FindPivots may have completed additional vertices in the caller's original tree. The assembly theorem proves that the union of those two regions is the caller's returned bounded tree. The postcondition theorems then add completeness of the explicitly completed FindPivots region and conclude *bmssp-post-full* for the whole non-base step.

end

end
theory *BMSSP-Partition-Pull-Bridge*
imports *BMSSP-Pull-Minimum BMSSP-Partition-Interface*
begin

22 Partition Pull to BMSSP Pull Minimum

The partition interface is graph-independent: it knows about keys and labels, but not about shortest-path trees. BMSSP, on the other hand, phrases a child recursive call as a lower split of the current source set by a tentative distance bound. This small bridge theory identifies those two views.

The conversion is simple but important. A label function d and a source set S induce a partition view whose keys are exactly S and whose values are

the labels d. A pull from that view is therefore a pull by tentative distance. The theorems below prove that the pulled key set is exactly *split-below* at the returned bound, and that the BMSSP precondition descends from the parent problem to the child problem selected by the pull.

This file is used twice in the development. The sorted reference pull uses it to justify the executable specification path. The operational and bucketed theories use the abstract *pull-separates* statement, so they can reason about BMSSP recursion without mentioning the representation of the partition data structure.

context *unique-shortest-digraph*
begin

definition *label-partition-view* **where**
label-partition-view d S = (|*keys-of* = *S*, *value-of* = *d*|)

lemma *label-partition-view-keys* [*simp*]:
keys-of (*label-partition-view d S*) = *S*
⟨*proof*⟩

lemma *label-partition-view-value* [*simp*]:
value-of (*label-partition-view d S*) = *d*
⟨*proof*⟩

The first two theorems are the semantic translation. For the sorted reference operation, *sorted-pull-set-eq-split-below* unfolds the sorted pull and the label partition view to obtain the mathematical lower split. For any implementation satisfying the abstract separator predicate, *pull-separates-label-set-eq-split-below* gives the same conclusion from *pull-separates*. The latter is the theorem used by the bucketed implementation after it proves its pull operation realises the separator contract.

theorem *sorted-pull-set-eq-split-below*:
fixes *M* :: *nat*
and *Bmax* :: *real*
assumes *finite-S*: *finite S*
and *upper*: ⋀*u*. *u* ∈ *S* ⟹ *d u* < *Bmax*
and *D-def*: *D* = *label-partition-view d S*
and *beta-def*: *beta* = *sorted-pull-bound M Bmax D*
shows *sorted-pull-set M D* = *split-below d S beta*
⟨*proof*⟩

theorem *pull-separates-label-set-eq-split-below*:
fixes *Bmax* :: *real*
assumes *pull*: *pull-separates* (*label-partition-view d S*) *M Bmax S*′ *beta D*′
and *upper*: ⋀*u*. *u* ∈ *S* ⟹ *d u* < *Bmax*
shows *S*′ = *split-below d S beta*
⟨*proof*⟩

theorem *sorted-pull-establishes-lower-pre*:
  fixes M :: *nat*
    and *Bmax* :: *real*
  assumes *pre*: *bmssp-pre-full d S* (*Fin Bmax*)
    and *upper*: $\bigwedge u.\ u \in S \Longrightarrow d\ u < Bmax$
    and *D-def*: D = *label-partition-view d S*
    and S'*-def*: S' = *sorted-pull-set M D*
    and *beta-def*: *beta* = *sorted-pull-bound M Bmax D*
  shows *bmssp-pre-full d* S' (*Fin beta*)
⟨*proof*⟩

The final theorem packages the precondition transfer needed by a recursive BMSSP child call. If the parent problem is valid up to *Bmax* and all current source labels lie below *Bmax*, then the set returned by a sorted pull is valid up to the returned child bound. The all-keys case is just the parent precondition again; the proper-prefix case uses the pull-minimum lemma for lower splits.

Later operational proofs use this theorem as a template for arbitrary partition implementations: once a pull is known to select the same lower split, the recursive child precondition follows from the same argument.

end

end
theory *BMSSP-Initialization*
  imports *BMSSP-Shortest-Path-Lemmas*
begin

23 Initial Labels

BMSSP starts from the usual shortest-path initialization: the source is complete and all other labels are conservative upper bounds. In an imperative implementation one would normally write this as distance zero for the source and infinity for every other vertex. The abstract correctness layer in this entry uses real-valued labels rather than an extended-real datatype, so this theory gives a finite replacement for infinity.

The replacement is the maximum true source distance over the finite vertex set. This is not intended as an executable way to discover shortest paths; it is a semantic initial label used in the proof stack. Because the graph locale is finite, the maximum exists. Because every reachable vertex has distance at most that maximum, the label function is sound. The source remains exact because the distance from any vertex to itself is zero under non-negative edge weights.

The lemmas below isolate precisely the facts needed by later top-level theorems: the source label is complete, the whole label function is sound, and the root call with source set $\{s\}$ and bound *Infinity* satisfies the BMSSP precondition.

context *finite-weighted-digraph*
begin

definition *finite-initial-label* **where**
finite-initial-label v = (*if v* = *s then 0 else Max* (*dist s* ‘ *V*))

The definition *finite-initial-label* is intentionally semantic. Its non-source value mentions *local.dist*, so it is not the data structure used by the exported example. Its role is to let the correctness proof start from a finite real-valued label function while preserving the same logical shape as Dijkstra's standard initialization.

lemma *dist-refl-zero*:
assumes *v* ∈ *V*
shows *dist v v* = *0*
⟨*proof*⟩

lemma *finite-initial-label-source-complete*:
finite-initial-label s = *dist s s*
⟨*proof*⟩

The proof of *?v* ∈ *V* ⟹ *local.dist ?v ?v* = *0* uses the one-vertex simple walk for the upper bound and non-negativity of walk weights for the lower bound. From it, *finite-initial-label s* = *local.dist s s* gives the exact source label required by root BMSSP initialization. The two soundness lemmas below differ only in whether reachability of every vertex is supplied explicitly or left as the local reachable premise of *sound-label*.

lemma *finite-initial-label-sound*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
shows *sound-label finite-initial-label*
⟨*proof*⟩

lemma *finite-initial-label-sound-reachable*:
shows *sound-label finite-initial-label*
⟨*proof*⟩

lemma *initialized-root-pre-full*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
shows *bmssp-pre finite-initial-label* {*s*} *Infinity*
⟨*proof*⟩

Finally, (⋀*v*. *v* ∈ *V* ⟹ *reachable s v*) ⟹ *bmssp-pre finite-initial-label* {*s*} *Infinity* packages the initialized labels for the root abstract BMSSP call. The source set is the singleton {*s*}, the bound is *Infinity*, and every reachable vertex lies on a shortest path through the source. Later theories strengthen this precondition to the complete-source form and then discharge the full algorithmic step relation.

end

end
theory *BMSSP-Concrete-Step*
imports *BMSSP-Pull-Minimum*
begin

24 Concrete One-Step BMSSP Assembly

This theory packages the concrete part of a non-base BMSSP call. The abstract correctness skeleton states its recursive middle phase with the predicate *partition-loop-post*: after FindPivots has chosen a pivot set, the loop must solve the bounded tree induced by those pivots. That statement is intentionally compact, but it hides the range structure that the implementation actually follows.

The concrete step exposed here replaces that abstract loop contract by an explicit monotone trace of child ranges. A trace contains the first lower boundary, the list of subsequent boundaries, the label function after the loop, and the collection of vertex sets solved by the child calls. The trace proof checks that the boundaries are nondecreasing, remain below the returned BMSSP bound, and cover exactly the chain of bounded shortest-path trees that the partition loop is supposed to assemble.

The final step definitions then restore FindPivots. The uncapped version uses the conceptual FindPivots operation, while the capped version uses the executable bounded search from the paper. Both definitions have the same shape: perform FindPivots, solve the pivot loop, keep the vertices completed directly by FindPivots, and return their union. The two correctness theorems below show that this concrete assembly is still a valid *bmssp-post-full* result.

context *unique-shortest-digraph*
begin

definition *concrete-partition-loop-trace* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U ⟷
bound-le B′ B ∧
nondecreasing-from a bs ∧
bounds-le B′ (*a* # *bs*) ∧
complete-on d′ (*bound-tree P* (*Fin a*)) ∧
list-all2 (λ*U X*. *U* = *X* ∧ *complete-on d′ U*) *Us*
(*range-tree-chain-list P a bs B′*) ∧
U = *bound-tree P* (*Fin a*) ∪ ⋃(*set Us*)

theorem *concrete-partition-loop-trace-post*:
assumes *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
shows *bmssp-post-full d P B d′ B′ U*
⟨*proof*⟩

definition *concrete-bmssp-step* **where**

```
  concrete-bmssp-step k d S B a bs d' B' Us U ⟷
    (let d-fp = find-pivots-label k d S B;
         P = find-pivots-pivots k d S B;
         W = {v ∈ bound-tree S B'. d-fp v = dist s v}
     in ∃ U-loop.
         concrete-partition-loop-trace P B a bs d' B' Us U-loop ∧
         complete-on d' W ∧
         U = U-loop ∪ W)

definition concrete-capped-bmssp-step where
  concrete-capped-bmssp-step k cap d S B a bs d' B' Us U ⟷
    (let d-fp = find-pivots-label-capped k cap d S B;
         P = find-pivots-pivots-capped k cap d S B;
         W = {v ∈ bound-tree S B'. d-fp v = dist s v}
     in ∃ U-loop.
         concrete-partition-loop-trace P B a bs d' B' Us U-loop ∧
         complete-on d' W ∧
         U = U-loop ∪ W)
```

The definition *concrete-partition-loop-trace* is the local certificate for the recursive partition loop. It does not say how pulls are generated; it says what must be true of the resulting range sequence. The theorem *concrete-partition-loop-trace ?P ?B ?a ?bs ?d' ?B' ?Us ?U ⟹ bmssp-post-full ?d ?P ?B ?d' ?B' ?U* then converts such a trace into *bmssp-post-full* for the pivot set. The step predicates *concrete-bmssp-step* and *concrete-capped-bmssp-step* wrap the trace with FindPivots and the final union of loop-completed and FindPivots-completed vertices.

```
theorem concrete-bmssp-step-correct:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and step: concrete-bmssp-step k d S B a bs d' B' Us U
  shows bmssp-post-full d S B d' B' U
⟨proof⟩

theorem concrete-capped-bmssp-step-correct:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and step: concrete-capped-bmssp-step k cap d S B a bs d' B' Us U
  shows bmssp-post-full d S B d' B' U
⟨proof⟩

end

end
theory BMSSP-Concrete-Top-Level
  imports BMSSP-Initialization BMSSP-Concrete-Step
begin
```

25 Concrete Top-Level Correctness

These theorems connect the finite initial labels to the concrete one-step BMSSP assembly relation. They avoid the abstract *abstract-bmssp-step* contract used by the abstract interface.

The top-level SSSP call is a BMSSP call with singleton source set $\{s\}$ and infinite bound. The finite initial label gives distance *0* at the source and an effectively infinite value elsewhere. Once a concrete BMSSP step proves the full BMSSP postcondition for this root problem, the infinite-bound tree is the whole vertex set, so completeness of the returned label function is exactly SSSP correctness.

Two entry points are provided because the development has both an uncapped concrete step and the capped step used by the paper's parameter schedule. They share the same proof: establish the root precondition, invoke the corresponding concrete-step correctness theorem, then collapse the root BMSSP postcondition to *sssp-correct*.

context *unique-shortest-digraph*
begin

theorem *finite-initial-label-concrete-top-level-correct*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *step*: *concrete-bmssp-step k finite-initial-label* $\{s\}$ *Infinity a bs*
      d' *Infinity Us U*
  shows *sssp-correct* d'
⟨*proof*⟩

theorem *finite-initial-label-concrete-capped-top-level-correct*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *step*: *concrete-capped-bmssp-step k cap finite-initial-label* $\{s\}$ *Infinity a bs*
      d' *Infinity Us U*
  shows *sssp-correct* d'
⟨*proof*⟩

end

end
theory *BMSSP-Recursive*
  imports *BMSSP-Base-Case BMSSP-Concrete-Top-Level*
begin

26 Level-Indexed BMSSP Relation

This theory introduces the first level-indexed BMSSP relation. Earlier theories prove a base case and a concrete non-base step in isolation. The inductive relation below is the small bridge that turns those one-step facts into a recursive BMSSP run indexed by a natural recursion level.

The relation is deliberately simple. At level zero it uses the checked base case for a singleton source set. At a successor level it accepts any concrete capped BMSSP step assembled from FindPivots and a monotone partition trace. This file does not yet model the operational generation of every pull and child call; that more detailed relation is proved separately in *BMSSP-Operational-Pull*. Here the purpose is to show that the base/step decomposition is already enough for semantic correctness.

The main theorem *concrete-bmssp-correct* below is an induction over the level-indexed relation. The base rule delegates to the base-case postcondition, and the step rule delegates to the concrete capped step theorem. The final theorem specializes the relation to the root source and infinite bound, obtaining ordinary single-source shortest-path correctness for the finite initial label.

context *unique-shortest-digraph*
begin

inductive *concrete-bmssp* **where**
Base:
$S = \{x\} \Longrightarrow$
concrete-bmssp k cap 0 d S B
(λv. *if* $v \in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
| *Step*:
concrete-capped-bmssp-step k cap d S B a bs d′ B′ Us U $\Longrightarrow$
concrete-bmssp k cap (*Suc l*) *d S B d′ B′ U*

The two rules of *concrete-bmssp* mirror the recursive structure of the paper. The base rule solves the bounded region reachable from one source by the base-case routine. The successor rule abstracts over the child runs already summarized by *concrete-capped-bmssp-step*. This keeps the induction proof short and makes clear which theorem is responsible for each layer of the algorithm.

lemma *bmssp-post-imp-post-full*:
assumes *bmssp-post d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

theorem *concrete-bmssp-correct*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *run*: *concrete-bmssp k cap l d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

The proof of ⟦*sound-label ?d*; *bmssp-pre-full ?d ?S ?B*; $\bigwedge x.\ x \in$ *?S* $\Longrightarrow$

reachable s x; *concrete-bmssp ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U*⟧ ⟹ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* is intentionally direct. All shortest-path reasoning has already been packaged in the base-case and concrete-step theories, so the induction only selects the appropriate correctness theorem for each rule. The remaining top-level theorem then supplies the standard root precondition, soundness of the initial label, and reachability of the singleton source set.

theorem *finite-initial-label-recursive-top-level-correct*:
  assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
    and *run*: *concrete-bmssp k cap l finite-initial-label* {*s*} *Infinity d′ Infinity U*
  shows *sssp-correct d′*
⟨*proof*⟩

end

end
theory *BMSSP-Operational-Pull*
  imports *BMSSP-Partition-Pull-Bridge BMSSP-Recursive*
begin

27 Operational Pull Step

This theory is the bridge from the abstract recursive BMSSP specification to an operational loop that explicitly pulls a bounded prefix out of a partition, recurses on that prefix, and then pushes newly discovered labels into the remaining partition. Earlier theories prove the abstract correctness theorem for *concrete-bmssp*; here we expose the loop shape that the later cost relations charge.

The paper's partition loop is not merely a list traversal. Each iteration selects a child source set below a temporary bound, runs BMSSP recursively on that child problem, and then refreshes the parent partition with two kinds of labels: edge relaxations out of the completed child tree and old source labels that now fall into the open range just processed. The predicate *pull-separates* is the abstract statement that the pull operation has found exactly such a prefix and retained an ordered residual partition.

The main proof obligation is a lifting argument. If a child recursive call satisfies *bmssp-post-full* for the split prefix, and if the parent already satisfied *bmssp-pre-full* up to the larger bound, then the child result can be viewed as progress for the parent. The connective tissue is the lower split *split-below*: it identifies the prefix chosen by the data structure with the mathematical set of vertices whose current labels are below the recursive child bound.

Later in the file the same idea is rephrased as inductive operational traces. The traces remember the sequence of child output bounds, the range trees *range-tree* that those bounds cut out, and the final tree assembled from all slices. This is the form needed by the range-synchronised cost theories, where

every loop iteration must be charged to the slice it actually completed.

context *unique-shortest-digraph*
begin

theorem *sorted-pull-recursive-child-lifts*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S* (*Fin Bmax*)
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *upper*: $\bigwedge u.\ u \in S \Longrightarrow$ *d u* $<$ *Bmax*
    and *S'-def*: *S'* = *sorted-pull-set M* (*label-partition-view d S*)
    and *beta-def*: *beta* = *sorted-pull-bound M Bmax* (*label-partition-view d S*)
    and *run*: *concrete-bmssp k cap l d S'* (*Fin beta*) *d' B' U*
  shows *bmssp-post-full d S* (*Fin Bmax*) *d' B' U*
⟨*proof*⟩

theorem *pull-separates-recursive-child-lifts*:
  fixes *Bmax* :: *real*
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S* (*Fin Bmax*)
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *upper*: $\bigwedge u.\ u \in S \Longrightarrow$ *d u* $<$ *Bmax*
    and *pull*: *pull-separates* (*label-partition-view d S*) *M Bmax S' beta D'*
    and *run*: *concrete-bmssp k cap l d S'* (*Fin beta*) *d' B' U*
  shows *bmssp-post-full d S* (*Fin Bmax*) *d' B' U*
⟨*proof*⟩

The first two lifting lemmas isolate the correctness content of a pull. The theorem $[\![$*sound-label ?d*; *bmssp-pre-full ?d ?S* (*Fin ?Bmax*); $\bigwedge x.\ x \in$ *?S* $\Longrightarrow$ *reachable s x*; $\bigwedge u.\ u \in$ *?S* $\Longrightarrow$ *?d u* $<$ *?Bmax*; *?S'* = *sorted-pull-set ?M* (*label-partition-view ?d ?S*); *?beta* = *sorted-pull-bound ?M ?Bmax* (*label-partition-view ?d ?S*); *concrete-bmssp ?k ?cap ?l ?d ?S'* (*Fin ?beta*) *?d' ?B' ?U*$]\!] \Longrightarrow$ *bmssp-post-full ?d ?S* (*Fin ?Bmax*) *?d' ?B' ?U* is phrased for the executable sorted-list partition view: the child set and child bound are computed by sorting labels and taking the first M positions. The theorem $[\![$*sound-label ?d*; *bmssp-pre-full ?d ?S* (*Fin ?Bmax*); $\bigwedge x.\ x \in$ *?S* $\Longrightarrow$ *reachable s x*; $\bigwedge u.\ u \in$ *?S* $\Longrightarrow$ *?d u* $<$ *?Bmax*; *pull-separates* (*label-partition-view ?d ?S*) *?M ?Bmax ?S' ?beta ?D'*; *concrete-bmssp ?k ?cap ?l ?d ?S'* (*Fin ?beta*) *?d' ?B' ?U*$]\!] \Longrightarrow$ *bmssp-post-full ?d ?S* (*Fin ?Bmax*) *?d' ?B' ?U* is the abstract version used by the bucketed partition and by the costed loop relations.

Both statements follow the same pattern. The pulled set is shown to equal the mathematical split below the child bound, the child precondition is inherited from the parent precondition, and the child postcondition is lifted through the parent's complete tree cover. The conclusion deliberately reuses *bmssp-post-full*; nothing about the later operational or costed presentation changes the semantic contract of BMSSP.

definition *complete-preserved* **where**

complete-preserved d d′ U ⟷ (*complete-on d U* ⟶ *complete-on d′ U*)

lemma *relax-edges-complete-preserved*:
assumes *sound*: *sound-label d*
and *edges*: ⋀*u v*. (*u*, *v*) ∈ *set es* ⟹ (*u*, *v*) ∈ *E*
and *reaches*: ⋀*u v*. (*u*, *v*) ∈ *set es* ⟹ *reachable s u*
shows *complete-preserved d* (*relax-edges d es*) *U*
⟨*proof*⟩

lemma *relax-frontier-complete-preserved*:
assumes *sound*: *sound-label d*
and *reaches*: ⋀*u*. *u* ∈ *F* ⟹ *reachable s u*
shows *complete-preserved d* (*relax-frontier d F*) *U*
⟨*proof*⟩

After a child call returns, the parent loop relaxes edges out of the completed child tree and may also reinsert source labels from the pulled set. The small predicate *complete-preserved* packages the only semantic requirement imposed on that refresh: labels already complete on the child slice stay complete after the refresh.

The lemmas ⟦*sound-label ?d*; ⋀*u v*. (*u*, *v*) ∈ *set ?es* ⟹ (*u*, *v*) ∈ *E*; ⋀*u v*. (*u*, *v*) ∈ *set ?es* ⟹ *reachable s u*⟧ ⟹ *complete-preserved ?d* (*relax-edges ?d ?es*) *?U* and ⟦*sound-label ?d*; ⋀*u*. *u* ∈ *?F* ⟹ *reachable s u*⟧ ⟹ *complete-preserved ?d* (*relax-frontier ?d ?F*) *?U* are deliberately local and modest. They do not prove a new shortest-path theorem; they just record that ordinary sound relaxation cannot destroy a vertex whose label has already reached its true distance. This lets the loop proofs compose recursive completion with parent-side batch updates without reopening the relaxation algebra every time.

definition *edge-relaxation-pairs-between* **where**
edge-relaxation-pairs-between d U L H =
map (λ(*u*, *v*). (*v*, *d u* + *w u v*))
(*filter* (λ(*u*, *v*). *L* ≤ *d u* + *w u v* ∧ *d u* + *w u v* < *H*)
(*edge-list-of* (*outgoing-edges U*)))

definition *label-pairs-between* **where**
label-pairs-between d S L H =
map (λ*x*. (*x*, *d x*))
(*filter* (λ*x*. *L* ≤ *d x* ∧ *d x* < *H*)
(*partition-key-order* (*label-partition-view d S*)))

lemma *edge-relaxation-pairs-between-value-le-high*:
assumes (*x*, *b*) ∈ *set* (*edge-relaxation-pairs-between d U L H*)
shows *b* ≤ *H*
⟨*proof*⟩

lemma *edge-relaxation-pairs-between-value-ge-low*:
assumes (*x*, *b*) ∈ *set* (*edge-relaxation-pairs-between d U L H*)

shows $L \leq b$
⟨*proof*⟩

lemma *label-pairs-between-value-le-high*:
assumes $(x, b) \in$ *set* (*label-pairs-between d S L H*)
shows $b \leq H$
⟨*proof*⟩

lemma *label-pairs-between-value-ge-low*:
assumes $(x, b) \in$ *set* (*label-pairs-between d S L H*)
shows $L \leq b$
⟨*proof*⟩

lemma *edge-relaxation-pairs-between-length-le-outgoing*:
length (*edge-relaxation-pairs-between d U L H*) ≤
card (*outgoing-edges U*)
⟨*proof*⟩

lemma *label-pairs-between-length-le-card*:
assumes *finite-S*: *finite S*
shows *length* (*label-pairs-between d S L H*) ≤ *card S*
⟨*proof*⟩

theorem *pull-separates-batch-prepend-for-relaxation-pairs*:
assumes *pull*: *pull-separates D M B S-pull beta D′*
shows *batch-prepend-admissible D′*
(*edge-relaxation-pairs-between d U L beta* @ *label-pairs-between d S L beta*)
⟨*proof*⟩

The two batch constructors separate the sources of new partition entries. *edge-relaxation-pairs-between* enumerates outgoing child-tree edges whose relaxed values lie in the current open range, while *label-pairs-between* keeps old source labels that also lie in that range. The length lemmas above connect these lists to the quantities charged later: outgoing edges for the first list and source cardinality for the second.

The admissibility theorem *pull-separates ?D ?M ?B ?S-pull ?beta ?D′* ⟹ *batch-prepend-admissible ?D′* (*edge-relaxation-pairs-between ?d ?U ?L ?beta* @ *label-pairs-between ?d ?S ?L ?beta*) is the key local invariant for pushing the batch into the residual partition. It says that every generated value is still below the bound exposed by the pull, so the residual ordered partition can accept the batch without invalidating the next pull.

lemma *range-tree-same-empty* [*simp*]:
range-tree S a (*Fin a*) = {}
⟨*proof*⟩

The relation below is the first operational presentation of the parent loop. It abstracts away the concrete partition representation, but it records the same sequence of events as the algorithm: stop when the lower bound has

already produced enough completed vertices, otherwise recurse on the split prefix, preserve completion through the parent refresh, and continue from the child output bound.

The lists *betas*, *bs*, and *Us* are not auxiliary clutter. They are the trace that later cost proofs consume. The child output bounds *bs* determine adjacent range slices, and each stored set in *Us* is identified with the corresponding *range-tree*. Proving this trace property once keeps the later runtime arguments independent of the details of the recursive correctness proof.

inductive *operational-partition-loop* **where**
  Done:
    bound-le (*Fin a*) *B* $\Longrightarrow$
    complete-on d′ (*bound-tree P* (*Fin a*)) $\Longrightarrow$
    operational-partition-loop k cap l d P B d′ a [] [] (*Fin a*)
      [*range-tree P a* (*Fin a*)]
      (*bound-tree P* (*Fin a*) $\cup$ $\bigcup$(*set* [*range-tree P a* (*Fin a*)]))
| *Step*:
    below-bound beta B $\Longrightarrow$
    a $\leq$ *b* $\Longrightarrow$
    complete-on d′ (*bound-tree P* (*Fin a*)) $\Longrightarrow$
    concrete-bmssp k cap l d (*split-below d P beta*) (*Fin beta*)
      d-child (*Fin b*) *U-child* $\Longrightarrow$
    complete-preserved d-child d′ U-child $\Longrightarrow$
    operational-partition-loop k cap l d P B d′ b betas bs B′ Us-tail U-tail $\Longrightarrow$
    operational-partition-loop k cap l d P B d′ a (*beta* # *betas*) (*b* # *bs*) *B′*
      (*range-tree P a* (*Fin b*) # *Us-tail*)
      (*bound-tree P* (*Fin a*) $\cup$ $\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)))

theorem *operational-partition-loop-trace*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d P B*
    and *P-reaches*: $\bigwedge$*x*. *x* $\in$ *P* $\Longrightarrow$ *reachable s x*
    and *loop*: *operational-partition-loop k cap l d P B d′ a betas bs B′ Us U*
  shows *concrete-partition-loop-trace P B a bs d′ B′ Us U*
  ⟨*proof*⟩

definition *operational-capped-bmssp-step* **where**
  operational-capped-bmssp-step k cap l d S B a betas bs d′ B′ Us U $\longleftrightarrow$
    (*let d-fp* = *find-pivots-label-capped k cap d S B*;
        P = *find-pivots-pivots-capped k cap d S B*;
        W = {*v* $\in$ *bound-tree S B′*. *d-fp v* = *dist s v*}
     in $\exists$ *U-loop*.
        operational-partition-loop k cap l d-fp P B d′ a betas bs B′ Us U-loop $\wedge$
        complete-on d′ W $\wedge$
        U = *U-loop* $\cup$ *W*)

theorem *operational-capped-bmssp-step-correct*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S B*

and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *step*: *operational-capped-bmssp-step k cap l d S B a betas bs d′ B′ Us U*
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

inductive *operational-bmssp* **where**
Base:
$S = \{x\} \Longrightarrow$
operational-bmssp k cap 0 d S B
(λv. *if* $v \in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
| *Step*:
operational-capped-bmssp-step k cap l d S B a betas bs d′ B′ Us U $\Longrightarrow$
operational-bmssp k cap (*Suc l*) *d S B d′ B′ U*

theorem *operational-bmssp-correct*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *run*: *operational-bmssp k cap l d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

theorem *finite-initial-label-operational-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*: *operational-bmssp k cap l finite-initial-label* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
⟨*proof*⟩

The plain operational relation still calls *concrete-bmssp* in each loop step. The full relation below unfolds that recursive call as another inductive branch. This mutual presentation is more verbose, but it gives the cost layer a single derivation tree whose nodes are either BMSSP calls or partition-loop iterations.

The theorem ⟦$\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*; *operational-bmssp ?k ?cap ?l finite-initial-label* {*s*} *Infinity ?d′ Infinity ?U*⟧ $\Longrightarrow$ *sssp-correct ?d′* remains the semantic endpoint for this layer: starting from the finite initial label at the source and an infinite top-level bound, any operational run computes a full single-source shortest-path labelling. The later relations refine this one by adding costs and range synchronisation; they do not change this correctness statement.

inductive *full-operational-partition-loop* **and** *full-operational-bmssp* **where**
Full-Loop-Done:
bound-le (*Fin a*) *B* $\Longrightarrow$
complete-on d′ (*bound-tree P* (*Fin a*)) $\Longrightarrow$
full-operational-partition-loop k cap l d P B d′ a [] [] (*Fin a*)
[*range-tree P a* (*Fin a*)]

(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)]))
| *Full-Loop-Step*:
below-bound beta B ⟹
a ≤ *b* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
full-operational-bmssp k cap l d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin b*) *U-child* ⟹
complete-preserved d-child d′ U-child ⟹
full-operational-partition-loop k cap l d P B d′ b betas bs B′ Us-tail U-tail ⟹
full-operational-partition-loop k cap l d P B d′ a (*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)))
| *Full-Loop-Step-Pre*:
bound-le (*Fin beta*) *B* ⟹
bmssp-pre-full d (*split-below d P beta*) (*Fin beta*) ⟹
a ≤ *b* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
full-operational-bmssp k cap l d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin b*) *U-child* ⟹
complete-preserved d-child d′ U-child ⟹
full-operational-partition-loop k cap l d P B d′ b betas bs B′ Us-tail U-tail ⟹
full-operational-partition-loop k cap l d P B d′ a (*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)))
| *Full-Base*:
S = {*x*} ⟹
full-operational-bmssp k cap 0 d S B
(λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
| *Full-Step*:
full-operational-partition-loop k cap l
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ a betas bs B′ Us U-loop* ⟹
complete-on d′
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
U = *U-loop* ∪
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
full-operational-bmssp k cap (*Suc l*) *d S B d′ B′ U*

theorem *full-operational-partition-loop-trace*:
full-operational-partition-loop k cap l d P B d′ a betas bs B′ Us U ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
concrete-partition-loop-trace P B a bs d′ B′ Us U
and *full-operational-bmssp-correct*:
full-operational-bmssp k cap l d S B d′ B′ U ⟹
sound-label d ⟹

bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*) $\Longrightarrow$
bmssp-post-full d S B d' B' U
⟨*proof*⟩

theorem *finite-initial-label-full-operational-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*: *full-operational-bmssp k cap l finite-initial-label* {*s*} *Infinity d' Infinity U*
shows *sssp-correct d'*
⟨*proof*⟩

inductive *full-operational-partition-loop-state* **where**
State-Done:
keys-of D = {} $\Longrightarrow$
bound-le (*Fin a*) *B* $\Longrightarrow$
complete-on d' (*bound-tree P* (*Fin a*)) $\Longrightarrow$
full-operational-partition-loop-state M t k cap l d P B d' D a [] [] (*Fin a*)
[*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ $\bigcup$(*set* [*range-tree P a* (*Fin a*)])) *0*
| *State-Step*:
pull-separates D M Bmax S-pull beta D-pull $\Longrightarrow$
bound-le (*Fin beta*) *B* $\Longrightarrow$
bmssp-pre-full d (*split-below d P beta*) (*Fin beta*) $\Longrightarrow$
S-pull = *split-below d P beta* $\Longrightarrow$
a ≤ *b* $\Longrightarrow$
complete-on d' (*bound-tree P* (*Fin a*)) $\Longrightarrow$
full-operational-bmssp k cap l d S-pull (*Fin beta*) *d-child* (*Fin b*) *U-child* $\Longrightarrow$
complete-preserved d-child d' U-child $\Longrightarrow$
batch =
edge-relaxation-pairs-between d-child U-child b beta @
label-pairs-between d S-pull b beta $\Longrightarrow$
D-next = *batch-min-update D-pull batch* $\Longrightarrow$
partition-pull-cost-bound c-pull S-pull $\Longrightarrow$
partition-batch-cost-bound c-batch t batch $\Longrightarrow$
full-operational-partition-loop-state M t k cap l d P B d' D-next b betas bs B'
Us-tail U-tail c-tail $\Longrightarrow$
c = *c-pull* + *c-batch* + *c-child* + *c-tail* $\Longrightarrow$
full-operational-partition-loop-state M t k cap l d P B d' D a (*beta* # *betas*) (*b* # *bs*) *B'*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪ $\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c*

lemma *full-operational-partition-loop-state-lengths*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d' D a
betas bs B' Us U c
shows *length betas* = *length bs*
and *length Us* = *Suc* (*length bs*)

⟨*proof*⟩

theorem *full-operational-partition-loop-state-refines*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
shows *full-operational-partition-loop k cap l d P B d′ a betas bs B′ Us U*
⟨*proof*⟩

theorem *full-operational-partition-loop-state-cost-bound*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
shows ∃ *child-costs pulls batches*.
c ≤ *sum-list child-costs* + *sum-list* (*map* (λ*S*. *card* (*S* :: *′a set*)) *pulls*) +
t ∗ *sum-list* (*map* (λ*batch*. *length* (*batch* :: (*′a* × *real*) *list*)) *batches*)
⟨*proof*⟩

theorem *full-operational-partition-loop-state-trace-and-cost*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
obtains *child-costs pulls batches* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
c ≤ *sum-list child-costs* + *sum-list* (*map* (λ*S*. *card* (*S* :: *′a set*)) *pulls*) +
t ∗ *sum-list* (*map* (λ*batch*. *length* (*batch* :: (*′a* × *real*) *list*)) *batches*)
⟨*proof*⟩

theorem *full-operational-partition-loop-state-cost-bound-by-child-edges*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
and *P-subset*: *P* ⊆ *V*
shows ∃ *child-costs child-sets*.
length child-costs = *length child-sets* ∧
c ≤ *sum-list child-costs* + *M* ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
M ∗ *length child-costs*)
⟨*proof*⟩

theorem *full-operational-partition-loop-state-trace-and-edge-cost*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
obtains *child-costs child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
length child-costs = *length child-sets*
c ≤ *sum-list child-costs* + *M* ∗ *length child-costs* +

$t * (\mathit{sum\text{-}list}\ (\mathit{map}\ (\lambda X.\ \mathit{card}\ (\mathit{outgoing\text{-}edges}\ X))\ \mathit{child\text{-}sets}) +$
$M * \mathit{length}\ \mathit{child\text{-}costs})$
⟨*proof*⟩

theorem *full-operational-partition-loop-state-cost-bound-by-child-edges-complete*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow \mathit{reachable}\ s\ x$
shows $\exists$ *child-costs child-sets*.
$\mathit{length}\ \mathit{child\text{-}costs} = \mathit{length}\ \mathit{child\text{-}sets} \wedge$
$(\forall X{\in}\mathit{set}\ \mathit{child\text{-}sets}.\ X \subseteq V) \wedge$
$c \le \mathit{sum\text{-}list}\ \mathit{child\text{-}costs} + M * \mathit{length}\ \mathit{child\text{-}costs} +$
$t * (\mathit{sum\text{-}list}\ (\mathit{map}\ (\lambda X.\ \mathit{card}\ (\mathit{outgoing\text{-}edges}\ X))\ \mathit{child\text{-}sets}) +$
$M * \mathit{length}\ \mathit{child\text{-}costs})$
⟨*proof*⟩

theorem *full-operational-partition-loop-state-trace-and-complete-edge-cost*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow \mathit{reachable}\ s\ x$
obtains *child-costs child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
$\mathit{length}\ \mathit{child\text{-}costs} = \mathit{length}\ \mathit{child\text{-}sets}$
$\bigwedge X.\ X \in \mathit{set}\ \mathit{child\text{-}sets} \Longrightarrow X \subseteq V$
$c \le \mathit{sum\text{-}list}\ \mathit{child\text{-}costs} + M * \mathit{length}\ \mathit{child\text{-}costs} +$
$t * (\mathit{sum\text{-}list}\ (\mathit{map}\ (\lambda X.\ \mathit{card}\ (\mathit{outgoing\text{-}edges}\ X))\ \mathit{child\text{-}sets}) +$
$M * \mathit{length}\ \mathit{child\text{-}costs})$
⟨*proof*⟩

end

end
theory *BMSSP-Complexity*
imports *HOL−Library.Landau-Symbols BMSSP-Operational-Pull*
begin

28 Core Cost Accounting

This theory starts the checked complexity side of the development. Earlier theories prove that the recursive BMSSP algorithm returns the right vertices and labels. Here the same traces are interpreted as cost recurrences. The first part is deliberately arithmetic: logarithmic target functions, level widths, level capacities, and the Big-O envelopes that eventually become the SSSP headline.

The file then moves inside the graph locale and introduces the graph-size quantities used by the cost bounds. Vertex counts, edge counts, outgoing ranges, and range-tree chains are all finite combinatorial objects. The proofs establish that the lists produced by a partition loop are disjoint slices, so their vertex and outgoing-edge costs can be summed without double counting.

The final part defines a costed operational relation for the abstract partition interface. It is intentionally not the bucketed implementation; it is the reusable recurrence layer that later theories refine. The bucketed proof plugs concrete local costs into these abstract cost expressions, while the top-level bounds instantiate the logarithmic parameter schedule.

definition *sssp-log-factor* **where**
sssp-log-factor n = (*ln* (*real n* + *2*)) *powr* (*2* / *3*)

definition *sssp-log-factor-one-third* **where**
sssp-log-factor-one-third n = (*ln* (*real n* + *2*)) *powr* (*1* / *3*)

definition *sssp-time-target* **where**
sssp-time-target m n = *real* (*m n*) ∗ *sssp-log-factor n*

definition *bmssp-level-width* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bmssp-level-width t l = *2* ^ (*l* ∗ *t*)

definition *bmssp-level-cap* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
bmssp-level-cap k t l = *k* ∗ *bmssp-level-width t l*

lemma *bmssp-level-width-pos* [*simp*]:
0 < *bmssp-level-width t l*
⟨*proof*⟩

lemma *bmssp-level-cap-eq*:
bmssp-level-cap k t l = *k* ∗ *2* ^ (*l* ∗ *t*)
⟨*proof*⟩

lemma *bmssp-level-width-mono*:
assumes *l* ≤ *l′*
shows *bmssp-level-width t l* ≤ *bmssp-level-width t l′*
⟨*proof*⟩

lemma *bmssp-level-width-Suc*:
bmssp-level-width t (*Suc l*) = *bmssp-level-width t l* ∗ *2* ^ *t*
⟨*proof*⟩

lemma *bmssp-level-width-prev-times-two-le*:
assumes *0* < *t 0* < *l*
shows *2* ∗ *bmssp-level-width t* (*l* − *1*) ≤ *bmssp-level-width t l*
⟨*proof*⟩

lemma *bmssp-level-cap-mono*:

assumes $l \leq l'$
shows *bmssp-level-cap k t l* $\leq$ *bmssp-level-cap k t l'*
⟨*proof*⟩

lemma *bmssp-level-cap-ge-level*:
$k \leq$ *bmssp-level-cap k t l*
⟨*proof*⟩

lemma *bmssp-level-size-constraint-arithmetic*:
assumes *t-pos*: $0 < t$
and *l-pos*: $0 < l$
and *before*: *acc* $<$ $k *$ *bmssp-level-width t l*
and *child*: *child* $\leq 4 * k *$ *bmssp-level-width t* $(l - 1)$
and *extra*: *extra* $\leq k *$ *bmssp-level-width t l*
and *U-size*: *U-size* $\leq$ *acc* + *child* + *extra*
shows *U-size* $\leq 4 * k *$ *bmssp-level-width t l*
⟨*proof*⟩

lemma *sssp-log-factor-pos* [*simp*]:
$0 <$ *sssp-log-factor n*
⟨*proof*⟩

lemma *sssp-log-factor-one-third-pos* [*simp*]:
$0 <$ *sssp-log-factor-one-third n*
⟨*proof*⟩

lemma *sssp-log-factor-mono*:
assumes $n \leq m$
shows *sssp-log-factor n* $\leq$ *sssp-log-factor m*
⟨*proof*⟩

lemma *sssp-log-factor-one-third-mono*:
assumes $n \leq m$
shows *sssp-log-factor-one-third n* $\leq$ *sssp-log-factor-one-third m*
⟨*proof*⟩

lemma *sssp-log-factor-square-arg-le*:
sssp-log-factor $(n * n) \leq 2 *$ *sssp-log-factor n*
⟨*proof*⟩

lemma *sssp-log-factor-one-third-square-arg-le*:
sssp-log-factor-one-third $(n * n) \leq$
$2 *$ *sssp-log-factor-one-third n*
⟨*proof*⟩

lemma *sssp-log-factor-one-third-square*:
sssp-log-factor-one-third n $*$ *sssp-log-factor-one-third n* $=$
sssp-log-factor n
⟨*proof*⟩

lemma *eventually-one-le-sssp-log-factor-one-third*:
eventually (λ*n*. *1* ≤ *sssp-log-factor-one-third n*) *at-top*
⟨*proof*⟩

lemma *sssp-time-target-nonneg* [*simp*]:
0 ≤ *sssp-time-target m n*
⟨*proof*⟩

theorem *sssp-time-target-bigoI-norm*:
assumes *C*: *0* < *C*
and *eventually-bound*:
eventually
(λ*n*. *norm* (*real* (*T n*)) ≤ *C* ∗ *norm* (*sssp-time-target m n*)) *at-top*
shows (λ*n*. *real* (*T n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
⟨*proof*⟩

theorem *sssp-time-target-bigoI*:
assumes *C*: *0* < *C*
and *eventually-bound*:
eventually (λ*n*. *real* (*T n*) ≤ *C* ∗ *sssp-time-target m n*) *at-top*
shows (λ*n*. *real* (*T n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
⟨*proof*⟩

definition *bmssp-graph-time-bound* **where**
bmssp-graph-time-bound A R l t m n =
2 ∗ *A n* ∗ (*2* ∗ *l n* + *1*) ∗ *n* + (*R n* + *l n* ∗ *t n*) ∗ *m n*

theorem *bmssp-graph-time-bound-bigo-sssp-time-target*:
assumes *Cn-pos*: *0* < *Cn*
and *Cv-pos*: *0* < *Cv*
and *Ce-pos*: *0* < *Ce*
and *vertex-count-dominated*:
eventually (λ*n*. *real n* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *vertex-factor*:
eventually
(λ*n*. *real* (*2* ∗ *A n* ∗ (*2* ∗ *l n* + *1*)) ≤
Cv ∗ *sssp-log-factor n*) *at-top*
and *edge-factor*:
eventually
(λ*n*. *real* (*R n* + *l n* ∗ *t n*) ≤
Ce ∗ *sssp-log-factor n*) *at-top*
shows (λ*n*. *real* (*bmssp-graph-time-bound A R l t m n*))
∈ *O*(λ*n*. *sssp-time-target m n*)
⟨*proof*⟩

theorem *bmssp-cost-bound-bigo-sssp-time-target*:
assumes *Cn-pos*: *0* < *Cn*
and *Cv-pos*: *0* < *Cv*

and *Ce-pos*: $0 < Ce$
and *vertex-count-dominated*:
$\forall^{\text{at-top}} n.\ \text{real}\ n \leq Cn * \text{real}\ (m\ n)$
and *vertex-factor*:
$\forall^{\text{at-top}} n.\ \text{real}\ (2 * A\ n * (2 * l\ n + 1)) \leq Cv * \text{sssp-log-factor}\ n$
and *edge-factor*:
$\forall^{\text{at-top}} n.\ \text{real}\ (R\ n + l\ n * t\ n) \leq Ce * \text{sssp-log-factor}\ n$
and *cost-bound*:
$\forall^{\text{at-top}} n.\ T\ n \leq \text{bmssp-graph-time-bound}\ A\ R\ l\ t\ m\ n$
shows $(\lambda n.\ \text{real}\ (T\ n)) \in O(\lambda n.\ \text{sssp-time-target}\ m\ n)$
⟨*proof*⟩

definition *bmssp-refined-graph-time-bound* **where**
bmssp-refined-graph-time-bound A R H l t m n =
$2 * A\ n * (2 * l\ n + 1) * n +$
$(R\ n + t\ n + l\ n * H\ n) * m\ n$

theorem *bmssp-refined-graph-time-bound-bigo-sssp-time-target*:
assumes *Cn-pos*: $0 < Cn$
and *Cv-pos*: $0 < Cv$
and *Ce-pos*: $0 < Ce$
and *vertex-count-dominated*:
$\forall^{\text{at-top}} n.\ \text{real}\ n \leq Cn * \text{real}\ (m\ n)$
and *vertex-factor*:
$\forall^{\text{at-top}} n.\ \text{real}\ (2 * A\ n * (2 * l\ n + 1)) \leq Cv * \text{sssp-log-factor}\ n$
and *edge-factor*:
$\forall^{\text{at-top}} n.\ \text{real}\ (R\ n + t\ n + l\ n * H\ n) \leq Ce * \text{sssp-log-factor}\ n$
shows $(\lambda n.\ \text{real}\ (\text{bmssp-refined-graph-time-bound}\ A\ R\ H\ l\ t\ m\ n))$
$\in O(\lambda n.\ \text{sssp-time-target}\ m\ n)$
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target*:
assumes *Cn-pos*: $0 < Cn$
and *Cv-pos*: $0 < Cv$
and *Ce-pos*: $0 < Ce$
and *vertex-count-dominated*:
$\forall^{\text{at-top}} n.\ \text{real}\ n \leq Cn * \text{real}\ (m\ n)$
and *vertex-factor*:
$\forall^{\text{at-top}} n.\ \text{real}\ (2 * A\ n * (2 * l\ n + 1)) \leq Cv * \text{sssp-log-factor}\ n$

and *edge-factor*:
eventually
$(\lambda n.\ real\ (R\ n + t\ n + l\ n * H\ n) \leq$
$Ce * sssp\text{-}log\text{-}factor\ n)$ *at-top*
and *cost-bound*:
eventually
$(\lambda n.\ T\ n \leq bmssp\text{-}refined\text{-}graph\text{-}time\text{-}bound\ A\ R\ H\ l\ t\ m\ n)$ *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds*:
assumes *Cn-pos*: $0 < Cn$
and *Ca-pos*: $0 < Ca$
and *Cl-nonneg*: $0 \leq Cl$
and *Cr-pos*: $0 < Cr$
and *Ct-pos*: $0 < Ct$
and *Ch-pos*: $0 < Ch$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \leq Cn * real\ (m\ n))$ *at-top*
and *A-bound*:
eventually
$(\lambda n.\ real\ (A\ n) \leq Ca * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$ *at-top*
and *l-bound*:
eventually
$(\lambda n.\ real\ (l\ n) \leq Cl * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$ *at-top*
and *R-bound*:
eventually
$(\lambda n.\ real\ (R\ n) \leq Cr * sssp\text{-}log\text{-}factor\ n)$ *at-top*
and *t-bound*:
eventually
$(\lambda n.\ real\ (t\ n) \leq Ct * sssp\text{-}log\text{-}factor\ n)$ *at-top*
and *H-bound*:
eventually
$(\lambda n.\ real\ (H\ n) \leq Ch * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$ *at-top*
and *cost-bound*:
eventually
$(\lambda n.\ T\ n \leq bmssp\text{-}refined\text{-}graph\text{-}time\text{-}bound\ A\ R\ H\ l\ t\ m\ n)$ *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *bmssp-refined-graph-time-bound-bigo-sssp-time-target-slack*:
assumes *Cn-pos*: $0 < Cn$
and *Cm-pos*: $0 < Cm$
and *Cv-pos*: $0 < Cv$
and *Ce-pos*: $0 < Ce$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ (v\ n) \leq Cn * real\ (m\ n))$ *at-top*
and *edge-count-dominated*:
eventually $(\lambda n.\ real\ (m'\ n) \leq Cm * real\ (m\ n))$ *at-top*

and *vertex-factor*:
eventually
$(\lambda n.\ real\ (2 * A\ n * (2 * l\ n + 1)) \leq$
$Cv * sssp\text{-}log\text{-}factor\ n)\ at\text{-}top$
and *edge-factor*:
eventually
$(\lambda n.\ real\ (R\ n + t\ n + l\ n * H\ n) \leq$
$Ce * sssp\text{-}log\text{-}factor\ n)\ at\text{-}top$
shows $(\lambda n.\ real\ (bmssp\text{-}refined\text{-}graph\text{-}time\text{-}bound$
$(\lambda\text{-}.\ A\ n)\ (\lambda\text{-}.\ R\ n)\ (\lambda\text{-}.\ H\ n)\ (\lambda\text{-}.\ l\ n)\ (\lambda\text{-}.\ t\ n)$
$(\lambda\text{-}.\ m'\ n)\ (v\ n))))$
$\in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds-slack*:
assumes *Cn-pos*: $0 < Cn$
and *Cm-pos*: $0 < Cm$
and *Ca-pos*: $0 < Ca$
and *Cl-nonneg*: $0 \leq Cl$
and *Cr-pos*: $0 < Cr$
and *Ct-pos*: $0 < Ct$
and *Ch-pos*: $0 < Ch$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ (v\ n) \leq Cn * real\ (m\ n))\ at\text{-}top$
and *edge-count-dominated*:
eventually $(\lambda n.\ real\ (m'\ n) \leq Cm * real\ (m\ n))\ at\text{-}top$
and *A-bound*:
eventually
$(\lambda n.\ real\ (A\ n) \leq Ca * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)\ at\text{-}top$
and *l-bound*:
eventually
$(\lambda n.\ real\ (l\ n) \leq Cl * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)\ at\text{-}top$
and *R-bound*:
eventually
$(\lambda n.\ real\ (R\ n) \leq Cr * sssp\text{-}log\text{-}factor\ n)\ at\text{-}top$
and *t-bound*:
eventually
$(\lambda n.\ real\ (t\ n) \leq Ct * sssp\text{-}log\text{-}factor\ n)\ at\text{-}top$
and *H-bound*:
eventually
$(\lambda n.\ real\ (H\ n) \leq Ch * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)\ at\text{-}top$
and *cost-bound*:
eventually
$(\lambda n.\ T\ n \leq bmssp\text{-}refined\text{-}graph\text{-}time\text{-}bound$
$(\lambda\text{-}.\ A\ n)\ (\lambda\text{-}.\ R\ n)\ (\lambda\text{-}.\ H\ n)\ (\lambda\text{-}.\ l\ n)\ (\lambda\text{-}.\ t\ n)$
$(\lambda\text{-}.\ m'\ n)\ (v\ n))\ at\text{-}top$
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

29 Logarithmic Parameter Schedule

The paper's schedule balances two logarithmic scales. The parameter usually called k is chosen on the order of $log^{\frown}(1/3)\ n$, while the recursion depth and partition width use the corresponding $log^{\frown}(2/3)\ n$ scale. The two definitions below use ceilings of the real-valued factors so that the executable and inductive parameters remain natural numbers.

The lemmas in this section are mostly analytic bookkeeping. They show that the ceiling parameters are eventually positive, monotone enough for the recurrence, and asymptotically within constant factors of the real logarithmic targets. Those facts are what let the refined graph-time expressions be rewritten as $O(m * log^{\frown}(2/3)\ n)$.

definition *sssp-log-one-third-param* **where**
sssp-log-one-third-param n = nat ⌈sssp-log-factor-one-third n⌉

definition *sssp-log-two-thirds-param* **where**
sssp-log-two-thirds-param n = nat ⌈sssp-log-factor n⌉

lemma *sssp-log-one-third-param-pos* [*simp*]:
0 < sssp-log-one-third-param n
⟨*proof*⟩

lemma *sssp-log-two-thirds-param-pos* [*simp*]:
0 < sssp-log-two-thirds-param n
⟨*proof*⟩

lemma *real-cuberoot-at-top*:
filterlim (λ*x*::*real*. *root 3 x*) *at-top at-top*
⟨*proof*⟩

lemma *sssp-log-factor-one-third-at-top*:
filterlim sssp-log-factor-one-third at-top at-top
⟨*proof*⟩

lemma *nat-ceiling-at-top*:
filterlim (λ*x*::*real*. *nat* ⌈*x*⌉) *at-top at-top*
⟨*proof*⟩

lemma *sssp-log-one-third-param-at-top*:
filterlim sssp-log-one-third-param at-top at-top
⟨*proof*⟩

lemma *eventually-less-sssp-log-one-third-param*:
eventually (λ*n*. *K < sssp-log-one-third-param n*) *at-top*
⟨*proof*⟩

lemma *sssp-log-two-thirds-param-le-one-third-square*:
sssp-log-two-thirds-param n ≤

sssp-log-one-third-param n ∗ sssp-log-one-third-param n
⟨*proof*⟩

lemma *real-nat-ceiling-le-twice*:
fixes *x* :: *real*
assumes *1* ≤ *x*
shows *real* (*nat* ⌈*x*⌉) ≤ *2* ∗ *x*
⟨*proof*⟩

lemma *eventually-one-le-sssp-log-factor*:
eventually (λ*n*. *1* ≤ *sssp-log-factor n*) *at-top*
⟨*proof*⟩

lemma *sssp-log-one-third-param-component-bound*:
eventually
(λ*n*. *real* (*sssp-log-one-third-param n*) ≤
2 ∗ *sssp-log-factor-one-third n*) *at-top*
⟨*proof*⟩

lemma *sssp-log-two-thirds-param-component-bound*:
eventually
(λ*n*. *real* (*sssp-log-two-thirds-param n*) ≤
2 ∗ *sssp-log-factor n*) *at-top*
⟨*proof*⟩

lemma *sssp-log-one-third-param-square-arg-component-bound*:
eventually
(λ*n*. *real* (*sssp-log-one-third-param* (*n* ∗ *n*)) ≤
4 ∗ *sssp-log-factor-one-third n*) *at-top*
⟨*proof*⟩

lemma *sssp-log-two-thirds-param-square-arg-component-bound*:
eventually
(λ*n*. *real* (*sssp-log-two-thirds-param* (*n* ∗ *n*)) ≤
4 ∗ *sssp-log-factor n*) *at-top*
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params*:
assumes *Cn-pos*: *0* < *Cn*
and *vertex-count-dominated*:
eventually (λ*n*. *real n* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *cost-bound*:
eventually
(λ*n*. *T n* ≤
bmssp-refined-graph-time-bound sssp-log-one-third-param
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param m n) *at-top*
shows (λ*n*. *real* (*T n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree*:
fixes D :: *nat*
assumes *Cn-pos*: $0 < Cn$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \leq Cn * real\ (m\ n))$ *at-top*
and *cost-bound*:
eventually
$(\lambda n.\ T\ n \leq$
bmssp-refined-graph-time-bound
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param m n) *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-slack*:
fixes D :: *nat*
assumes *Cn-pos*: $0 < Cn$
and *Cm-pos*: $0 < Cm$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ (v\ n) \leq Cn * real\ (m\ n))$ *at-top*
and *edge-count-dominated*:
eventually $(\lambda n.\ real\ (m'\ n) \leq Cm * real\ (m\ n))$ *at-top*
and *cost-bound*:
eventually
$(\lambda n.\ T\ n \leq$
bmssp-refined-graph-time-bound
$(\lambda\text{-}.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
$(\lambda\text{-}.\ sssp\text{-}log\text{-}two\text{-}thirds\text{-}param\ n)$
$(\lambda\text{-}.\ sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
$(\lambda\text{-}.\ sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
$(\lambda\text{-}.\ sssp\text{-}log\text{-}two\text{-}thirds\text{-}param\ n)$
$(\lambda\text{-}.\ m'\ n)\ (v\ n))$ *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg*:
fixes D :: *nat*
assumes *Cn-pos*: $0 < Cn$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \leq Cn * real\ (m\ n))$ *at-top*
and *cost-bound*:
eventually
$(\lambda n.\ T\ n \leq$
bmssp-refined-graph-time-bound
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ (n * n))$

(λn. sssp-log-two-thirds-param (n * n))
(λn. sssp-log-one-third-param (n * n))
(λn. sssp-log-one-third-param (n * n))
(λn. sssp-log-two-thirds-param (n * n)) m n) at-top
shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
⟨proof⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg-slack*:
fixes D :: nat
assumes Cn-pos: 0 < Cn
and Cm-pos: 0 < Cm
and vertex-count-dominated:
eventually (λn. real (v n) ≤ Cn * real (m n)) at-top
and edge-count-dominated:
eventually (λn. real (m' n) ≤ Cm * real (m n)) at-top
and cost-bound:
eventually
(λn. T n ≤
bmssp-refined-graph-time-bound
(λ-. Suc D * sssp-log-one-third-param (n * n))
(λ-. sssp-log-two-thirds-param (n * n))
(λ-. sssp-log-one-third-param (n * n))
(λ-. sssp-log-one-third-param (n * n))
(λ-. sssp-log-two-thirds-param (n * n))
(λ-. m' n) (v n)) at-top
shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
⟨proof⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg-real-slack*:
fixes D :: nat
assumes Cn-pos: 0 < Cn
and Ccost-pos: 0 < Ccost
and vertex-count-dominated:
eventually (λn. real n ≤ Cn * real (m n)) at-top
and cost-bound:
eventually
(λn. real (T n) ≤ Ccost *
real (bmssp-refined-graph-time-bound
(λn. Suc D * sssp-log-one-third-param (n * n))
(λn. sssp-log-two-thirds-param (n * n))
(λn. sssp-log-one-third-param (n * n))
(λn. sssp-log-one-third-param (n * n))
(λn. sssp-log-two-thirds-param (n * n)) m n)) at-top
shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
⟨proof⟩

theorem *bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-*

real-slack:
fixes $D :: nat$
assumes *Cn-pos*: $0 < Cn$
and *Ccost-pos*: $0 < Ccost$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \leq Cn * real\ (m\ n))$ *at-top*
and *cost-bound*:
eventually
$(\lambda n.\ real\ (T\ n) \leq Ccost *$
real (*bmssp-refined-graph-time-bound*
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param $m\ n))$ *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

context *unique-shortest-digraph*
begin

From this point on the arithmetic bounds are tied to a concrete finite graph. The definitions *vertex-count* and *edge-count* are simple wrappers around the locale's finite vertex and edge sets, but naming them keeps the final theorems independent of internal set notation. The parent-edge construction is used to relate reachable vertices to edges, which is the standard sparse graph move behind replacing a vertex term by an edge term when all vertices are reachable.

definition *vertex-count* **where**
vertex-count = *card V*

definition *edge-count* **where**
edge-count = *card E*

definition *parent-edge-to* **where**
parent-edge-to v = (*last* (*butlast* (*shortest-path-to v*)), *v*)

lemma *vertex-count-pos* [*simp*]:
$0 <$ *vertex-count*
⟨*proof*⟩

lemma *edge-count-le-vertex-count-square*:
edge-count ≤ *vertex-count* * *vertex-count*
⟨*proof*⟩

lemma *parent-edge-to-in-E*:
assumes *reach-v*: *reachable s v*
and *v-ne*: $v \neq s$
shows *parent-edge-to v* ∈ *E*
⟨*proof*⟩

theorem *vertex-count-le-Suc-edge-count-if-all-reachable*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
shows *vertex-count* $\leq$ *Suc edge-count*
⟨*proof*⟩

corollary *vertex-count-le-twice-edge-count-if-all-reachable*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *nontrivial*: *0* $<$ *edge-count*
shows *vertex-count* $\leq$ *2* $*$ *edge-count*
⟨*proof*⟩

lemma *card-subset-vertex-count*:
assumes $S \subseteq V$
shows *card S* $\leq$ *vertex-count*
⟨*proof*⟩

lemma *eventually-top-level-threshold-exceeds-vertex-count*:
eventually
(λ*N*. *vertex-count* $<$
sssp-log-one-third-param N $*$
bmssp-level-cap (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
at-top
⟨*proof*⟩

lemma *find-pivots-pivots-capped-singleton-empty-if-params-exceed-vertex-count*:
assumes *k-gt*: *vertex-count* $<$ *k*
and *cap-ge*: *vertex-count* $\leq$ *cap*
shows *find-pivots-pivots-capped k cap d* {*s*} *B* $=$ {}
⟨*proof*⟩

lemma *eventually-top-level-pivots-empty*:
eventually
(λ*N*. *find-pivots-pivots-capped* (*sssp-log-one-third-param N*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
d {*s*} *B* $=$ {})
at-top
⟨*proof*⟩

lemma *edge-count-outgoing-bound*:
card (*outgoing-edges X*) $\leq$ *edge-count*
⟨*proof*⟩

lemma *edge-count-outgoing-bound-tree*:
card (*outgoing-edges* (*bound-tree S B*)) $\leq$ *edge-count*
⟨*proof*⟩

The range-based edge definitions are the combinatorial support for the direct-insert and lower-batch split. An edge is charged to a range according

to the relaxed value at its source, not merely according to the source vertex. This matches the algorithm: a completed child range may generate relaxations that fall below the pull bound, inside the parent output bound, or outside the current problem.

The length lemmas connect executable lists of relaxation pairs to the mathematical edge ranges. The disjointness and split lemmas that follow make it possible to sum per-child edge charges and bound them by the outgoing edges of the assembled parent range.

definition *outgoing-edges-range* **where**
outgoing-edges-range U a B =
{(*u*, *v*) ∈ *E*. *u* ∈ *U* ∧ *a* ≤ *dist s u* + *w u v* ∧
below-bound (*dist s u* + *w u v*) *B*}

lemma *outgoing-edges-range-subset-E*:
outgoing-edges-range U a B ⊆ *E*
⟨*proof*⟩

lemma *outgoing-edges-range-subset-outgoing-edges*:
outgoing-edges-range U a B ⊆ *outgoing-edges U*
⟨*proof*⟩

lemma *finite-outgoing-edges-range* [*simp*]:
finite (*outgoing-edges-range U a B*)
⟨*proof*⟩

lemma *card-outgoing-edges-range-le-edge-count*:
card (*outgoing-edges-range U a B*) ≤ *edge-count*
⟨*proof*⟩

definition *edge-relaxation-pairs-in-bound* **where**
edge-relaxation-pairs-in-bound d U L B =
map (λ(*u*, *v*). (*v*, *d u* + *w u v*))
(*filter* (λ(*u*, *v*). *L* ≤ *d u* + *w u v* ∧
below-bound (*d u* + *w u v*) *B*)
(*edge-list-of* (*outgoing-edges U*)))

lemma *edge-relaxation-pairs-in-bound-value-ge-low*:
assumes (*x*, *b*) ∈ *set* (*edge-relaxation-pairs-in-bound d U L B*)
shows *L* ≤ *b*
⟨*proof*⟩

lemma *edge-relaxation-pairs-in-bound-value-below-bound*:
assumes (*x*, *b*) ∈ *set* (*edge-relaxation-pairs-in-bound d U L B*)
shows *below-bound b B*
⟨*proof*⟩

lemma *edge-relaxation-pairs-in-bound-length-le-outgoing-edges-range*:
assumes *complete*: *complete-on d U*

and *reaches*: $\bigwedge u.\ u \in U \Longrightarrow$ *reachable* $s\ u$
shows *length* (*edge-relaxation-pairs-in-bound* $d\ U\ L\ B$) $\leq$
card (*outgoing-edges-range* $U\ L\ B$)
⟨*proof*⟩

lemma *partition-batch-cost-bound-edge-relaxation-pairs-in-bound*:
assumes *cost*:
partition-batch-cost-bound $c\ t$
(*edge-relaxation-pairs-in-bound* $d\ U\ L\ B$)
and *complete*: *complete-on* $d\ U$
and *reaches*: $\bigwedge u.\ u \in U \Longrightarrow$ *reachable* $s\ u$
shows $c \leq t * card$ (*outgoing-edges-range* $U\ L\ B$)
⟨*proof*⟩

lemma *edge-relaxation-pairs-between-length-le-outgoing-edges-range*:
assumes *complete*: *complete-on* $d\ U$
and *reaches*: $\bigwedge u.\ u \in U \Longrightarrow$ *reachable* $s\ u$
shows *length* (*edge-relaxation-pairs-between* $d\ U\ L\ H$) $\leq$
card (*outgoing-edges-range* $U\ L$ (*Fin* H))
⟨*proof*⟩

lemma *edge-relaxation-pairs-between-length-le-range-tree-outgoing-edges-range*:
assumes *complete*: *complete-on* d (*range-tree* $S\ a\ B$)
shows *length* (*edge-relaxation-pairs-between* d (*range-tree* $S\ a\ B$) $L\ H$) $\leq$
card (*outgoing-edges-range* (*range-tree* $S\ a\ B$) L (*Fin* H))
⟨*proof*⟩

lemma *outgoing-edges-range-mono-sources*:
assumes $U \subseteq W$
shows *outgoing-edges-range* $U\ a\ B \subseteq$ *outgoing-edges-range* $W\ a\ B$
⟨*proof*⟩

lemma *outgoing-edges-range-split*:
assumes *a-le-b*: $a \leq b$
and *b-le-B*: *bound-le* (*Fin* b) B
shows *outgoing-edges-range* $U\ a\ B =$
outgoing-edges-range $U\ a$ (*Fin* b) $\cup$ *outgoing-edges-range* $U\ b\ B$
⟨*proof*⟩

lemma *outgoing-edges-range-split-disjoint*:
outgoing-edges-range $U\ a$ (*Fin* b) $\cap$ *outgoing-edges-range* $U\ b\ B = \{\}$
⟨*proof*⟩

lemma *card-outgoing-edges-range-split*:
assumes *a-le-b*: $a \leq b$
and *b-le-B*: *bound-le* (*Fin* b) B
shows *card* (*outgoing-edges-range* $U\ a\ B$) =
card (*outgoing-edges-range* $U\ a$ (*Fin* b)) +
card (*outgoing-edges-range* $U\ b\ B$)

⟨*proof*⟩

lemma *card-outgoing-edges-range-lower-direct-split*:
assumes *b-le-beta*: $b \leq beta$
and *beta-le-B*: *bound-le* (*Fin beta*) *B*
shows *card* (*outgoing-edges-range U b* (*Fin beta*)) +
card (*outgoing-edges-range U beta B*) =
card (*outgoing-edges-range U b B*)
⟨*proof*⟩

lemma *card-outgoing-edges-range-zero-lower-direct-split-le*:
assumes *beta-le-B*: *bound-le* (*Fin beta*) *B*
and *nonneg*:
$\bigwedge u\ v.\ [\![(u, v) \in E;\ u \in U]\!] \Longrightarrow 0 \leq dist\ s\ u + w\ u\ v$
shows *card* (*outgoing-edges-range U 0* (*Fin beta*)) +
card (*outgoing-edges-range U beta B*) ≤
card (*outgoing-edges-range U 0 B*)
⟨*proof*⟩

lemma *outgoing-edges-range-disjoint-sources*:
assumes $U \cap W = \{\}$
shows *outgoing-edges-range U a B* ∩ *outgoing-edges-range W c D* = {}
⟨*proof*⟩

fun *pairwise-disjoint-list* **where**
pairwise-disjoint-list [] ⟷ *True*
| *pairwise-disjoint-list* (*X* # *Xs*) ⟷
(∀ *Y*∈*set Xs*. *X* ∩ *Y* = {}) ∧ *pairwise-disjoint-list Xs*

lemma *pairwise-disjoint-list-outgoing-edges-range*:
assumes *pairwise-disjoint-list Us*
shows *pairwise-disjoint-list* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*)
⟨*proof*⟩

lemma *edge-outdegree-le-edge-count*:
edge-outdegree-le edge-count
⟨*proof*⟩

lemma *card-Union-pairwise-disjoint-list*:
assumes *disj*: *pairwise-disjoint-list Xs*
and *finite-sets*: $\bigwedge X.\ X \in set\ Xs \Longrightarrow finite\ X$
shows *card* (⋃(*set Xs*)) = *sum-list* (*map card Xs*)
⟨*proof*⟩

lemma *sum-card-outgoing-edges-range-disjoint-sources-le-edge-count*:
assumes *disj*: *pairwise-disjoint-list Us*
shows *sum-list* (*map* (λ*U*. *card* (*outgoing-edges-range U a B*)) *Us*) ≤ *edge-count*
⟨*proof*⟩

lemma *outgoing-edges-disjoint-if-sources-disjoint*:
assumes $A \cap B = \{\}$
shows *outgoing-edges* $A \cap$ *outgoing-edges* $B = \{\}$
⟨*proof*⟩

lemma *pairwise-disjoint-list-outgoing-edges*:
assumes *pairwise-disjoint-list Xs*
shows *pairwise-disjoint-list* (*map outgoing-edges Xs*)
⟨*proof*⟩

lemma *outgoing-edges-Union-list*:
outgoing-edges ($\bigcup$(*set Xs*)) = $\bigcup$(*set* (*map outgoing-edges Xs*))
⟨*proof*⟩

lemma *outgoing-edges-mono*:
assumes $A \subseteq B$
shows *outgoing-edges* $A \subseteq$ *outgoing-edges* B
⟨*proof*⟩

fun *range-tree-child-list* **where**
range-tree-child-list S a [] = []
| *range-tree-child-list S a* (*b* # *bs*) =
range-tree S a (*Fin b*) # *range-tree-child-list S b bs*

lemma *length-range-tree-child-list* [*simp*]:
length (*range-tree-child-list S a bs*) = *length bs*
⟨*proof*⟩

lemma *range-tree-child-list-set-subset-chain-list*:
set (*range-tree-child-list S a bs*) ⊆ *set* (*range-tree-chain-list S a bs B*)
⟨*proof*⟩

lemma *Union-range-tree-child-list-subset-chain*:
$\bigcup$(*set* (*range-tree-child-list S a bs*)) ⊆ *range-tree-chain S a bs B*
⟨*proof*⟩

fun *range-tree-child-edge-range-list* **where**
range-tree-child-edge-range-list S a [] [] = []
| *range-tree-child-edge-range-list S a* [] (*b* # *bs*) = []
| *range-tree-child-edge-range-list S a* (*beta* # *betas*) [] = []
| *range-tree-child-edge-range-list S a* (*beta* # *betas*) (*b* # *bs*) =
outgoing-edges-range (*range-tree S a* (*Fin b*)) *b* (*Fin beta*) #
range-tree-child-edge-range-list S b betas bs

lemma *range-tree-child-edge-range-list-subset-outgoing-sources*:
assumes *Y* ∈ *set* (*range-tree-child-edge-range-list S a betas bs*)
shows ∃ *X*∈*set* (*range-tree-child-list S a bs*). *Y* ⊆ *outgoing-edges X*
⟨*proof*⟩

lemma *range-tree-child-edge-range-list-member-subset-E*:
  assumes $Y \in$ *set* (*range-tree-child-edge-range-list S a betas bs*)
  shows $Y \subseteq E$
⟨*proof*⟩

lemma *pairwise-disjoint-list-range-tree-child-edge-range-list*:
  assumes *mono*: *nondecreasing-from a bs*
  shows *pairwise-disjoint-list* (*range-tree-child-edge-range-list S a betas bs*)
  ⟨*proof*⟩

lemma *sum-card-range-tree-child-edge-range-list-le-edge-count*:
  assumes *mono*: *nondecreasing-from a bs*
  shows *sum-list* (*map card* (*range-tree-child-edge-range-list S a betas bs*))
    ≤ *edge-count*
⟨*proof*⟩

lemma *sum-card-range-tree-child-edge-range-list-le-outgoing-edges-chain*:
  assumes *mono*: *nondecreasing-from a bs*
  shows *sum-list* (*map card* (*range-tree-child-edge-range-list S a betas bs*))
    ≤ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

fun *range-tree-child-direct-edge-range-list* **where**
  range-tree-child-direct-edge-range-list S B a [] [] = []
| *range-tree-child-direct-edge-range-list S B a* [] (*b* # *bs*) = []
| *range-tree-child-direct-edge-range-list S B a* (*beta* # *betas*) [] = []
| *range-tree-child-direct-edge-range-list S B a* (*beta* # *betas*) (*b* # *bs*) =
    outgoing-edges-range (*range-tree S a* (*Fin b*)) *beta B* #
    range-tree-child-direct-edge-range-list S B b betas bs

fun *range-tree-child-zero-edge-range-list* **where**
  range-tree-child-zero-edge-range-list S a [] [] = []
| *range-tree-child-zero-edge-range-list S a* [] (*b* # *bs*) = []
| *range-tree-child-zero-edge-range-list S a* (*beta* # *betas*) [] = []
| *range-tree-child-zero-edge-range-list S a* (*beta* # *betas*) (*b* # *bs*) =
    outgoing-edges-range (*range-tree S a* (*Fin b*)) *0* (*Fin beta*) #
    range-tree-child-zero-edge-range-list S b betas bs

fun *range-tree-child-full-edge-range-list* **where**
  range-tree-child-full-edge-range-list S B a [] = []
| *range-tree-child-full-edge-range-list S B a* (*b* # *bs*) =
    outgoing-edges-range (*range-tree S a* (*Fin b*)) *0 B* #
    range-tree-child-full-edge-range-list S B b bs

fun *range-tree-child-bound-pair-list* **where**
  range-tree-child-bound-pair-list S a [] [] = []
| *range-tree-child-bound-pair-list S a* [] (*b* # *bs*) = []
| *range-tree-child-bound-pair-list S a* (*beta* # *betas*) [] = []
| *range-tree-child-bound-pair-list S a* (*beta* # *betas*) (*b* # *bs*) =

(*range-tree S a* (*Fin b*), *Fin beta*) #
range-tree-child-bound-pair-list S b betas bs

lemma *range-tree-child-direct-edge-range-list-subset-outgoing-sources*:
assumes $Y \in$ *set* (*range-tree-child-direct-edge-range-list S B a betas bs*)
shows $\exists X \in$*set* (*range-tree-child-list S a bs*). $Y \subseteq$ *outgoing-edges X*
⟨*proof*⟩

lemma *range-tree-child-direct-edge-range-list-member-subset-E*:
assumes $Y \in$ *set* (*range-tree-child-direct-edge-range-list S B a betas bs*)
shows $Y \subseteq E$
⟨*proof*⟩

lemma *pairwise-disjoint-list-range-tree-child-direct-edge-range-list*:
assumes *mono*: *nondecreasing-from a bs*
shows *pairwise-disjoint-list*
(*range-tree-child-direct-edge-range-list S B a betas bs*)
⟨*proof*⟩

lemma *sum-card-range-tree-child-direct-edge-range-list-le-edge-count*:
assumes *mono*: *nondecreasing-from a bs*
shows *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list S B a betas bs*))
≤ *edge-count*
⟨*proof*⟩

lemma *sum-card-range-tree-child-direct-edge-range-list-le-outgoing-edges-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list S B' a betas bs*))
≤ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

lemma *range-tree-nonneg-edge-value*:
assumes *edge*: $(u, v) \in E$
and *u-range*: $u \in$ *range-tree S a B*
shows $0 \leq$ *dist s u* + *w u v*
⟨*proof*⟩

lemma *sum-card-range-tree-child-zero-direct-edge-ranges-le-full*:
assumes *bounds*: *bounds-le B betas*
shows
sum-list (*map card* (*range-tree-child-zero-edge-range-list S a betas bs*)) +
sum-list
(*map card* (*range-tree-child-direct-edge-range-list S B a betas bs*))
≤ *sum-list* (*map card* (*range-tree-child-full-edge-range-list S B a bs*))
⟨*proof*⟩

lemma *range-tree-child-full-edge-range-list-eq-map*:

range-tree-child-full-edge-range-list S B a bs =
map (λ*U*. *outgoing-edges-range U 0 B*) (*range-tree-child-list S a bs*)
⟨*proof*⟩

lemma *sum-card-range-tree-child-full-edge-range-list-le-outgoing-edges-range-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows *sum-list* (*map card* (*range-tree-child-full-edge-range-list S B a bs*))
≤ *card* (*outgoing-edges-range* (*range-tree-chain S a bs C*) *0 B*)
⟨*proof*⟩

lemma *sum-card-range-tree-child-zero-direct-edge-ranges-le-outgoing-edges-range-chain*:
assumes *mono*: *nondecreasing-from a bs*
and *bounds*: *bounds-le B betas*
shows
sum-list (*map card* (*range-tree-child-zero-edge-range-list S a betas bs*)) +
sum-list
(*map card* (*range-tree-child-direct-edge-range-list S B a betas bs*))
≤ *card* (*outgoing-edges-range* (*range-tree-chain S a bs C*) *0 B*)
⟨*proof*⟩

lemma *weighted-sum-child-lower-direct-edge-ranges-le*:
assumes *mono*: *nondecreasing-from a bs*
shows
h ∗ *sum-list* (*map card* (*range-tree-child-edge-range-list S a betas bs*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list S B a betas bs*))
≤ (*h* + *t*) ∗ *edge-count*
⟨*proof*⟩

lemma *weighted-sum-child-lower-direct-edge-ranges-le-outgoing-edges-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows
h ∗ *sum-list* (*map card* (*range-tree-child-edge-range-list S a betas bs*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list S B′ a betas bs*))
≤ (*h* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

lemma *sum-card-outgoing-edges-le-degree*:
assumes *degree*: *edge-outdegree-le* Δ
and *sets*: ⋀*X*. *X* ∈ *set Xs* ⟹ *X* ⊆ *V*
shows *sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *Xs*) ≤
Δ ∗ *sum-list* (*map card Xs*)
⟨*proof*⟩

lemma *card-outgoing-edges-Union-list-eq-sum*:
assumes *disj*: *pairwise-disjoint-list Xs*
shows *card* (*outgoing-edges* (⋃(*set Xs*))) =

sum-list (*map* (λ*X*. *card* (*outgoing-edges X*)) *Xs*)
⟨*proof*⟩

lemma *range-tree-chain-list-pairwise-disjoint*:
assumes *mono*: *nondecreasing-from a bs*
shows *pairwise-disjoint-list* (*range-tree-chain-list S a bs B*)
⟨*proof*⟩

lemma *range-tree-child-list-pairwise-disjoint*:
assumes *mono*: *nondecreasing-from a bs*
shows *pairwise-disjoint-list* (*range-tree-child-list S a bs*)
⟨*proof*⟩

theorem *card-range-tree-child-list-le-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows *sum-list* (*map card* (*range-tree-child-list S a bs*)) ≤
card (*range-tree-chain S a bs B*)
⟨*proof*⟩

lemma *sum-card-list-all2-le*:
assumes *list-all2* (λ*X Y*. *card X* ≤ *card Y*) *Xs Ys*
shows *sum-list* (*map card Xs*) ≤ *sum-list* (*map card Ys*)
⟨*proof*⟩

lemma *sum-card-dominated-by-disjoint-union*:
assumes *disj*: *pairwise-disjoint-list Ys*
and *finite-sets*: ⋀*Y*. *Y* ∈ *set Ys* ⟹ *finite Y*
and *dominated*: *list-all2* (λ*X Y*. *card X* ≤ *card Y*) *Xs Ys*
shows *sum-list* (*map card Xs*) ≤ *card* (⋃(*set Ys*))
⟨*proof*⟩

theorem *sum-card-dominated-by-range-tree-child-list-le-chain*:
assumes *mono*: *nondecreasing-from a bs*
and *dominated*:
list-all2 (λ*X R*. *card X* ≤ *card R*) *Xs*
(*range-tree-child-list S a bs*)
shows *sum-list* (*map card Xs*) ≤ *card* (*range-tree-chain S a bs B*)
⟨*proof*⟩

theorem *card-outgoing-edges-range-tree-chain-eq-sum-list*:
assumes *mono*: *nondecreasing-from a bs*
shows *card* (*outgoing-edges* (*range-tree-chain S a bs B*)) =
sum-list
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-chain-list S a bs B*))
⟨*proof*⟩

theorem *card-outgoing-edges-range-tree-child-list-le-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows *sum-list*

(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list S a bs*)) ≤
card (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

lemma *sum-outgoing-card-list-all2-le*:
assumes *list-all2*
(λ*X Y*. *card* (*outgoing-edges X*) ≤ *card* (*outgoing-edges Y*)) *Xs Ys*
shows *sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *Xs*) ≤
sum-list (*map* (λ*Y*. *card* (*outgoing-edges Y*)) *Ys*)
⟨*proof*⟩

theorem *sum-outgoing-dominated-by-range-tree-child-list-le-chain*:
assumes *mono*: *nondecreasing-from a bs*
and *dominated*:
list-all2
(λ*X R*. *card* (*outgoing-edges X*) ≤ *card* (*outgoing-edges R*)) *Xs*
(*range-tree-child-list S a bs*)
shows *sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *Xs*) ≤
card (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

lemma *edge-costs-le-range-tree-chain-outgoing*:
assumes *mono*: *nondecreasing-from a bs*
and *costs*: *list-all2* (λ*c X*. *c* ≤ *C* ∗ *card* (*outgoing-edges X*)) *costs*
(*range-tree-chain-list S a bs B*)
shows *sum-list costs* ≤ *C* ∗ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

These two cost shapes are the local language of the runtime proof. The level bound *level-range-cost-bound* is the simple recurrence form: a vertex term scaled by the recursion level and an outgoing-edge term scaled by the range coefficient. The amortised bound *bmssp-amortized-cost-bound* adds the extra direct-edge range term used by the bucketed partition, where direct inserts have their own cost story.

The summation lemmas below say that if every recursive child call satisfies one of these bounds on its own range slice, then the sum of child costs can be rewritten as sums over the corresponding child range lists. Later theories collapse those sums using the disjointness lemmas above.

definition *level-range-cost-bound* **where**
level-range-cost-bound A R l X =
A ∗ *l* ∗ *card X* + *R* ∗ *card* (*outgoing-edges X*)

definition *bmssp-amortized-cost-bound* **where**
bmssp-amortized-cost-bound A R h t q L B X =
A ∗ *L* ∗ *card X* + (*R* + *q* ∗ *h*) ∗ *card* (*outgoing-edges X*) +
t ∗ *card* (*outgoing-edges-range X 0 B*)

lemma *sum-bmssp-amortized-child-bounds-le*:

assumes *bounds*:
list-all2 (λ*c-child UB*. *case UB of* (*U-child*, *B-child*) ⇒
c-child ≤
bmssp-amortized-cost-bound A R h t q L B-child U-child)
child-costs (*range-tree-child-bound-pair-list S a betas bs*)
shows *sum-list child-costs* ≤
A ∗ *L* ∗ *sum-list* (*map card* (*range-tree-child-list S a bs*)) +
(*R* + *q* ∗ *h*) ∗
sum-list (*map* (λ*U*. *card* (*outgoing-edges U*))
(*range-tree-child-list S a bs*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-zero-edge-range-list S a betas bs*))
⟨*proof*⟩

lemma *child-costs-le-level-range-chain-bound*:
assumes *mono*: *nondecreasing-from a bs*
and *costs*: *list-all2* (λ*c X*. *c* ≤ *level-range-cost-bound A R l X*) *costs*
(*range-tree-chain-list S a bs B*)
shows *sum-list costs* ≤
A ∗ *l* ∗ *card* (*range-tree-chain S a bs B*) +
R ∗ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

theorem *concrete-partition-loop-trace-card-decomposition*:
assumes *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
shows *card U* =
card (*bound-tree P* (*Fin a*)) + *card* (*range-tree-chain P a bs B′*)
⟨*proof*⟩

theorem *concrete-partition-loop-trace-range-card-le*:
assumes *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
shows *card* (*range-tree-chain P a bs B′*) ≤ *card U*
⟨*proof*⟩

definition *concrete-capped-step-core-cost* **where**
concrete-capped-step-core-cost k cap d S B child-costs =
fp-iter-capped-scan-cost k cap d S S B + *sum-list child-costs*

definition *base-case-scan-cost* **where**
base-case-scan-cost Δ *k x B* = *card* (*outgoing-edges* (*base-case-vertices k x B*))

theorem *base-case-scan-cost-le*:
assumes *degree*: *edge-outdegree-le* Δ
shows *base-case-scan-cost* Δ *k x B* ≤ Δ ∗ *k*
⟨*proof*⟩

theorem *base-case-scan-cost-le-outgoing-edges*:
base-case-scan-cost Δ *k x B* ≤
card (*outgoing-edges* (*base-case-vertices k x B*))

⟨*proof*⟩

theorem *base-case-scan-cost-le-edge-count*:
base-case-scan-cost edge-count k x B ≤ *edge-count* ∗ *k*
⟨*proof*⟩

definition *concrete-capped-step-accounted-cost* **where**
concrete-capped-step-accounted-cost k cap d S B child-costs edge-costs =
fp-iter-capped-scan-cost k cap d S S B +
sum-list child-costs + *sum-list edge-costs*

theorem *concrete-capped-step-core-cost-le-range*:
assumes *S-subset*: *S* ⊆ *V*
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* ≤ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* (λ*c X*. *c* ≤ *C* ∗ *card X*) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
shows *concrete-capped-step-core-cost k cap d S B costs* ≤
k ∗ Δ ∗ *cap* +
C ∗ *card* (*range-tree-chain*
(*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
⟨*proof*⟩

theorem *concrete-capped-step-core-cost-le-level-chain*:
assumes *S-subset*: *S* ⊆ *V*
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* ≤ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* (λ*c X*. *c* ≤ *level-range-cost-bound A R l X*) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
shows *concrete-capped-step-core-cost k cap d S B costs* ≤
k ∗ Δ ∗ *cap* +
A ∗ *l* ∗
card (*range-tree-chain* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*) +
R ∗ *card* (*outgoing-edges*
(*range-tree-chain* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*))
⟨*proof*⟩

theorem *concrete-capped-step-core-cost-closes-level-bound*:
assumes *S-subset*: *S* ⊆ *V*
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* ≤ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* (λ*c X*. *c* ≤ *level-range-cost-bound A R l X*) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)

and *scan-budget*: $k * \Delta * cap \leq A * card\ U\text{-}loop$
shows *concrete-capped-step-core-cost k cap d S B costs* $\leq$
level-range-cost-bound A R (*Suc l*) *U-loop*
⟨*proof*⟩

lemma *scan-budget-from-loop-card*:
assumes *degree-factor*: $\Delta \leq A$
and *progress*: $k * cap \leq card\ U\text{-}loop$
shows $k * \Delta * cap \leq A * card\ U\text{-}loop$
⟨*proof*⟩

lemma *scan-insert-budget-from-loop-card*:
assumes *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *progress*: $k * cap \leq card\ U\text{-}loop$
shows $k * \Delta * cap + t * cap \leq 2 * A * card\ U\text{-}loop$
⟨*proof*⟩

lemma *loop-overhead-budget-from-progress*:
assumes *factor*: $Suc\ t \leq A$
and *progress*: $M * q \leq card\ U$
shows $(Suc\ t) * M * q \leq A * card\ U$
⟨*proof*⟩

theorem *concrete-capped-step-core-cost-closes-level-bound-from-progress*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* $\leq$ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* ($\lambda c\ X.\ c \leq$ *level-range-cost-bound A R l X*) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
and *degree-factor*: $\Delta \leq A$
and *progress*: $k * cap \leq card\ U\text{-}loop$
shows *concrete-capped-step-core-cost k cap d S B costs* $\leq$
level-range-cost-bound A R (*Suc l*) *U-loop*
⟨*proof*⟩

theorem *concrete-capped-step-core-cost-le-loop-vertices*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* $\leq$ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* ($\lambda c\ X.\ c \leq C * card\ X$) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
shows *concrete-capped-step-core-cost k cap d S B costs* $\leq$
$k * \Delta * cap + C * card\ U\text{-}loop$
⟨*proof*⟩

theorem *concrete-capped-step-accounted-cost-le*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card* $S \leq$ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* ($\lambda c\ X.\ c \leq C * card\ X$) *child-costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
and *edge-costs*: *list-all2* ($\lambda c\ X.\ c \leq R * card\ (outgoing\text{-}edges\ X)$) *edge-costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
shows *concrete-capped-step-accounted-cost k cap d S B child-costs edge-costs* $\leq$
$k * \Delta * cap + C * card\ U\text{-}loop +$
$R * card\ (outgoing\text{-}edges$
(*range-tree-chain* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*))
⟨*proof*⟩

theorem *full-operational-partition-loop-state-trace-and-degree-cost*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d′ D a betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *degree*: *edge-outdegree-le* Δ
obtains *child-costs child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
length child-costs = *length child-sets*
$\bigwedge X.\ X \in set\ child\text{-}sets \Longrightarrow X \subseteq V$
$c \leq sum\text{-}list\ child\text{-}costs + M * length\ child\text{-}costs +$
$t * (\Delta * sum\text{-}list\ (map\ card\ child\text{-}sets) + M * length\ child\text{-}costs)$
⟨*proof*⟩

definition *partition-initial-insert-cost-bound* **where**
partition-initial-insert-cost-bound c t P $\longleftrightarrow$ $c \leq t * card\ P$

lemma *partition-initial-insert-cost-boundD*:
assumes *partition-initial-insert-cost-bound c t P*
shows $c \leq t * card\ P$
⟨*proof*⟩

lemma *partition-initial-insert-cost-bound-le*:
assumes *cost*: *partition-initial-insert-cost-bound c t P*
and *card-le*: *card* $P \leq N$
shows $c \leq t * N$
⟨*proof*⟩

lemma *find-pivots-pivots-capped-subset*:
find-pivots-pivots-capped k cap d S B $\subseteq S$
⟨*proof*⟩

lemma *find-pivots-pivots-capped-card-le*:
assumes *S-subset*: $S \subseteq V$
shows *card* (*find-pivots-pivots-capped k cap d S B*) $\leq$ *card S*
⟨*proof*⟩

lemma *find-pivots-pivots-capped-scaled-card-le*:
assumes *S-subset*: $S \subseteq V$
and *S-k-cap*: $k * card\ S \leq cap$
shows $k * card$ (*find-pivots-pivots-capped k cap d S B*) $\leq$ *cap*
⟨*proof*⟩

lemma *find-pivots-pivots-capped-tree-antichain*:
assumes *anti*: *tree-antichain S*
shows *tree-antichain* (*find-pivots-pivots-capped k cap d S B*)
⟨*proof*⟩

lemma *partition-initial-insert-cost-bound-capped-pivots-le*:
assumes *cost*: *partition-initial-insert-cost-bound c t*
(*find-pivots-pivots-capped k cap d S B*)
and *S-subset*: $S \subseteq V$
and *S-cap*: *card S* $\leq$ *cap*
shows $c \leq t * cap$
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-cost-le*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* $\leq$ *cap*
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
$k * \Delta * cap + t * cap$
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-cost-le-edge-count*:
assumes *S-subset*: $S \subseteq V$
and *S-cap*: *card S* $\leq$ *cap*
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
$k * edge\text{-}count * cap + t * cap$
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-budget-from-progress*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card S* $\leq$ *cap*
and *insert*: *partition-initial-insert-cost-bound c-insert t*

(*find-pivots-pivots-capped k cap d S B*)
and *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *progress*: $k * cap \leq card\ U$
shows *fp-iter-capped-scan-cost k cap d S S B + c-insert* $\leq$
$2 * A * card\ U$
⟨*proof*⟩

lemma *find-pivots-pivots-capped-card-le-seen-if-no-overflow*:
assumes *S-subset*: $S \subseteq V$
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *no-overflow*:
$\neg\ card$ (*find-pivots-seen-capped k cap d S B*) $> cap$
shows *card* (*find-pivots-pivots-capped k cap d S B*) $\leq$
card (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *partition-initial-insert-cost-bound-capped-pivots-le-seen-scaled-from-mult*:
assumes *insert*: *partition-initial-insert-cost-bound c t P*
and *pivot-mult*: $k * card\ P \leq card\ W$
and *insert-factor*: $t \leq A * k$
shows $c \leq A * card\ W$
⟨*proof*⟩

lemma *find-pivots-pivots-capped-card-times-le-seen*:
assumes *S-subset*: $S \subseteq V$
and *anti*: *tree-antichain S*
and *S-k-cap*: $k * card\ S \leq cap$
shows $k * card$ (*find-pivots-pivots-capped k cap d S B*) $\leq$
card (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *partition-initial-insert-cost-bound-capped-pivots-le-seen-scaled*:
assumes *insert*: *partition-initial-insert-cost-bound c t*
(*find-pivots-pivots-capped k cap d S B*)
and *S-subset*: $S \subseteq V$
and *anti*: *tree-antichain S*
and *S-k-cap*: $k * card\ S \leq cap$
and *insert-factor*: $t \leq A * k$
shows $c \leq A * card$ (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *partition-initial-insert-cost-bound-capped-pivots-le-seen-if-no-overflow*:
assumes *insert*: *partition-initial-insert-cost-bound c t*
(*find-pivots-pivots-capped k cap d S B*)
and *S-subset*: $S \subseteq V$
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$

and *no-overflow*:
¬ *card* (*find-pivots-seen-capped k cap d S B*) > *cap*
shows $c \leq t * card$ (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *partition-initial-insert-cost-bound-capped-pivots-le-seen*:
assumes *insert*: *partition-initial-insert-cost-bound c t*
(*find-pivots-pivots-capped k cap d S B*)
and *S-subset*: $S \subseteq V$
and *S-cap*: *card* $S \leq$ *cap*
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
shows $c \leq t * card$ (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-cost-le-seen-if-no-overflow*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *no-overflow*:
¬ *card* (*find-pivots-seen-capped k cap d S B*) > *cap*
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
shows *fp-iter-capped-scan-cost k cap d S S B + c-insert* ≤
$(k * \Delta + t) * card$ (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-cost-le-seen*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card* $S \leq$ *cap*
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
shows *fp-iter-capped-scan-cost k cap d S S B + c-insert* ≤
$(k * \Delta + t) * card$ (*find-pivots-seen-capped k cap d S B*)
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-cost-le-seen-scaled*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *anti*: *tree-antichain S*
and *S-k-cap*: $k * card\ S \leq cap$
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
and *insert-factor*: $t \leq A\text{-}insert * k$
shows *fp-iter-capped-scan-cost k cap d S S B + c-insert* ≤

$(k * \Delta + A\text{-}insert) * card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B)$
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-budget-from-seen-scaled*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *anti*: *tree-antichain S*
and *S-k-cap*: $k * card\ S \leq cap$
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
and *insert-factor*: $t \leq A\text{-}insert * k$
and *factor*: $k * \Delta + A\text{-}insert \leq A$
and *seen-progress*:
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ U$
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* ≤
$A * card\ U$
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-budget-from-seen*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *no-overflow*:
$\neg\ card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B) > cap$
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
and *factor*: $k * \Delta + t \leq A$
and *seen-progress*:
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ U$
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* ≤
$A * card\ U$
⟨*proof*⟩

lemma *find-pivots-scan-and-initial-insert-budget-from-seen-total*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: $card\ S \leq cap$
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *insert*: *partition-initial-insert-cost-bound c-insert t*
(*find-pivots-pivots-capped k cap d S B*)
and *factor*: $k * \Delta + t \leq A$
and *seen-progress*:
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ U$
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* ≤
$A * card\ U$
⟨*proof*⟩

The relation below is the first costed operational run in this file. It

refines the full operational relation by adding a natural-number cost to each BMSSP call and each partition loop. The constructors charge base-case scans, FindPivots scans and initial inserts, pulls, batch prepends, recursive child calls, and the remaining tail of the loop.

This relation is intentionally generic in the partition-cost predicates. It does not know whether a sorted list, bucketed partition, or another data structure implements the primitive operations. Its job is to expose the exact cost skeleton that the later refinement layers and the final headline theorem instantiate.

```
inductive costed-full-operational-partition-loop-state
  and costed-full-operational-bmssp where
  Cost-State-Done:
    keys-of D = {} ⟹
     bound-le (Fin a) B ⟹
     complete-on d' (bound-tree P (Fin a)) ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
       [] [] (Fin a) [range-tree P a (Fin a)]
       (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0
| Cost-State-Step:
    pull-separates D (M-of l) Bmax S-pull beta D-pull ⟹
     bound-le (Fin beta) B ⟹
     bmssp-pre-full d (split-below d P beta) (Fin beta) ⟹
     S-pull = split-below d P beta ⟹
     a ≤ b ⟹
     complete-on d' (bound-tree P (Fin a)) ⟹
     costed-full-operational-bmssp Δ M-of t k cap l d S-pull (Fin beta)
       d-child (Fin b) U-child c-child ⟹
     complete-preserved d-child d' U-child ⟹
     batch =
       edge-relaxation-pairs-between d-child U-child b beta @
       label-pairs-between d S-pull b beta ⟹
     D-next = batch-min-update D-pull batch ⟹
     partition-pull-cost-bound c-pull S-pull ⟹
     partition-batch-cost-bound c-batch t batch ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d'
       D-next b betas bs B' Us-tail U-tail c-tail ⟹
     c = c-pull + c-batch + c-child + c-tail ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
       (beta # betas) (b # bs) B'
       (range-tree P a (Fin b) # Us-tail)
       (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # Us-tail))) c
| Cost-Base:
    S = {x} ⟹
     costed-full-operational-bmssp Δ M-of t k cap 0 d S B
       (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
       (base-case-bound k x B)
       (base-case-vertices k x B)
```

```
      (base-case-scan-cost Δ k x B)
| Cost-Step:
    D = label-partition-view
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) ⟹
     partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B) ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop ⟹
     complete-on d'
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     U = U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
     costed-full-operational-bmssp Δ M-of t k cap (Suc l) d S B d' B' U c

theorem costed-full-operational-partition-loop-state-refines:
  costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
    betas bs B' Us U c ⟹
   full-operational-partition-loop-state (M-of l) t k cap l d P B d' D a
    betas bs B' Us U c
and costed-full-operational-bmssp-refines:
  costed-full-operational-bmssp Δ M-of t k cap l d S B d' B' U c ⟹
   full-operational-bmssp k cap l d S B d' B' U
⟨proof⟩

theorem costed-full-operational-bmssp-correct:
  assumes run:
    costed-full-operational-bmssp Δ M-of t k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bmssp-post-full d S B d' B' U
⟨proof⟩

lemma costed-full-operational-partition-loop-state-lengths:
  assumes run:
    costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
      betas bs B' Us U c
  shows length betas = length bs
    and length Us = Suc (length bs)
⟨proof⟩

theorem finite-initial-label-costed-full-operational-top-level-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run:
      costed-full-operational-bmssp Δ M-of t k cap l finite-initial-label {s}
```

Infinity d′ Infinity U c
shows *sssp-correct d′*
⟨*proof*⟩

theorem *costed-full-operational-partition-loop-state-trace-and-degree-cost*:
assumes *run*:
costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d′ D a
betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *degree*: *edge-outdegree-le Δ*
obtains *child-costs child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
length child-costs = *length child-sets*
⋀*X*. *X* ∈ *set child-sets* ⟹ *X* ⊆ *V*
c ≤ *sum-list child-costs* + (*M-of l*) ∗ *length child-costs* +
t ∗ (*Δ* ∗ *sum-list* (*map card child-sets*) + (*M-of l*) ∗ *length child-costs*)
⟨*proof*⟩

theorem *costed-full-operational-partition-loop-state-child-call-data*:
costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d′ D a
betas bs B′ Us U c ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
∃ *child-costs child-sets*.
length child-costs = *length child-sets* ∧
length child-costs = *length bs* ∧
list-all2
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-costs child-sets ∧
(∀ *X*∈*set child-sets*. *X* ⊆ *V*) ∧
c ≤ *sum-list child-costs* + (*M-of l*) ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
(*M-of l*) ∗ *length child-costs*)
and *costed-full-operational-bmssp-child-call-data-trivial*:
costed-full-operational-bmssp Δ M-of t k cap l d S B d′ B′ U c ⟹ *True*
⟨*proof*⟩

theorem *costed-full-operational-partition-loop-state-child-calls-and-cost*:
assumes *run*:
costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d′ D a
betas bs B′ Us U c

and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
obtains *child-costs child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
length child-costs = *length child-sets*
length child-costs = *length bs*
list-all2
(λ*c-child U-child*. $\exists$ *S-child B-child d-child B-child′*.
costed-full-operational-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l*)
child-costs child-sets
$\bigwedge X.\ X \in$ *set child-sets* $\Longrightarrow X \subseteq V$
c $\leq$ *sum-list child-costs* + (*M-of l*) $*$ *length child-costs* +
t $*$ (*sum-list* (*map* ($\lambda X.$ *card* (*outgoing-edges X*)) *child-sets*) +
(*M-of l*) $*$ *length child-costs*)
⟨*proof*⟩

theorem *costed-full-operational-partition-loop-state-cost-from-child-bound*:
assumes *run*:
costed-full-operational-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *degree*: *edge-outdegree-le* Δ
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*costed-full-operational-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *C* $*$ *card U-child*
obtains *child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
$\bigwedge X.\ X \in$ *set child-sets* $\Longrightarrow X \subseteq V$
length child-sets = *length bs*
c $\leq$ (*C* + *t* $*$ Δ) $*$ *sum-list* (*map card child-sets*) +
(*Suc t*) $*$ (*M-of l*) $*$ *length bs*
⟨*proof*⟩

end

end
theory *BMSSP-Range-Costed*

imports *BMSSP-Complexity*
begin

30 Range-Synchronised Costed Runs

The algorithm's time proof charges the recursive calls in one BMSSP loop to the disjoint distance ranges cut out by the successive returned bounds. The earlier costed relation keeps enough information for correctness and local data-structure costs, but its child-call cost is attached to the raw child output. This theory strengthens the operational run with the range invariant: every child output is exactly the next range slice.

That strengthening is the point at which the informal amortised analysis in the paper becomes a reusable Isabelle recurrence. A child call is not charged against the whole parent tree. It is charged against a slice *range-tree* between the previous loop bound and the child output bound. The list version *range-tree-child-list* records all such slices for one parent loop. Since those slices are disjoint and lie inside the final loop output, their vertex and outgoing-edge sums can be bounded by the final output itself.

The exported cost shape is *level-range-cost-bound*. Its first term counts completed vertices multiplied by a level factor; its second term counts outgoing edges with the range cost coefficient. The main technical work below is to show that the local costs of pulling, batching, and recursive child calls close under this shape when moving from level l to *Suc l*.

There are two related relations. The first one uses a single batch cost parameter for the concatenation of edge relaxations and reinserted source labels. The split relation refines it by charging edge batches and source batches separately. That split form is useful for later progress arguments: edge batches are paid for by outgoing edges of the completed ranges, while source batches are paid for by the number of child sources exposed by the partition pulls.

context *unique-shortest-digraph*
begin

lemma *child-costs-le-level-range-child-list-bound*:
  assumes *mono*: *nondecreasing-from a bs*
    and *costs*: *list-all2* ($\lambda c\ X.\ c \leq$ *level-range-cost-bound A R l X*) *costs*
      (*range-tree-child-list S a bs*)
  shows *sum-list costs* $\leq$
    $A * l *$ *card* (*range-tree-chain S a bs B*) $+$
    $R *$ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
⟨*proof*⟩

inductive *range-costed-partition-loop-state*
  and *range-costed-bmssp* **where**
  Range-State-Done:

keys-of D = {} ⟹
bound-le (Fin a) B ⟹
complete-on d′ (bound-tree P B) ⟹
range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
[] [] B [range-tree P a B]
(bound-tree P (Fin a) ∪ ⋃(set [range-tree P a B])) 0 []
| *Range-State-Stop*:
bound-le (Fin a) B ⟹
complete-on d′ (bound-tree P (Fin a)) ⟹
k ∗ cap ≤ card (bound-tree P (Fin a)) ⟹
range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
[] [] (Fin a) [range-tree P a (Fin a)]
(bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0 []
| *Range-State-Step*:
pull-separates D (M-of l) Bmax S-pull beta D-pull ⟹
bound-le (Fin beta) B ⟹
bmssp-pre-full d S-pull (Fin beta) ⟹
(∀ x∈S-pull. reachable s x) ⟹
a ≤ b ⟹
bound-le (Fin a) B′ ⟹
complete-on d′ (bound-tree P (Fin a)) ⟹
card (bound-tree P (Fin a)) < k ∗ cap ⟹
range-costed-bmssp Δ M-of t k cap l d S-pull (Fin beta)
d-child (Fin b) U-child c-child ⟹
U-child = range-tree P a (Fin b) ⟹
complete-preserved d-child d′ U-child ⟹
batch =
edge-relaxation-pairs-between d-child U-child b beta @
label-pairs-between d S-pull b beta ⟹
D-next = batch-min-update D-pull batch ⟹
partition-pull-cost-bound c-pull S-pull ⟹
partition-batch-cost-bound c-batch t batch ⟹
range-costed-partition-loop-state Δ M-of t k cap l d P B d′
D-next b betas bs B′ Us-tail U-tail c-tail child-costs-tail ⟹
c = c-pull + c-batch + c-child + c-tail ⟹
range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
(beta # betas) (b # bs) B′
(range-tree P a (Fin b) # Us-tail)
(bound-tree P (Fin a) ∪
⋃(set (range-tree P a (Fin b) # Us-tail))) c
(c-child # child-costs-tail)
| *Range-Cost-Base*:
S = {x} ⟹
range-costed-bmssp Δ M-of t k cap 0 d S B
(λv. if v ∈ base-case-vertices k x B then dist s v else d v)
(base-case-bound k x B)
(base-case-vertices k x B)
(base-case-scan-cost Δ k x B)
| *Range-Cost-Step*:

D = label-partition-view
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) ⟹
partition-initial-insert-cost-bound c-insert t
(find-pivots-pivots-capped k cap d S B) ⟹
range-costed-partition-loop-state Δ M-of t k cap l
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) B d′ D a betas bs B′
Us U-loop c-loop child-costs-loop ⟹
complete-on d′
{v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
U = U-loop ∪
{v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
range-costed-bmssp Δ M-of t k cap (Suc l) d S B d′ B′ U c

The mutually inductive relation above adds the accounting data that the operational theory intentionally omitted. A loop derivation carries the pull bounds *betas*, the child output bounds *bs*, the concrete range slices *Us*, a total cost *c*, and the list of recursive child costs. The step constructor synchronises these fields by requiring the returned child set to be exactly *range-tree* for the current interval.

The BMSSP branch keeps the usual split between the base case and the recursive step. In the recursive step, FindPivots pays the scan and initial insertion cost, then the range-synchronised loop pays all pulls, batches, and child calls. The correctness argument later forgets the costs and reuses the operational trace theorem; the runtime argument keeps the cost and child-cost lists.

inductive-cases *range-costed-bmssp-zeroE*:
range-costed-bmssp Δ M-of t k cap 0 d S B d′ B′ U c

inductive-cases *range-costed-bmssp-SucE*:
range-costed-bmssp Δ M-of t k cap (Suc l) d S B d′ B′ U c

lemma *range-costed-partition-loop-state-mono*:
assumes *run*:
range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
betas bs B′ Us U c child-costs
shows *nondecreasing-from a bs*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-child-call-ranges*:
assumes *run*:
range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
betas bs B′ Us U c child-costs
shows *length child-costs = length bs*
and *list-all2*
(λc-child U-child. ∃ S-child B-child d-child B-child′.

range-costed-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child' U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
$(\forall\, x{\in}$*S-child. reachable s x*$)\ \wedge$
card S-child $\leq$ *M-of l*)
child-costs (*range-tree-child-list P a bs*)
⟨*proof*⟩

lemma *range-costed-pull-cost-le-M*:
assumes *pull*: *pull-separates D M Bmax S beta D-pull*
and *cost*: *partition-pull-cost-bound c S*
shows $c \leq M$
⟨*proof*⟩

lemma *range-costed-batch-cost-le*:
assumes *pre*: *bmssp-pre-full d S* (*Fin beta*)
and *batch*:
batch =
edge-relaxation-pairs-between d-child U b beta @
label-pairs-between d S b beta
and *cost*: *partition-batch-cost-bound c t batch*
and *card-S*: *card S* $\leq$ *M*
shows $c \leq t * ($*card* (*outgoing-edges U*) $+$ *M*$)$
⟨*proof*⟩

lemma *range-costed-split-batch-cost-le*:
assumes *pre*: *bmssp-pre-full d S* (*Fin beta*)
and *edge-cost*:
partition-batch-cost-bound c-edges t
(*edge-relaxation-pairs-between d-child U b beta*)
and *source-cost*:
partition-batch-cost-bound c-sources h
(*label-pairs-between d S b beta*)
and *c-def*: *c* $=$ *c-edges* $+$ *c-sources*
and *card-S*: *card S* $\leq$ *M*
shows $c \leq t * $*card* (*outgoing-edges U*) $+\ h * M$
⟨*proof*⟩

lemma *range-costed-split-batch-cost-le-card*:
assumes *pre*: *bmssp-pre-full d S* (*Fin beta*)
and *edge-cost*:
partition-batch-cost-bound c-edges t
(*edge-relaxation-pairs-between d-child U b beta*)
and *source-cost*:
partition-batch-cost-bound c-sources h
(*label-pairs-between d S b beta*)
and *c-def*: *c* $=$ *c-edges* $+$ *c-sources*
shows $c \leq t * $*card* (*outgoing-edges U*) $+\ h *$ *card S*
⟨*proof*⟩

lemma *range-costed-split-batch-cost-le-range-card*:
  assumes *pre*: *bmssp-pre-full d S* (*Fin beta*)
    and *edge-cost*:
      partition-batch-cost-bound c-edges t
        (*edge-relaxation-pairs-between d-child U b beta*)
    and *source-cost*:
      partition-batch-cost-bound c-sources h
        (*label-pairs-between d S b beta*)
    and *complete*: *complete-on d-child U*
    and *reaches*: $\bigwedge u.\ u \in U \Longrightarrow$ *reachable s u*
    and *c-def*: *c* = *c-edges* + *c-sources*
  shows *c* ≤ *t* ∗ *card* (*outgoing-edges-range U b* (*Fin beta*)) + *h* ∗ *card S*
⟨*proof*⟩

The single-batch relation is convenient, but the sharper progress lemmas need to distinguish where a batch entry came from. Edge entries are generated by *edge-relaxation-pairs-between* and are charged to outgoing edges of the just-completed range. Source entries are generated by *label-pairs-between* and are charged to the number of pulled child sources. The split relation below records both costs explicitly.

This is a refinement, not a different algorithm. Its proof to the single-batch relation uses the append rule for *partition-batch-cost-bound* and the elementary fact that a source-batch coefficient bounded by the edge coefficient can be absorbed into the combined batch budget.

inductive *split-range-costed-partition-loop-state*
  and *split-range-costed-bmssp* **where**
  Split-Range-State-Done:
    keys-of D = {} ⟹
     bound-le (*Fin a*) *B* ⟹
     complete-on d′ (*bound-tree P B*) ⟹
     split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
       [] [] *B* [*range-tree P a B*]
       (*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a B*])) *0* []
| *Split-Range-State-Stop*:
    bound-le (*Fin a*) *B* ⟹
     complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
     k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ⟹
     split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
       [] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
       (*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)])) *0* []
| *Split-Range-State-Step*:
    pull-separates D (*M-of l*) *Bmax S-pull beta D-pull* ⟹
     bound-le (*Fin beta*) *B* ⟹
     bmssp-pre-full d S-pull (*Fin beta*) ⟹
     (∀ *x*∈*S-pull*. *reachable s x*) ⟹
     a ≤ *b* ⟹
     bound-le (*Fin a*) *B′* ⟹

complete-on d' (*bound-tree* P (*Fin* a)) $\Longrightarrow$
card (*bound-tree* P (*Fin* a)) $<$ $k * cap$ $\Longrightarrow$
split-range-costed-bmssp Δ *M-of* t h k *cap* l d *S-pull* (*Fin beta*)
d-child (*Fin* b) *U-child c-child* $\Longrightarrow$
U-child = *range-tree* P a (*Fin* b) $\Longrightarrow$
complete-preserved d-child d' *U-child* $\Longrightarrow$
edge-batch = *edge-relaxation-pairs-between d-child U-child b beta* $\Longrightarrow$
source-batch = *label-pairs-between d S-pull b beta* $\Longrightarrow$
batch = *edge-batch* @ *source-batch* $\Longrightarrow$
D-next = *batch-min-update D-pull batch* $\Longrightarrow$
partition-pull-cost-bound c-pull S-pull $\Longrightarrow$
partition-batch-cost-bound c-edges t edge-batch $\Longrightarrow$
partition-batch-cost-bound c-sources h source-batch $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l d P B d'
D-next b betas bs B' *Us-tail U-tail c-tail child-costs-tail* $\Longrightarrow$
c = *c-pull* + *c-edges* + *c-sources* + *c-child* + *c-tail* $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l d P B d' D a
(*beta* # *betas*) (b # *bs*) B'
(*range-tree* P a (*Fin* b) # *Us-tail*)
(*bound-tree* P (*Fin* a) $\cup$
$\bigcup$(*set* (*range-tree* P a (*Fin* b) # *Us-tail*))) c
(*c-child* # *child-costs-tail*)
| *Split-Range-Cost-Base*:
$S = \{x\}$ $\Longrightarrow$
split-range-costed-bmssp Δ *M-of* t h k *cap* 0 d S B
(λv. *if* $v \in$ *base-case-vertices* k x B *then dist* s v *else* d v)
(*base-case-bound* k x B)
(*base-case-vertices* k x B)
(*base-case-scan-cost* Δ k x B)
| *Split-Range-Cost-Step*:
D = *label-partition-view*
(*find-pivots-label-capped* k *cap* d S B)
(*find-pivots-pivots-capped* k *cap* d S B) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped* k *cap* d S B) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l
(*find-pivots-label-capped* k *cap* d S B)
(*find-pivots-pivots-capped* k *cap* d S B) B d' D a *betas bs* B'
Us U-loop c-loop child-costs-loop $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* S B'. *find-pivots-label-capped* k *cap* d S B v = *dist* s $v\}$ $\Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree* S B'. *find-pivots-label-capped* k *cap* d S B v = *dist* s $v\}$ $\Longrightarrow$
c = *fp-iter-capped-scan-cost* k *cap* d S S B + *c-insert* + *c-loop* $\Longrightarrow$
split-range-costed-bmssp Δ *M-of* t h k *cap* (*Suc* l) d S B d' B' U c

inductive-cases *split-range-costed-bmssp-zeroE*:
split-range-costed-bmssp Δ *M-of* t h k *cap* 0 d S B d' B' U c

inductive-cases *split-range-costed-bmssp-SucE*:
split-range-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*

theorem *split-range-costed-partition-loop-state-refines*:
shows
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
h ≤ *t* ⟹
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹
h ≤ *t* ⟹
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c*
⟨*proof*⟩

The refinement theorem ⟦*split-range-costed-partition-loop-state ?*Δ *?M-of ?t ?h ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs*; *?h* ≤ *?t*⟧ ⟹ *range-costed-partition-loop-state ?*Δ *?M-of ?t ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs*

⟦*split-range-costed-bmssp ?*Δ *?M-of ?t ?h ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U ?c*; *?h* ≤ *?t*⟧ ⟹ *range-costed-bmssp ?*Δ *?M-of ?t ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U ?c* is the formal connection between the two presentations. Once it is proved, any semantic theorem established for the single-batch relation can be reused for the split relation. The remaining split-specific lemmas therefore focus on costs: extracting child sources, bounding edge and source batches separately, and then recombining those bounds into the level recurrence.

lemma *split-range-costed-partition-loop-state-mono*:
assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
shows *nondecreasing-from a bs*
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-child-call-ranges*:
assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
shows *length child-costs* = *length bs*
and *list-all2*
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-costs (*range-tree-child-list P a bs*)
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-cost-bound-by-child-sources*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
$\exists$ *child-sources*.
length child-sources = *length bs* $\wedge$
list-all2
(λ*S-child U-child*. $\exists$ *c-child B-child d-child B-child′*.
split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l*)
child-sources (*range-tree-child-list P a bs*) $\wedge$
c $\leq$ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t $*$ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
h $*$ *sum-list* (*map card child-sources*)
and *split-range-costed-bmssp-child-sources-trivial*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-cost-from-child-and-source-bounds*:
assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child $\leq$ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x*. *x* $\in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
shows *c* $\leq$
A $*$ *L* $*$ *card* (*range-tree-chain P a bs B′*) +
(*R* + *t*) $*$ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
Suc h $*$ *card* (*range-tree-chain P a bs B′*)
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-child-cost-bounds*:
assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-bound*:

$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*
shows *list-all2* (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-cost-from-child-bound*:
assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*⟧
⟹ *card S-child* ≤ *card U-child*
shows *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
(*R* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
Suc h ∗ *card* (*range-tree-chain P a bs B′*)
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-trace*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
($\bigwedge$*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
concrete-partition-loop-trace P B a bs d′ B′ Us U
and *split-range-costed-bmssp-correct*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹

$(\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x) \Longrightarrow$
$\mathit{bmssp\text{-}post\text{-}full}\ d\ S\ B\ d'\ B'\ U$
⟨*proof*⟩

The next block closes the level recurrence for the split relation. The input hypotheses have the same shape as the informal induction in the paper: each child call already satisfies a level-L bound on its own output slice, and every pulled child source set makes enough progress to be charged to that slice. The trace theorem supplies the geometric fact needed to sum these charges: the range slices form a chain inside the parent output.

Once the slice sums are bounded by the parent output and its outgoing edges, the arithmetic is straightforward. The child level factor contributes $A * L$, the source-progress term contributes one more A, and the edge batch cost contributes the extra t in the range coefficient. This is why the target has level $\mathit{Suc}\ L$ and coefficient $R + t$.

theorem *split-range-costed-partition-loop-state-closes-level-bound-from-child-cost-bounds-general*:
assumes *run*:
$\mathit{split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ P\ B\ d'\ D\ a$
$\mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ U\ c\ \mathit{child\text{-}costs}$
and *sound*: $\mathit{sound\text{-}label}\ d$
and *pre*: $\mathit{bmssp\text{-}pre\text{-}full}\ d\ P\ B$
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow \mathit{reachable}\ s\ x$
and *child-cost-bounds*:
$\mathit{list\text{-}all2}\ (\lambda \mathit{c\text{-}child}\ \mathit{U\text{-}child}.$
$\mathit{c\text{-}child} \leq \mathit{level\text{-}range\text{-}cost\text{-}bound}\ A\ R\ L\ \mathit{U\text{-}child})$
$\mathit{child\text{-}costs}\ (\mathit{range\text{-}tree\text{-}child\text{-}list}\ P\ a\ \mathit{bs})$
and *source-progress*:
$\bigwedge \mathit{S\text{-}child}\ \mathit{B\text{-}child}\ \mathit{d\text{-}child}\ \mathit{B\text{-}child}'\ \mathit{U\text{-}child}\ \mathit{c\text{-}child}.$
$[\![\mathit{split\text{-}range\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ \mathit{S\text{-}child}\ \mathit{B\text{-}child}$
$\mathit{d\text{-}child}\ \mathit{B\text{-}child}'\ \mathit{U\text{-}child}\ \mathit{c\text{-}child};$
$\mathit{bmssp\text{-}pre\text{-}full}\ d\ \mathit{S\text{-}child}\ \mathit{B\text{-}child};$
$\bigwedge x.\ x \in \mathit{S\text{-}child} \Longrightarrow \mathit{reachable}\ s\ x;$
$\mathit{card}\ \mathit{S\text{-}child} \leq \mathit{M\text{-}of}\ l]\!]$
$\Longrightarrow \mathit{card}\ \mathit{S\text{-}child} \leq \mathit{card}\ \mathit{U\text{-}child}$
and *source-factor*: $\mathit{Suc}\ h \leq A$
shows $c \leq A * \mathit{Suc}\ L * \mathit{card}\ U + (R + t) * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ U)$
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-closes-level-bound-from-source-progress-general*:
assumes *run*:
$\mathit{split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ P\ B\ d'\ D\ a$
$\mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ U\ c\ \mathit{child\text{-}costs}$
and *sound*: $\mathit{sound\text{-}label}\ d$
and *pre*: $\mathit{bmssp\text{-}pre\text{-}full}\ d\ P\ B$
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow \mathit{reachable}\ s\ x$
and *child-bound*:

$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
and *source-factor*: *Suc h* $\leq$ *A*
shows $c \leq A * Suc\ L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *split-range-costed-nonbase-step-closes-level-bound-from-source-progress-general*:
assumes *loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
and *source-factor*: *Suc h* $\leq$ *A*
and *U-def*: *U* = *U-loop* $\cup$ *W*
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* $\leq$ *A* $*$ *card U*
and *c-def*: *c* = *c-scan-insert* + *c-loop*
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$

$(R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

lemma *split-range-costed-base-level-bound*:
assumes *R-pos*: $0 < R$
shows *base-case-scan-cost* $\Delta\ k\ x\ B \leq$
level-range-cost-bound $A\ R\ (Suc\ 0)\ (base\text{-}case\text{-}vertices\ k\ x\ B)$
⟨*proof*⟩

theorem *split-range-costed-bmssp-level-bound-from-source-progress-and-local-budgets*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$card\ S \leq$ *M-of l* $\Longrightarrow$
$card\ S \leq card\ U$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d′ a betas bs B′ Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d′
$\{v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* $= dist\ s\ v\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* $= dist\ s\ v\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq A * card\ U$
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x$
shows $c \leq$ *level-range-cost-bound* $A\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *split-range-costed-bmssp-level-bound-from-source-progress-budgets*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: *Suc* $h \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ $k * cap \leq$ *card U*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *c* $\leq$ *level-range-cost-bound* $(2 * A)$ $(R + l * t)$ $(2 * l + 1)$ *U*
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'*
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-local-budget-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: *Suc h* $\leq A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
fp-iter-capped-scan-cost k cap d S S B $+$ *c-insert* $\leq A *$ *card U*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
$c \leq$ *level-range-cost-bound A* $(R + l * t)$ $(2 * l + 1)$ *U*
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-local-budget-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: *Suc h* $\leq A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$

bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*) $\Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
{*v* $\in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*} $\Longrightarrow$
U = *U-loop* $\cup$
{*v* $\in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*} $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*) $\Longrightarrow$
fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq$ *A* $*$ *card U*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* {*s*}
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
c $\leq$ *A* $*$ (*2* $*$ *l* + *1*) $*$ *vertex-count* + (*R* + *l* $*$ *t*) $*$ *edge-count*
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-source-progress-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: *0* < *R*
and *insert-factor*: *t* $\leq$ *A* $*$ *k*
and *source-factor*: *Suc h* $\leq$ *2* $*$ *A*
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*) $\Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.

$D = \textit{label-partition-view}$
($\textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B$)
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $\Longrightarrow$
$\textit{partition-initial-insert-cost-bound}\ \textit{c-insert}\ t$
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $\Longrightarrow$
$\textit{split-range-costed-partition-loop-state}\ \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l$
($\textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B$)
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $B\ d'\ D\ a\ \textit{betas}\ bs\ B'$
$\textit{Us}\ \textit{U-loop}\ \textit{c-loop}\ \textit{child-costs} \Longrightarrow$
$\textit{complete-on}\ d'$
$\{v \in \textit{bound-tree}\ S\ B'.\ \textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B\ v = \textit{dist}\ s\ v\} \Longrightarrow$
$U = \textit{U-loop} \cup$
$\{v \in \textit{bound-tree}\ S\ B'.\ \textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B\ v = \textit{dist}\ s\ v\} \Longrightarrow$
$\textit{sound-label}\ d \Longrightarrow$
$\textit{bmssp-pre-full}\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{reachable}\ s\ x) \Longrightarrow$
$\textit{card}\ S \leq \textit{cap} \wedge k * \textit{cap} \leq \textit{card}\ U$
and *run*:
$\textit{split-range-costed-bmssp}\ \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l\ \textit{finite-initial-label}\ \{s\}$
$\textit{Infinity}\ d'\ \textit{Infinity}\ U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq \textit{level-range-cost-bound}\ (2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow \textit{reachable}\ s\ v$
and *degree*: $\textit{edge-outdegree-le}\ \Delta$
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: $\textit{Suc}\ h \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
$\textit{split-range-costed-bmssp}\ \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow$
$\textit{bmssp-pre-full}\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{reachable}\ s\ x) \Longrightarrow$
$\textit{card}\ S \leq \textit{M-of}\ l \Longrightarrow$
$\textit{card}\ S \leq \textit{card}\ U$
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D\ \textit{c-insert}\ d'\ a\ \textit{betas}\ bs\ B'\ \textit{Us}\ \textit{U-loop}\ \textit{c-loop}$
$\textit{child-costs}\ U.$
$D = \textit{label-partition-view}$
($\textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B$)
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $\Longrightarrow$
$\textit{partition-initial-insert-cost-bound}\ \textit{c-insert}\ t$
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $\Longrightarrow$
$\textit{split-range-costed-partition-loop-state}\ \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l$
($\textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B$)
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $B\ d'\ D\ a\ \textit{betas}\ bs\ B'$

Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
card $S \leq cap \wedge k * cap \leq$ *card* U
and *run*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k\ cap\ l$ *finite-initial-label* $\{s\}$
Infinity d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
⟨*proof*⟩

theorem *split-range-costed-bmssp-source-card-le-if-sound-label-below-output*:
assumes *run*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k\ cap\ l\ d\ S\ B\ d'\ B'\ U\ c$
and *sound*: *sound-label* d
and *pre*: *bmssp-pre-full* $d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)\ B'$
shows *card* $S \leq$ *card* U
⟨*proof*⟩

theorem *split-range-costed-partition-loop-state-success-or-threshold*:
split-range-costed-partition-loop-state Δ *M-of* $t\ h\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
betas bs B' *Us* $U\ c$ *child-costs* $\Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$B' = B \vee k * cap \leq$ *card* U
and *split-range-costed-bmssp-success-or-threshold-trivial*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k\ cap\ l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow$ *True*
⟨*proof*⟩

theorem *split-range-costed-bmssp-Suc-success-or-threshold*:
assumes *run*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k\ cap$ $(Suc\ l)$ $d\ S\ B\ d'\ B'\ U\ c$
and *sound*: *sound-label* d
and *pre*: *bmssp-pre-full* $d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
shows $B' = B \vee k * cap \leq$ *card* U
⟨*proof*⟩

theorem *split-range-costed-bmssp-Suc-source-card-le-from-label-below*:
assumes *run*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k\ cap$ $(Suc\ l)$ $d\ S\ B\ d'\ B'\ U\ c$

and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *S-cap*: *card S* $\leq$ *cap*
and *k-pos*: *0* $<$ *k*
shows *card S* $\leq$ *card U*
⟨*proof*⟩

theorem *split-range-costed-capped-step-threshold-if-not-success*:
assumes *loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* $\cup$
{*v* $\in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *not-success*: $B' \neq B$
shows *k* $*$ *cap* $\leq$ *card U*
⟨*proof*⟩

theorem *split-range-costed-capped-step-scan-insert-budget-if-not-success*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* $\cup$
{*v* $\in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *S-cap*: *card S* $\leq$ *cap*
and *not-success*: $B' \neq B$
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
2 $*$ *A* $*$ *card U*
⟨*proof*⟩

theorem *split-range-costed-capped-step-scan-insert-budget-from-seen-or-threshold*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *seen-factor*: $k * \Delta + t \leq 2 * A$
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *S-cap*: *card S* $\leq$ *cap*
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *seen-success*:
$B' = B \Longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
$2 * A *$ *card U*
⟨*proof*⟩

theorem *split-range-costed-capped-step-scan-insert-budget-from-scaled-seen-or-threshold*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*

and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *S-k-cap*: $k * \mathit{card}\ S \leq \mathit{cap}$
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *seen-success*:
$B' = B \Longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
$2 * A * \mathit{card}\ U$
⟨*proof*⟩

theorem *split-range-costed-bmssp-level-bound-from-seen-progress-budgets*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *seen-factor*: $k * \Delta + t \leq 2 * A$
and *source-factor*: $\mathit{Suc}\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\} \Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*)
and *run*:

split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *split-range-costed-bmssp-level-bound-from-scaled-seen-progress-budgets*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-scaled-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$k *$ *card S* $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*$)$
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*

and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-seen-progress-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *seen-factor*: $k * \Delta + t \leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*$)$
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
$c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-scaled-seen-progress-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of* $t\ h\ k$ *cap* $l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
card $S \leq$ *M-of* $l \Longrightarrow$
card $S \leq$ *card* U
and *step-scaled-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert* $d'\ a$ *betas* $bs\ B'$ *Us U-loop c-loop*
child-costs $U.$
$D =$ *label-partition-view*
(*find-pivots-label-capped* k *cap* $d\ S\ B$)
(*find-pivots-pivots-capped* k *cap* $d\ S\ B) \Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped* k *cap* $d\ S\ B) \Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of* $t\ h\ k$ *cap* l
(*find-pivots-label-capped* k *cap* $d\ S\ B$)
(*find-pivots-pivots-capped* k *cap* $d\ S\ B)\ B\ d'\ D\ a$ *betas* $bs\ B'$
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'.$ *find-pivots-label-capped* k *cap* $d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree* $S\ B'.$ *find-pivots-label-capped* k *cap* $d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$k *$ *card* $S \leq$ *cap* $\wedge$ *tree-antichain* $S\ \wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped* k *cap* $d\ S\ B) \leq$ *card* $U)$
and *run*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k$ *cap* l *finite-initial-label* $\{s\}$
Infinity d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-seen-progress-*

graph-bound:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *seen-factor*: $k * \Delta + t \leq 2 * A$
and *source-factor*: *Suc* $h \leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of* $t\ h\ k$ *cap* $l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
card $S \leq$ *M-of* $l \Longrightarrow$
card $S \leq$ *card* U
and *step-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert* $d'\ a$ *betas bs* B' *Us U-loop c-loop*
child-costs $U.$
$D =$ *label-partition-view*
(*find-pivots-label-capped* k *cap* $d\ S\ B$)
(*find-pivots-pivots-capped* k *cap* $d\ S\ B) \Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped* k *cap* $d\ S\ B) \Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of* $t\ h\ k$ *cap* l
(*find-pivots-label-capped* k *cap* $d\ S\ B$)
(*find-pivots-pivots-capped* k *cap* $d\ S\ B)\ B\ d'\ D\ a$ *betas bs* B'
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'.$ *find-pivots-label-capped* k *cap* $d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in$ *bound-tree* $S\ B'.$ *find-pivots-label-capped* k *cap* $d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
card $S \leq$ *cap* $\wedge$ *tree-antichain* $S \wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped* k *cap* $d\ S\ B) \leq$ *card* $U)$
and *run*:
split-range-costed-bmssp Δ *M-of* $t\ h\ k$ *cap* l *finite-initial-label* $\{s\}$
Infinity d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
⟨*proof*⟩

theorem *finite-initial-label-split-range-costed-top-level-correct-and-scaled-seen-progress-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ

and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-scaled-seen-progress*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$k *$ *card S* $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*)
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-cost-bound*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs
shows $c \leq$ *sum-list child-costs* $+$ (*M-of l*) $*$ *length child-costs* $+$
$t *$ (*sum-list*

(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
(*M-of l*) ∗ *length child-costs*)
⟨*proof*⟩

theorem *range-costed-partition-loop-state-cost-bound-by-child-sources*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
∃ *child-sources*.
length child-sources = *length bs* ∧
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
range-costed-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-sources (*range-tree-child-list P a bs*) ∧
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ (*sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
sum-list (*map card child-sources*))
and *range-costed-bmssp-child-sources-trivial*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c* ⟹ *True*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-cost-from-child-and-source-bounds*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*⟧
⟹ *card S-child* ≤ *card U-child*
shows *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
(*R* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
Suc t ∗ *card* (*range-tree-chain P a bs B′*)
⟨*proof*⟩

theorem *range-costed-partition-loop-state-cost-from-child-cost-bounds*:
assumes *run*:

range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child $\leq$ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
shows $c \leq$
$A * L *$ *card* (*range-tree-chain P a bs B'*) +
$(R + t) *$ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*)) +
(*Suc t*) $*$ (*M-of l*) $*$ *length bs*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-child-cost-bounds*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
shows *list-all2* (λ*c-child U-child*.
c-child $\leq$ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
⟨*proof*⟩

theorem *range-costed-partition-loop-state-cost-from-child-bound*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
shows $c \leq$
$A * L *$ *card* (*range-tree-chain P a bs B'*) +
$(R + t) *$ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*)) +
(*Suc t*) $*$ (*M-of l*) $*$ *length bs*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-trace*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*

betas bs B′ Us U c child-costs ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
concrete-partition-loop-trace P B a bs d′ B′ Us U
and *range-costed-bmssp-correct*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c* ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
bmssp-post-full d S B d′ B′ U
⟨*proof*⟩

theorem *range-costed-bmssp-source-card-le-if-label-below-output*:
assumes *run*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *complete*: *complete-on d S*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B′*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *range-costed-bmssp-source-card-le-if-sound-label-below-output*:
assumes *run*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B′*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-success-or-threshold*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
B′ = *B* ∨ *k* ∗ *cap* ≤ *card U*
and *range-costed-bmssp-success-or-threshold-trivial*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c* ⟹ *True*
⟨*proof*⟩

theorem *range-costed-bmssp-Suc-success-or-threshold*:
assumes *run*:
range-costed-bmssp Δ *M-of t k cap* (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*

and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $B' = B \vee k * cap \leq card\ U$
⟨*proof*⟩

theorem *range-costed-bmssp-Suc-source-card-le-from-complete-sources*:
assumes *run*:
range-costed-bmssp Δ *M-of t k cap* (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *complete*: *complete-on d S*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *S-cap*: *card S* ≤ *cap*
and *k-pos*: *0* < *k*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *range-costed-bmssp-Suc-source-card-le-from-label-below*:
assumes *run*:
range-costed-bmssp Δ *M-of t k cap* (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *S-cap*: *card S* ≤ *cap*
and *k-pos*: *0* < *k*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *range-costed-capped-step-threshold-if-not-success*:
assumes *loop*:
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *not-success*: $B' \neq B$
shows $k * cap \leq card\ U$
⟨*proof*⟩

theorem *range-costed-capped-step-scan-insert-budget-if-not-success*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*

and *insert-factor*: $t \leq A * k$
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v* = *dist s v*$\}$
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *S-cap*: *card S* $\leq$ *cap*
and *not-success*: $B' \neq B$
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
2 $*$ *A* $*$ *card U*
⟨*proof*⟩

theorem *range-costed-partition-loop-state-closes-level-bound-general*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *overhead*: (*Suc t*) $*$ (*M-of l*) $*$ *length bs* $\leq$ *A* $*$ *card U*
shows *c* $\leq$ *A* $*$ *Suc L* $*$ *card U* + (*R* + *t*) $*$ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *range-costed-partition-loop-state-closes-level-bound-from-source-progress-general*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-bound*:

$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
and *source-factor*: *Suc t* $\leq$ *A*
shows $c \leq A * Suc\ L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *range-costed-partition-loop-state-closes-level-bound-from-child-cost-bounds-general*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child $\leq$ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *overhead*: (*Suc t*) $*$ (*M-of l*) $*$ *length bs* $\leq A * card\ U$
shows $c \leq A * Suc\ L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *range-costed-partition-loop-state-closes-level-bound*:
assumes *run*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧

$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R l U-child*
and *overhead*: (*Suc t*) $*$ (*M-of l*) $*$ *length bs* $\leq$ *A* $*$ *card U*
shows *c* $\leq$ *A* $*$ *Suc l* $*$ *card U* + (*R* + *t*) $*$ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *range-costed-nonbase-step-closes-level-bound-general*:
assumes *loop*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U-loop c-loop child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge$*x*. *x* $\in$ *P* $\Longrightarrow$ *reachable s x*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x*. *x* $\in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *overhead*: (*Suc t*) $*$ (*M-of l*) $*$ *length bs* $\leq$ *A* $*$ *card U-loop*
and *U-def*: *U* = *U-loop* $\cup$ *W*
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* $\leq$ *A* $*$ *card U*
and *c-def*: *c* = *c-scan-insert* + *c-loop*
shows *c* $\leq$ *A* $*$ *Suc* (*Suc L*) $*$ *card U* +
(*R* + *t*) $*$ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *range-costed-nonbase-step-closes-level-bound-from-source-progress-general*:
assumes *loop*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U-loop c-loop child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge$*x*. *x* $\in$ *P* $\Longrightarrow$ *reachable s x*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x*. *x* $\in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
⟦*range-costed-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;

$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x;$
$card\ S\text{-}child \leq M\text{-}of\ l\rrbracket$
$\Longrightarrow card\ S\text{-}child \leq card\ U\text{-}child$
and *source-factor*: $Suc\ t \leq A$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: $finite\ U$
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$(R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *range-costed-nonbase-step-closes-level-bound-from-child-cost-bounds-general*:
assumes *loop*:
range-costed-partition-loop-state $\Delta\ M\text{-}of\ t\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
$betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs$
and *sound*: $sound\text{-}label\ d$
and *pre*: $bmssp\text{-}pre\text{-}full\ d\ P\ B$
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *child-cost-bounds*:
$list\text{-}all2\ (\lambda c\text{-}child\ U\text{-}child.$
$c\text{-}child \leq level\text{-}range\text{-}cost\text{-}bound\ A\ R\ L\ U\text{-}child)$
$child\text{-}costs\ (range\text{-}tree\text{-}child\text{-}list\ P\ a\ bs)$
and *overhead*: $(Suc\ t) * (M\text{-}of\ l) * length\ bs \leq A * card\ U\text{-}loop$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: $finite\ U$
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$(R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *range-costed-nonbase-step-closes-level-bound*:
assumes *loop*:
range-costed-partition-loop-state $\Delta\ M\text{-}of\ t\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
$betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs$
and *sound*: $sound\text{-}label\ d$
and *pre*: $bmssp\text{-}pre\text{-}full\ d\ P\ B$
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *child-bound*:
$\bigwedge c\text{-}child\ U\text{-}child\ S\text{-}child\ B\text{-}child\ d\text{-}child\ B\text{-}child'.$
$\llbracket range\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ k\ cap\ l\ d\ S\text{-}child\ B\text{-}child$
$d\text{-}child\ B\text{-}child'\ U\text{-}child\ c\text{-}child;$
$bmssp\text{-}pre\text{-}full\ d\ S\text{-}child\ B\text{-}child;$
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x;$
$card\ S\text{-}child \leq M\text{-}of\ l\rrbracket$
$\Longrightarrow c\text{-}child \leq level\text{-}range\text{-}cost\text{-}bound\ A\ R\ l\ U\text{-}child$
and *overhead*: $(Suc\ t) * (M\text{-}of\ l) * length\ bs \leq A * card\ U\text{-}loop$

and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite* U
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ l) * card\ U +$
$(R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *range-costed-base-level-bound*:
assumes *R-pos*: $0 < R$
shows *base-case-scan-cost* $\Delta\ k\ x\ B \leq$
level-range-cost-bound $A\ R\ (Suc\ 0)\ (base\text{-}case\text{-}vertices\ k\ x\ B)$
⟨*proof*⟩

The unsplit relation repeats the same recurrence in a form that is closer to the abstract cost interface. The base theorem $0 < ?R \Longrightarrow base\text{-}case\text{-}scan\text{-}cost\ ?\Delta\ ?k\ ?x\ ?B \leq level\text{-}range\text{-}cost\text{-}bound\ ?A\ ?R\ (Suc\ 0)\ (base\text{-}case\text{-}vertices\ ?k\ ?x\ ?B)$ pays for the bounded base scan with the outgoing-edge component of *level-range-cost-bound*. The main theorem below then performs induction on the recursion level: the induction hypothesis bounds each child call, while the supplied local-budget hypotheses pay for loop overhead and FindPivots overhead at the current level.

This theorem is intentionally parameterised by the constants A, R, t, and *M-of*. Later theories instantiate those parameters with the BMSSP schedule and with the bucketed partition costs, but the recurrence itself does not depend on how the partition primitive is implemented.

theorem *range-costed-bmssp-level-bound-from-local-budgets*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *loop-budget*:
$\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
range-costed-partition-loop-state $\Delta\ M\text{-}of\ t\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
$betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(Suc\ t) * (M\text{-}of\ l) * length\ bs \leq A * card\ U$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped* $k\ cap\ d\ S\ B$)
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $\Longrightarrow$
partition-initial-insert-cost-bound $c\text{-}insert\ t$
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $\Longrightarrow$
range-costed-partition-loop-state $\Delta\ M\text{-}of\ t\ k\ cap\ l$
(*find-pivots-label-capped* $k\ cap\ d\ S\ B$)
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $B\ d'\ D\ a\ betas\ bs\ B'$

Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v =$ *dist* $s\ v\}$ $\Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v =$ *dist* $s\ v\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full $d\ S\ B$ $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x)$ $\Longrightarrow$
fp-iter-capped-scan-cost $k\ cap\ d\ S\ S\ B + c\text{-}insert \leq A * card\ U$
and *run*:
range-costed-bmssp Δ *M-of* $t\ k\ cap\ l\ d\ S\ B\ d'\ B'\ U\ c$
and *sound*: *sound-label* d
and *pre*: *bmssp-pre-full* $d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x$
shows $c \leq$ *level-range-cost-bound* $A\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *range-costed-bmssp-level-bound-from-source-progress-and-local-budgets*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: *Suc* $t \leq A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
range-costed-bmssp Δ *M-of* $t\ k\ cap\ l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow$
bmssp-pre-full $d\ S\ B$ $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x)$ $\Longrightarrow$
card $S \leq$ *M-of* l $\Longrightarrow$
card $S \leq$ *card* U
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped* $k\ cap\ d\ S\ B$)
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $\Longrightarrow$
partition-initial-insert-cost-bound $c\text{-}insert\ t$
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of* $t\ k\ cap\ l$
(*find-pivots-label-capped* $k\ cap\ d\ S\ B$)
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $B\ d'\ D\ a\ betas\ bs\ B'$
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v =$ *dist* $s\ v\}$ $\Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v =$ *dist* $s\ v\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full $d\ S\ B$ $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x)$ $\Longrightarrow$
fp-iter-capped-scan-cost $k\ cap\ d\ S\ S\ B + c\text{-}insert \leq A * card\ U$
and *run*:

range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $c \leq$ *level-range-cost-bound A* $(R + l * t)$ $(2 * l + 1)$ *U*
⟨*proof*⟩

theorem *range-costed-bmssp-level-bound-from-source-progress-budgets*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: *Suc t* $\leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d′ a betas bs B′ Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d′
$\{v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\} \Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ $k * cap \leq$ *card U*
and *run*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $c \leq$ *level-range-cost-bound* $(2 * A)$ $(R + l * t)$ $(2 * l + 1)$ *U*
⟨*proof*⟩

theorem *range-costed-bmssp-level-bound-from-progress-budgets*:

assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *loop-factor*: $Suc\ t \leq 2 * A$
and *loop-progress*:
$\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a betas bs B' Us U c child-costs* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(M\text{-}of\ l) * length\ bs \leq card\ U$
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B' Us U-loop c-loop child-costs* $\Longrightarrow$
complete-on d'
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$card\ S \leq cap \wedge k * cap \leq card\ U$
and *run*:
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x$
shows $c \leq level\text{-}range\text{-}cost\text{-}bound\ (2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-range-costed-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow reachable\ s\ v$
and *run*:
range-costed-bmssp Δ *M-of t k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'*
⟨*proof*⟩

theorem *finite-initial-label-range-costed-top-level-correct-and-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *loop-factor*: *Suc* $t \leq 2 * A$
and *loop-progress*:
$\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
range-costed-partition-loop-state Δ *M-of* $t\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
betas bs B' *Us* $U\ c$ *child-costs* $\Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
(*M-of* l) $*$ *length bs* $\leq$ *card* U
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert* $d'\ a\ betas\ bs\ B'\ Us$ *U-loop c-loop*
child-costs $U.$
$D =$ *label-partition-view*
(*find-pivots-label-capped* $k\ cap\ d\ S\ B$)
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B) \Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B) \Longrightarrow$
range-costed-partition-loop-state Δ *M-of* $t\ k\ cap\ l$
(*find-pivots-label-capped* $k\ cap\ d\ S\ B$)
(*find-pivots-pivots-capped* $k\ cap\ d\ S\ B$) $B\ d'\ D\ a\ betas\ bs\ B'$
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* $k\ cap\ d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
card $S \leq cap \wedge k * cap \leq$ *card* U
and *run*:
range-costed-bmssp Δ *M-of* $t\ k\ cap\ l$ *finite-initial-label* $\{s\}$
Infinity d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-range-costed-top-level-correct-and-source-progress-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$

and *source-factor*: $Suc\ t \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ $k * cap \leq$ *card U*
and *run*:
range-costed-bmssp Δ *M-of t k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
$c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-range-costed-top-level-correct-and-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: $Suc\ t \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*

and *step-progress*:
⋀*l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) ⟹
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) ⟹
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs ⟹
complete-on d′
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
U = *U-loop* ∪
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
card S ≤ *cap* ∧ *k* ∗ *cap* ≤ *card U*
and *run*:
range-costed-bmssp Δ *M-of t k cap l finite-initial-label* {*s*}
Infinity d′ Infinity U c
shows *sssp-correct d′* ∧
c ≤ 2 ∗ *A* ∗ (2 ∗ *l* + 1) ∗ *vertex-count* + (*R* + *l* ∗ *t*) ∗ *edge-count*
⟨*proof*⟩

end

end
theory *BMSSP-Exact-Range-Costed*
imports *BMSSP-Range-Costed*
begin

31 Exact Split Range-Costed Runs

The range-costed relation records the algorithm's range accounting, but leaves the pulled child source abstract. The exact variant below keeps the same cost and range information while additionally recording that each pull source is exactly the lower label split used by the operational algorithm. It refines the existing range-costed relation, so the established correctness and cost lemmas remain reusable.

This extra equality is small syntactically and important analytically. A pull operation returns a child source set and a child bound. For correctness it is enough to know that the child call satisfies the BMSSP precondition. For progress and runtime, however, the source set must be identified with the mathematical split of the parent sources below the child bound. Only

then can the proof inherit the antichain and scaled-cardinality facts that were proved for lower splits.

The theory therefore sits between range synchronisation and direct insertion. It keeps the range slices, child-cost list, and edge/source batch split from the preceding layer, while adding enough exactness to discharge the source progress side conditions needed by the recurrence. Later theories refine this one by splitting edge batches further; they rely on the exact source information introduced here rather than reproving it.

No new algorithm is introduced. Each theorem either forgets the exactness to recover the split-range relation, or uses exactness to derive stronger cost premises for the same loop trace.

context *unique-shortest-digraph*
begin

inductive *exact-split-range-costed-partition-loop-state*
and *exact-split-range-costed-bmssp* **where**
Exact-Split-Range-State-Done:
keys-of D = {} ⟹
bound-le (Fin a) B ⟹
complete-on d′ (bound-tree P B) ⟹
exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
[] [] B [range-tree P a B]
(bound-tree P (Fin a) ∪ ⋃(set [range-tree P a B])) 0 []
| *Exact-Split-Range-State-Stop*:
bound-le (Fin a) B ⟹
complete-on d′ (bound-tree P (Fin a)) ⟹
k ∗ cap ≤ card (bound-tree P (Fin a)) ⟹
exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
[] [] (Fin a) [range-tree P a (Fin a)]
(bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0 []
| *Exact-Split-Range-State-Step*:
pull-separates D (M-of l) Bmax S-pull beta D-pull ⟹
bound-le (Fin beta) B ⟹
bmssp-pre-full d S-pull (Fin beta) ⟹
S-pull = split-below d P beta ⟹
(∀ x∈S-pull. reachable s x) ⟹
a ≤ b ⟹
bound-le (Fin a) B′ ⟹
complete-on d′ (bound-tree P (Fin a)) ⟹
card (bound-tree P (Fin a)) < k ∗ cap ⟹
exact-split-range-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
d-child (Fin b) U-child c-child ⟹
U-child = range-tree P a (Fin b) ⟹
complete-preserved d-child d′ U-child ⟹
edge-batch = edge-relaxation-pairs-between d-child U-child b beta ⟹
source-batch = label-pairs-between d S-pull b beta ⟹
batch = edge-batch @ source-batch ⟹
D-next = batch-min-update D-pull batch ⟹

partition-pull-cost-bound c-pull S-pull $\Longrightarrow$
partition-batch-cost-bound c-edges t edge-batch $\Longrightarrow$
partition-batch-cost-bound c-sources h source-batch $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′*
D-next b betas bs B′ Us-tail U-tail c-tail child-costs-tail $\Longrightarrow$
$c = c\text{-}pull + c\text{-}edges + c\text{-}sources + c\text{-}child + c\text{-}tail \Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
(*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) $\cup$
$\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c*
(*c-child* # *child-costs-tail*)

| *Exact-Split-Range-Cost-Base*:
$S = \{x\} \Longrightarrow$
exact-split-range-costed-bmssp Δ *M-of t h k cap 0 d S B*
(λv. *if* $v \in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
(*base-case-scan-cost* Δ *k x B*)

| *Exact-Split-Range-Cost-Step*:
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs-loop $\Longrightarrow$
complete-on d′
$\{v \in$ *bound-tree S B′. find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B′. find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
$c = fp\text{-}iter\text{-}capped\text{-}scan\text{-}cost\ k\ cap\ d\ S\ S\ B + c\text{-}insert + c\text{-}loop \Longrightarrow$
exact-split-range-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*

The defining difference from the split-range relation is the premise *S-pull* = *split-below d P beta* in the loop step. This says that the implementation-level pull has not merely produced an admissible child set; it has produced exactly the semantic lower split of the parent problem. The rest of the constructor mirrors the earlier relation: edge and source batches are charged separately, the child output is the next *range-tree*, and the total cost is the sum of local and recursive costs.

The equality to *split-below* is deliberately stored at the point where the pull happens. Downstream proofs then extract it from the derivation tree rather than reconstructing it from the partition invariant. This keeps the cost proofs focused on summing costs and ranges.

inductive-cases *exact-split-range-costed-bmssp-zeroE*:

exact-split-range-costed-bmssp Δ *M-of t h k cap 0 d S B d′ B′ U c*

inductive-cases *exact-split-range-costed-bmssp-SucE*:
exact-split-range-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*

theorem *exact-split-range-costed-refines-split-range-costed*:
shows
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
⟨*proof*⟩

theorem *exact-split-range-costed-bmssp-correct*:
assumes *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′*
⟨*proof*⟩

The refinement theorem *exact-split-range-costed-partition-loop-state ?*Δ *?M-of ?t ?h ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs* $\Longrightarrow$ *split-range-costed-partition-loop-state ?*Δ *?M-of ?t ?h ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs* is the compatibility layer: forgetting the exact source equality gives a valid split-range run. Correctness and top-level SSSP correctness are then inherited immediately from the previous theory.

The remaining results use the exact source equality in the other direction. They expose the child calls together with the lower-split facts needed for the induction hypothesis: bounded cardinality, reachability, and labels below the child bound. Those facts are the source-progress side of the runtime proof.

theorem *exact-split-range-costed-partition-loop-state-child-cost-calls-below*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
length child-costs = *length bs* $\wedge$

list-all2
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-costs (*range-tree-child-list P a bs*)
and *exact-split-range-costed-bmssp-child-cost-calls-below-trivial*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹ *True*
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-child-cost-bounds*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*
shows *list-all2* (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-child-cost-bounds-with-invariants*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
P ⊆ *V* ⟹
k ∗ *card P* ≤ *cap* ⟹
tree-antichain P ⟹
(⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*;
k ∗ *card S-child* ≤ *cap*;
tree-antichain S-child⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*) ⟹
list-all2 (λ*c-child U-child*.

c-child $\leq$ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *exact-split-range-costed-bmssp-child-cost-bounds-with-invariants-trivial*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

The child-source extraction theorem packages one loop trace as a list of source sets aligned with *range-tree-child-list*. Alignment matters: each source set is charged against the same range slice as its recursive child call. Once this list exists, subsequent cost theorems can use ordinary list-wise reasoning instead of induction over the full operational relation.

theorem *exact-split-range-costed-partition-loop-state-child-sources-below*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
$\exists$ *child-sources.*
length child-sources = *length bs* $\wedge$
list-all2
(*λS-child U-child.* $\exists$ *c-child B-child d-child B-child′.*
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child. reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child. below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*)
and *exact-split-range-costed-bmssp-child-sources-below-trivial*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

The next cost bounds combine the extracted child sources with the explicit edge and source batch costs stored in the relation. First the local loop cost is bounded by the sum of child costs, source cardinalities, and outgoing-edge charges over the child ranges. Then the range-chain lemmas replace those list sums by the corresponding counts for the parent loop output.

theorem *exact-split-range-costed-partition-loop-state-cost-bound-by-child-sources-below*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
$\exists$ *child-sources.*
length child-sources = *length bs* $\wedge$
list-all2
(*λS-child U-child.* $\exists$ *c-child B-child d-child B-child′.*
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child. reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child. below-bound* (*d x*) *B-child*))

child-sources (*range-tree-child-list P a bs*) ∧
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
h ∗ *sum-list* (*map card child-sources*)
and *exact-split-range-costed-bmssp-cost-child-sources-below-trivial*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹ *True*
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-cost-bound-by-child-sources-and-edge-ranges*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
sound-label d ⟹
∃ *child-sources*.
length child-sources = *length bs* ∧
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*) ∧
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
h ∗ *sum-list* (*map card child-sources*)
and *exact-split-range-costed-bmssp-cost-child-sources-and-edge-ranges-trivial*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹ *True*
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-cost-from-child-source-and-edge-range-bounds*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;

card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*$\rrbracket$
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
shows *c* $\leq$
A $*$ *L* $*$ *card* (*range-tree-chain P a bs B'*) $+$
R $*$ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*)) $+$
t $*$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) $+$
(*Suc h*) $*$ *card* (*range-tree-chain P a bs B'*)
⟨*proof*⟩

corollary *exact-split-range-costed-partition-loop-state-cost-from-child-source-and-edge-range-graph-bound*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *sound*: *sound-label d*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child $\leq$ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child' U-child c-child*.
$\llbracket$*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*$\rrbracket$
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
shows *c* $\leq$
A $*$ *L* $*$ *card* (*range-tree-chain P a bs B'*) $+$
R $*$ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*)) $+$
t $*$ *edge-count* $+$
(*Suc h*) $*$ *card* (*range-tree-chain P a bs B'*)
⟨*proof*⟩

The final theorem in this theory is the reusable recurrence step for exact split ranges. Given child costs already bounded by *level-range-cost-bound* and a source-progress hypothesis, it proves the parent-loop bound with one additional level of vertex cost and one additional batch coefficient on outgoing edges. This is the shape consumed by the direct-insert and bucketed refinements.

theorem *exact-split-range-costed-partition-loop-state-cost-from-child-and-source-bounds*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *child-cost-bounds*:

```
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
    and source-progress:
      ⋀S-child B-child d-child B-child′ U-child c-child.
        ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child′ U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l;
          ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
        ⟹ card S-child ≤ card U-child
  shows c ≤
    A * L * card (range-tree-chain P a bs B′) +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B′)) +
    (Suc h) * card (range-tree-chain P a bs B′)
⟨proof⟩

end

end
theory BMSSP-Direct-Insert-Costed
  imports BMSSP-Exact-Range-Costed
begin
```

32 Direct-Insert Costed Runs

This theory refines the range-costed BMSSP run with the final accounting distinction needed by the bucketed partition implementation. The earlier split-range relation separates edge-generated batch entries from source-label batch entries. Here the edge-generated entries are split again. Some relaxed edge values belong directly in the current parent range and are paid for by the ordinary insert cost parameter t. Other relaxed values fall below the next child bound and are prepended into the lower block; those are paid for by the batch parameter h.

The distinction mirrors the bucketed data structure. A direct insert is the operation that searches for a bucket position and may trigger the logarithmic bucket-splitting work. A lower-range prepend is cheaper in the abstract accounting because it is grouped with other items destined for the next pull. Recording the two lists separately lets the later amortised theorem keep the direct-insert term visible instead of hiding it inside one combined batch budget.

The semantic content is intentionally unchanged. The inductive run still produces the same completed range slices as the exact range-costed relation, and the same BMSSP postcondition is recovered by forgetting the extra accounting fields. What changes is the cost expression: each loop step now

exposes a direct-edge batch, a lower-edge batch, a source batch, the child recursive cost, and the tail-loop cost.

This is the last local-cost layer before the global recurrence is instantiated. The closing theorems near the end of the file package the separated charges into two forms: an amortised bound that keeps direct range inserts explicit, and a level bound that can feed the headline top-level analysis.

context *unique-shortest-digraph*
begin

inductive *direct-insert-costed-partition-loop-state*
  and *direct-insert-costed-bmssp* **where**
  Direct-Insert-Costed-State-Done:
    keys-of D = {} ⟹
    bound-le (Fin a) B ⟹
    complete-on d′ (bound-tree P B) ⟹
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      [] [] B [range-tree P a B]
      (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a B])) 0 []
| *Direct-Insert-Costed-State-Stop*:
    bound-le (Fin a) B ⟹
    complete-on d′ (bound-tree P (Fin a)) ⟹
    k ∗ cap ≤ card (bound-tree P (Fin a)) ⟹
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      [] [] (Fin a) [range-tree P a (Fin a)]
      (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0 []
| *Direct-Insert-Costed-State-Step*:
    pull-separates D (M-of l) Bmax S-pull beta D-pull ⟹
    bound-le (Fin beta) B ⟹
    bmssp-pre-full d S-pull (Fin beta) ⟹
    S-pull = split-below d P beta ⟹
    (∀ x∈S-pull. reachable s x) ⟹
    a ≤ b ⟹
    bound-le (Fin a) B′ ⟹
    complete-on d′ (bound-tree P (Fin a)) ⟹
    card (bound-tree P (Fin a)) < k ∗ cap ⟹
    direct-insert-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
      d-child (Fin b) U-child c-child ⟹
    U-child = range-tree P a (Fin b) ⟹
    complete-preserved d-child d′ U-child ⟹
    direct-edge-batch = edge-relaxation-pairs-in-bound d-child U-child beta B ⟹
    lower-edge-batch = edge-relaxation-pairs-between d-child U-child b beta ⟹
    source-batch = label-pairs-between d S-pull b beta ⟹
    batch = direct-edge-batch @ lower-edge-batch @ source-batch ⟹
    D-next = batch-min-update D-pull batch ⟹
    partition-pull-cost-bound c-pull S-pull ⟹
    partition-batch-cost-bound c-direct t direct-edge-batch ⟹
    partition-batch-cost-bound c-lower h lower-edge-batch ⟹
    partition-batch-cost-bound c-sources h source-batch ⟹

```
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d'
      D-next b betas bs B' Us-tail U-tail c-tail child-costs-tail ⟹
    c = c-pull + c-direct + c-lower + c-sources + c-child + c-tail ⟹
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      (beta # betas) (b # bs) B'
      (range-tree P a (Fin b) # Us-tail)
      (bound-tree P (Fin a) ∪
       ⋃(set (range-tree P a (Fin b) # Us-tail))) c
      (c-child # child-costs-tail)
| Direct-Insert-Costed-Base:
    S = {x} ⟹
    direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B
      (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
      (base-case-bound k x B)
      (base-case-vertices k x B)
      (base-case-scan-cost Δ k x B)
| Direct-Insert-Costed-Step:
    D = label-partition-view
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) ⟹
    partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B) ⟹
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs-loop ⟹
    complete-on d'
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    U = U-loop ∪
     {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
     direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
```

The constructors are the same operational skeleton as before, but the step constructor now names three batches. *edge-relaxation-pairs-in-bound* extracts direct edge relaxations in the parent range, *edge-relaxation-pairs-between* extracts lower edge relaxations between the child output bound and the pull bound, and *label-pairs-between* extracts old source labels in the same lower interval.

Keeping these lists in the derivation tree is useful because each has a different combinatorial owner. Direct edge entries are counted by the direct range lists; lower edge entries are counted by child edge-range lists; source entries are counted through child source sets and then by the progress lemma. The proof scripts that follow mostly say that this extra bookkeeping is conservative: it refines the earlier correctness trace without changing which vertices are completed.

```
inductive-cases direct-insert-costed-bmssp-zeroE:
  direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
```

inductive-cases *direct-insert-costed-bmssp-SucE*:
direct-insert-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*

lemma *direct-insert-costed-partition-loop-state-mono*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
shows *nondecreasing-from a bs*
⟨*proof*⟩

lemma *direct-insert-costed-partition-loop-state-beta-bounds*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
shows *bounds-le B betas*
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-trace*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
($\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*) $\Longrightarrow$
concrete-partition-loop-trace P B a bs d′ B′ Us U
and *direct-insert-costed-bmssp-correct*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*) $\Longrightarrow$
bmssp-post-full d S B d′ B′ U
⟨*proof*⟩

theorem *finite-initial-label-direct-insert-costed-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′*
⟨*proof*⟩

theorem *finite-initial-label-direct-insert-costed-top-level-reachable-correct*:
assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′*
⟨*proof*⟩

The correctness theorem ⟦*direct-insert-costed-bmssp ?*Δ *?M-of ?t ?h ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U ?c*; *sound-label ?d*; *bmssp-pre-full ?d ?S ?B*;

$\bigwedge x.\ x \in ?S \Longrightarrow \mathit{reachable}\ s\ x]\!] \Longrightarrow \mathit{bmssp\text{-}post\text{-}full}\ ?d\ ?S\ ?B\ ?d'\ ?B'\ ?U$ is the semantic projection of the direct-insert relation. Once projected, the top-level corollaries are the familiar root BMSSP statements: from the finite initial label at the source and the infinite output bound, the returned label function is a complete SSSP solution.

The next group of lemmas supplies the progress side conditions used by the cost recurrence. They show that successful BMSSP calls either return the original bound or have accumulated enough completed vertices to meet the threshold, and that a source set whose labels are below the output bound is dominated by the output tree. These facts turn source-batch costs into vertex-count terms.

theorem *direct-insert-costed-bmssp-source-card-le-if-label-below-output*:
  assumes *run*:
      $\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c$
    and *sound*: $\mathit{sound\text{-}label}\ d$
    and *pre*: $\mathit{bmssp\text{-}pre\text{-}full}\ d\ S\ B$
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x$
    and *complete*: $\mathit{complete\text{-}on}\ d\ S$
    and *below*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{below\text{-}bound}\ (d\ x)\ B'$
  shows $\mathit{card}\ S \leq \mathit{card}\ U$
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-source-card-le-if-sound-label-below-output*:
  assumes *run*:
      $\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c$
    and *sound*: $\mathit{sound\text{-}label}\ d$
    and *pre*: $\mathit{bmssp\text{-}pre\text{-}full}\ d\ S\ B$
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x$
    and *below*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{below\text{-}bound}\ (d\ x)\ B'$
  shows $\mathit{card}\ S \leq \mathit{card}\ U$
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-success-or-threshold*:
  $\mathit{direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ P\ B\ d'\ D\ a$
      $\mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ U\ c\ \mathit{child\text{-}costs} \Longrightarrow$
   $\mathit{sound\text{-}label}\ d \Longrightarrow$
   $\mathit{bmssp\text{-}pre\text{-}full}\ d\ P\ B \Longrightarrow$
   $(\bigwedge x.\ x \in P \Longrightarrow \mathit{reachable}\ s\ x) \Longrightarrow$
   $B' = B \vee k * \mathit{cap} \leq \mathit{card}\ U$
and *direct-insert-costed-bmssp-success-or-threshold-trivial*:
  $\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow \mathit{True}$
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-Suc-success-or-threshold*:
  assumes *run*:
      $\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ (\mathit{Suc}\ l)\ d\ S\ B\ d'\ B'\ U\ c$
    and *sound*: $\mathit{sound\text{-}label}\ d$
    and *pre*: $\mathit{bmssp\text{-}pre\text{-}full}\ d\ S\ B$

and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x$
shows $B' = B \vee k * \mathit{cap} \leq \mathit{card}\ U$
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-Suc-source-card-le-from-label-below*:
assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{below\text{-}bound}\ (d\ x)\ B$
and *S-cap*: *card S* $\leq$ *cap*
and *k-pos*: $0 < k$
shows *card S* $\leq$ *card U*
⟨*proof*⟩

The child-cost lemmas extract the recursive calls hidden inside a partition loop. Each child call is paired with the range slice it completed, together with the exact source set pulled for that child and the proof obligations needed to apply the induction hypothesis. The additional *below-bound* $(d\ x)$ *B-child* invariant is what makes source progress available without re-deriving the pull/split equality at every later use.

theorem *direct-insert-costed-partition-loop-state-child-cost-calls-below*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
length child-costs = *length bs* $\wedge$
list-all2
(λ*c-child U-child*. $\exists$ *S-child B-child d-child B-child′*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*))
child-costs (*range-tree-child-list P a bs*)
and *direct-insert-costed-bmssp-child-cost-calls-below-trivial*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-child-cost-calls-below-with-bounds*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
list-all2
(λ*c-child UB*. *case UB of* (*U-child*, *B-child*) $\Rightarrow$
$\exists$ *S-child d-child B-child′*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$

card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child. below-bound (d x) B-child))*
child-costs (range-tree-child-bound-pair-list P a betas bs)
and *direct-insert-costed-bmssp-child-cost-calls-below-with-bounds-trivial*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-child-cost-bounds*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′.*
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x. x* $\in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge$*x. x* $\in$ *S-child* $\Longrightarrow$ *below-bound (d x) B-child*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
shows *list-all2* (*λc-child U-child.*
c-child $\leq$ *level-range-cost-bound A R L U-child)*
child-costs (range-tree-child-list P a bs)
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-child-cost-bounds-with-invariants*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
P $\subseteq$ *V* $\Longrightarrow$
k $*$ *card P* $\leq$ *cap* $\Longrightarrow$
tree-antichain P $\Longrightarrow$
(*(*$\bigwedge$*c-child U-child S-child B-child d-child B-child′.*
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge$*x. x* $\in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge$*x. x* $\in$ *S-child* $\Longrightarrow$ *below-bound (d x) B-child*;
k $*$ *card S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child)* $\Longrightarrow$
list-all2 (*λc-child U-child.*
c-child $\leq$ *level-range-cost-bound A R L U-child)*
child-costs (range-tree-child-list P a bs)
and *direct-insert-costed-bmssp-child-cost-bounds-with-invariants-trivial*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-child-amortized-cost-bounds-with-*

invariants:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
$P \subseteq V \Longrightarrow$
$k * card\ P \leq cap \Longrightarrow$
tree-antichain $P \Longrightarrow$
$(\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
$[\![$*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* $(d\ x)$ *B-child*;
$k * card$ *S-child* $\leq$ *cap*;
tree-antichain S-child$]\!]$
$\Longrightarrow$ *c-child* $\leq$
bmssp-amortized-cost-bound A R h t q L B-child U-child) $\Longrightarrow$
list-all2 $(\lambda$*c-child UB*. *case UB of* (*U-child*, *B-child*) $\Rightarrow$
c-child $\leq$ *bmssp-amortized-cost-bound A R h t q L B-child U-child*)
child-costs (*range-tree-child-bound-pair-list P a betas bs*)
and *direct-insert-costed-bmssp-child-amortized-cost-bounds-with-invariants-trivial*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
⟨*proof*⟩

The central loop cost bound separates the four local charges that survive after recursive child costs have been bounded: pulled child sources, direct edge-range inserts, lower edge-range batches, and source batches. The list functions over *range-tree-child-list* and its edge-range variants are the combinatorial map from one loop trace to disjoint graph regions. Their disjointness lemmas are what prevent the loop from paying for the same edge range repeatedly.

theorem *direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-and-edge-ranges*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
$\exists$ *child-sources*.
length child-sources = *length bs* $\wedge$
list-all2
$(\lambda$*S-child U-child*. $\exists$ *c-child B-child d-child B-child′*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
$(\forall\, x{\in}$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
$(\forall\, x{\in}$*S-child*. *below-bound* $(d\ x)$ *B-child*))
child-sources (*range-tree-child-list P a bs*) $\wedge$
$c \leq$ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +

$h * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ P\ a\ \mathit{betas}\ \mathit{bs})) +$
$h * \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{card}\ \mathit{child\text{-}sources})$
and *direct-insert-costed-bmssp-cost-child-sources-and-edge-ranges-trivial*:
$\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c \Longrightarrow \mathit{True}$
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-cost-from-child-source-and-edge-range-bounds*:
assumes *run*:
$\mathit{direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ P\ B\ d'\ D\ a$
$\mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ U\ c\ \mathit{child\text{-}costs}$
and *sound*: $\mathit{sound\text{-}label}\ d$
and *child-cost-bounds*:
$\mathit{list\text{-}all2}\ (\lambda \mathit{c\text{-}child}\ \mathit{U\text{-}child}.$
$\mathit{c\text{-}child} \leq \mathit{level\text{-}range\text{-}cost\text{-}bound}\ A\ R\ L\ \mathit{U\text{-}child})$
$\mathit{child\text{-}costs}\ (\mathit{range\text{-}tree\text{-}child\text{-}list}\ P\ a\ \mathit{bs})$
and *source-progress*:
$\bigwedge \mathit{S\text{-}child}\ \mathit{B\text{-}child}\ \mathit{d\text{-}child}\ \mathit{B\text{-}child'}\ \mathit{U\text{-}child}\ \mathit{c\text{-}child}.$
$[\![\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ \mathit{S\text{-}child}\ \mathit{B\text{-}child}$
$\mathit{d\text{-}child}\ \mathit{B\text{-}child'}\ \mathit{U\text{-}child}\ \mathit{c\text{-}child};$
$\mathit{bmssp\text{-}pre\text{-}full}\ d\ \mathit{S\text{-}child}\ \mathit{B\text{-}child};$
$\bigwedge x.\ x \in \mathit{S\text{-}child} \Longrightarrow \mathit{reachable}\ s\ x;$
$\mathit{card}\ \mathit{S\text{-}child} \leq \mathit{M\text{-}of}\ l;$
$\bigwedge x.\ x \in \mathit{S\text{-}child} \Longrightarrow \mathit{below\text{-}bound}\ (d\ x)\ \mathit{B\text{-}child}]\!]$
$\Longrightarrow \mathit{card}\ \mathit{S\text{-}child} \leq \mathit{card}\ \mathit{U\text{-}child}$
shows $c \leq$
$A * L * \mathit{card}\ (\mathit{range\text{-}tree\text{-}chain}\ P\ a\ \mathit{bs}\ B') +$
$R * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ P\ a\ \mathit{bs}\ B')) +$
$t * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}direct\text{-}edge\text{-}range\text{-}list}\ P\ B\ a\ \mathit{betas}\ \mathit{bs})) +$
$h * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ P\ a\ \mathit{betas}\ \mathit{bs})) +$
$(\mathit{Suc}\ h) * \mathit{card}\ (\mathit{range\text{-}tree\text{-}chain}\ P\ a\ \mathit{bs}\ B')$
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-cost-from-child-source-and-amortized-bounds*:
assumes *run*:
$\mathit{direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ P\ B\ d'\ D\ a$
$\mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ U\ c\ \mathit{child\text{-}costs}$
and *sound*: $\mathit{sound\text{-}label}\ d$
and *child-cost-bounds*:
$\mathit{list\text{-}all2}\ (\lambda \mathit{c\text{-}child}\ \mathit{UB}.\ \mathit{case}\ \mathit{UB}\ \mathit{of}\ (\mathit{U\text{-}child},\ \mathit{B\text{-}child}) \Rightarrow$
$\mathit{c\text{-}child} \leq \mathit{bmssp\text{-}amortized\text{-}cost\text{-}bound}\ A\ R\ h\ t\ q\ L\ \mathit{B\text{-}child}\ \mathit{U\text{-}child})$
$\mathit{child\text{-}costs}\ (\mathit{range\text{-}tree\text{-}child\text{-}bound\text{-}pair\text{-}list}\ P\ a\ \mathit{betas}\ \mathit{bs})$
and *source-progress*:
$\bigwedge \mathit{S\text{-}child}\ \mathit{B\text{-}child}\ \mathit{d\text{-}child}\ \mathit{B\text{-}child'}\ \mathit{U\text{-}child}\ \mathit{c\text{-}child}.$
$[\![\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ \mathit{S\text{-}child}\ \mathit{B\text{-}child}$

d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*$\rrbracket$
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
shows $c \leq$
$A * L *$ *card* (*range-tree-chain P a bs B′*) +
$(R + q * h) *$ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
$t *$ *sum-list*
(*map card* (*range-tree-child-zero-edge-range-list P a betas bs*)) +
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h *$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
(*Suc h*) $*$ *card* (*range-tree-chain P a bs B′*)
⟨*proof*⟩

The amortised closing theorem below keeps the direct-insert range term visible because the bucketed implementation has a sharper accounting story for it. The child calls contribute a lower-level amortised bound, the source-progress hypothesis absorbs the pulled source sets into completed child ranges, and the edge-range disjointness lemmas collapse the direct and lower edge lists into the outgoing-edge terms of the parent output.

theorem *direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-cost-bounds*:
list-all2 (λ*c-child UB. case UB of* (*U-child*, *B-child*) $\Rightarrow$
c-child $\leq$ *bmssp-amortized-cost-bound A R h t q L B-child U-child*)
child-costs (*range-tree-child-bound-pair-list P a betas bs*)
and *source-factor*: *Suc h* $\leq$ *A*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child.*
$\llbracket$*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*$\rrbracket$
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
shows $c \leq$ *bmssp-amortized-cost-bound A R h t* (*Suc q*) (*Suc L*) *B U*
⟨*proof*⟩

The final level-bound theorem is the non-amortised companion used by the abstract headline recurrence. It closes the same loop skeleton, but leaves the direct and lower edge-range sums explicit rather than folding them into *bmssp-amortized-cost-bound*. This gives later theories the freedom to plug in either an abstract partition budget or the bucketed amortised budget without changing the semantic development.

theorem *direct-insert-costed-partition-loop-state-closes-level-bound-from-child-costs-and-edge-ranges*:
  assumes *run*:
    direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
      betas bs B′ Us U c child-costs
    and *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d P B*
    and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
    and *child-cost-bounds*:
      list-all2 (λ*c-child U-child*.
        c-child $\leq$ *level-range-cost-bound A R L U-child*)
        child-costs (*range-tree-child-list P a bs*)
    and *source-factor*: *Suc h* $\leq$ *A*
    and *source-progress*:
      $\bigwedge$*S-child B-child d-child B-child′ U-child c-child*.
        ⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
            d-child B-child′ U-child c-child;
          bmssp-pre-full d S-child B-child;
          $\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
          card S-child $\leq$ *M-of l*;
          $\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*⟧
        $\Longrightarrow$ *card S-child* $\leq$ *card U-child*
  shows *c* $\leq$
    A $*$ *Suc L* $*$ *card U* + *R* $*$ *card* (*outgoing-edges U*) +
    t $*$ *sum-list*
      (*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
    h $*$ *sum-list*
      (*map card* (*range-tree-child-edge-range-list P a betas bs*))
⟨*proof*⟩

end

end
theory *BMSSP-Strict-Tie-Breaking*
  imports *BMSSP-Direct-Insert-Costed*
begin

33 Strict Tie-Breaking Consequences

Lexicographic tie-breaking removes distance ties by comparing extended path values. The existing development abstracts uniqueness of shortest paths,

but source progress for the base case also needs strict growth along the selected shortest-path tree. This file records exactly that consequence, so the remaining complexity proof can be connected either to a formal extended-distance construction or to any graph model that already provides the same strict tree order.

The cost recurrence uses progress statements of the form "a source set is no larger than the output range that completed it". For non-base recursive calls this comes from the BMSSP threshold: either the call succeeds with the old bound, or it completes enough vertices. The base case is subtler. It takes the first $k + 1$ vertices in distance order from a singleton source. To know that the singleton source itself is included, the proof needs the source to be the strict first element of that order.

The strict tie-breaking locale supplies this missing monotonicity. Along the chosen shortest-path tree, descendants have strictly larger distance than ancestors. Consequently, in the singleton bound tree rooted at x, the root x is the unique minimum-distance vertex. This is the formal counterpart of the paper's assumption that ties can be perturbed away without changing the combinatorial structure of the algorithm.

The later half of the theory threads this progress fact through the exact range-costed and direct-insert costed relations. It closes the source- progress side condition required by the local-budget theorems, then packages the resulting level and amortised graph bounds for the top-level run.

context *finite-weighted-digraph*
begin

lemma *walk-weight-pos-if-positive-edges*:
  assumes *positive*: $\bigwedge u\ v.\ (u,\ v) \in E \Longrightarrow 0 < w\ u\ v$
    and *walk-p*: *walk p*
    and *len*: $1 <$ *length p*
  shows $0 <$ *walk-weight p*
⟨*proof*⟩

end

context *unique-shortest-digraph*
begin

lemma *tree-path-strict-dist-if-positive-edges*:
  assumes *positive*: $\bigwedge u\ v.\ (u,\ v) \in E \Longrightarrow 0 < w\ u\ v$
    and *tree*: *tree-path u v*
    and *neq*: $u \neq v$
  shows *dist s u* $<$ *dist s v*
⟨*proof*⟩

end

locale *strict-tie-breaking-digraph* = *unique-shortest-digraph* +
  assumes *tree-path-strict-dist*:
    ⟦*tree-path u v*; *u* ≠ *v*⟧ ⟹ *dist s u* < *dist s v*

locale *positive-unique-shortest-digraph* = *unique-shortest-digraph* +
  assumes *positive-weight*: (*u*, *v*) ∈ *E* ⟹ *0* < *w u v*
begin

sublocale *strict-tie-breaking-digraph*
⟨*proof*⟩

end

The first locale bridge shows one concrete way to obtain strict tree growth: positive edge weights imply every non-empty suffix of a shortest path has positive weight, and therefore every proper descendant in the selected shortest-path tree has strictly larger distance. The abstract *strict-tie-breaking-digraph* locale records only the consequence needed by the runtime proof, avoiding a commitment to any particular perturbation or tie-breaking construction.

context *strict-tie-breaking-digraph*
begin

lemma *bound-tree-singleton-tree-path*:
  assumes *y* ∈ *bound-tree* {*x*} *B*
  shows *tree-path x y*
⟨*proof*⟩

lemma *bound-tree-singleton-root-strict-min*:
  assumes *x-bound*: *x* ∈ *bound-tree* {*x*} *B*
    and *y-bound*: *y* ∈ *bound-tree* {*x*} *B*
    and *neq*: *y* ≠ *x*
  shows *dist s x* < *dist s y*
⟨*proof*⟩

lemma *base-case-order-hd-root*:
  assumes *x-bound*: *x* ∈ *bound-tree* {*x*} *B*
  shows *base-case-order x B* ≠ [] ∧ *hd* (*base-case-order x B*) = *x*
⟨*proof*⟩

lemma *source-in-base-case-vertices*:
  assumes *x-bound*: *x* ∈ *bound-tree* {*x*} *B*
    and *k-pos*: *0* < *k*
  shows *x* ∈ *base-case-vertices k x B*
⟨*proof*⟩

The base-case lemmas turn strict tree growth into the concrete progress fact used by the recurrence. In a singleton problem, every vertex in *bound-tree* {*x*} *B* lies below *x* in the selected shortest-path tree. Strictness makes *x* the first element of *base-case-order x B*. Hence the bounded base scan always

includes its source whenever $k > 0$.

This small fact is what lets the base case participate in the same source accounting as the recursive cases: the source set has cardinality one and is contained in the returned base-case vertex set.

lemma *tree-path-tight-successor*:
  assumes *tight*: *tight-edge-step u v*
  shows *tree-path u v*
⟨*proof*⟩

lemma *fp-next-bound-tree*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S B*
    and *F-bound*: $F \subseteq$ *bound-tree S B*
    and *v-next*: $v \in$ *fp-next d F B*
  shows $v \in$ *bound-tree S B*
⟨*proof*⟩

lemma *fp-iter-capped-seen-subset-bound-tree*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S B*
    and *F-bound*: $F \subseteq$ *bound-tree S B*
    and *W-bound*: $W \subseteq$ *bound-tree S B*
  shows *fp-seen* (*fp-iter-capped n cap d F W B*) $\subseteq$ *bound-tree S B*
  ⟨*proof*⟩

lemma *find-pivots-seen-capped-subset-bound-tree*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S B*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
    and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
  shows *find-pivots-seen-capped k cap d S B* $\subseteq$ *bound-tree S B*
⟨*proof*⟩

The next block connects FindPivots' "seen" set to completed output ranges. When a non-base call succeeds with the original bound, the final assembly theorem says that the loop output together with already-complete labelled vertices is exactly the parent bound tree. Since the seen set is contained in that tree, successful calls can charge the number of seen vertices to the returned output.

The same argument is stated for both exact split-range runs and direct-insert runs. The direct-insert version first forgets its extra accounting structure down to the exact range trace, then applies the same assembly reasoning. These lemmas are later paired with threshold lemmas for the unsuccessful branch.

theorem *exact-split-range-costed-step-seen-success*:
  assumes *loop*:
    exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*

(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *success*: *B'* = *B*
shows *card* (*find-pivots-seen-capped k cap d S B*) ≤ *card U*
⟨*proof*⟩

theorem *direct-insert-costed-step-seen-success*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *success*: *B'* = *B*
shows *card* (*find-pivots-seen-capped k cap d S B*) ≤ *card U*
⟨*proof*⟩

theorem *direct-insert-costed-step-seen-success-subset*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *success*: *B'* = *B*
shows *find-pivots-seen-capped k cap d S B* ⊆ *U*

⟨*proof*⟩

theorem *direct-insert-costed-capped-step-threshold-if-not-success*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{$v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *not-success*: $B' \neq B$
shows $k * cap \leq$ *card U*
⟨*proof*⟩

theorem *direct-insert-costed-capped-step-scan-insert-budget-from-scaled-seen-or-threshold*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{$v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *S-k-cap*: $k *$ *card S* $\leq$ *cap*
and *anti*: *tree-antichain S*
and *k-pos*: $0 < k$
and *seen-success*:
$B' = B \Longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
$2 * A *$ *card U*

⟨*proof*⟩

theorem *split-range-costed-bmssp-zero-source-card-le-from-label-below*:
assumes *run*:
split-range-costed-bmssp Δ *M-of t h k cap 0 d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *k-pos*: *0* < *k*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *split-range-costed-bmssp-source-card-le-from-label-below-all-levels*:
assumes *run*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *S-cap*: *card S* ≤ *cap*
and *k-pos*: *0* < *k*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *exact-split-range-costed-bmssp-source-card-le-from-label-below-all-levels*:
assumes *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *S-cap*: *card S* ≤ *cap*
and *k-pos*: *0* < *k*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-zero-source-card-le-from-label-below*:
assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap 0 d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *k-pos*: *0* < *k*
shows *card S* ≤ *card U*
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-source-card-le-from-label-below-all-levels*:

assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *S-cap*: *card S* $\leq$ *cap*
and *k-pos*: $0 < k$
shows *card S* $\leq$ *card U*
⟨*proof*⟩

From here the theory closes the recurrence under the strict progress assumptions. The source-progress hypothesis in the direct-insert cost layer is discharged by a case split on the recursion level. At level zero, the strict base-case order gives the source in the returned base set. At positive levels, the success-or-threshold lemmas and the seen-set bound show that a child call's sources are dominated by its output.

The resulting loop-closing theorems are wrappers around the direct-insert cost lemmas from the previous file. Their purpose is to remove the abstract source-progress assumption and replace it by explicit invariants: bounded child source size, tree antichains, and the scaled cardinality condition that the BMSSP schedule maintains.

theorem *direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-ranges*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k *$ *card P* $\leq$ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
$k *$ *card S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq A$
and *k-pos*: $0 < k$
shows $c \leq$
$A *$ *Suc L* $*$ *card U* $+ R *$ *card* (*outgoing-edges U*) $+$

$t * \textit{sum-list}$
$(\textit{map card } (\textit{range-tree-child-direct-edge-range-list } P\ B\ a\ \textit{betas } \textit{bs})) +$
$h * \textit{sum-list}$
$(\textit{map card } (\textit{range-tree-child-edge-range-list } P\ a\ \textit{betas } \textit{bs}))$
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-bound-with-invariants*:
assumes *run*:
$\textit{direct-insert-costed-partition-loop-state } \Delta\ \textit{M-of } t\ h\ k\ \textit{cap } l\ d\ P\ B\ d'\ D\ a$
$\textit{betas } \textit{bs } B'\ \textit{Us } U\ c\ \textit{child-costs}$
and *sound*: $\textit{sound-label } d$
and *pre*: $\textit{bmssp-pre-full } d\ P\ B$
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow \textit{reachable } s\ x$
and *P-k-cap*: $k * \textit{card } P \leq \textit{cap}$
and *P-anti*: $\textit{tree-antichain } P$
and *child-bound*:
$\bigwedge \textit{c-child U-child S-child B-child d-child B-child}'.$
$[\![\textit{direct-insert-costed-bmssp } \Delta\ \textit{M-of } t\ h\ k\ \textit{cap } l\ d\ \textit{S-child B-child}$
$\textit{d-child B-child}'\ \textit{U-child c-child};$
$\textit{bmssp-pre-full } d\ \textit{S-child B-child};$
$\bigwedge x.\ x \in \textit{S-child} \Longrightarrow \textit{reachable } s\ x;$
$\textit{card S-child} \leq \textit{M-of } l;$
$\bigwedge x.\ x \in \textit{S-child} \Longrightarrow \textit{below-bound } (d\ x)\ \textit{B-child};$
$k * \textit{card S-child} \leq \textit{cap};$
$\textit{tree-antichain S-child}]\!]$
$\Longrightarrow \textit{c-child} \leq$
$\textit{bmssp-amortized-cost-bound } A\ R\ h\ t\ q\ L\ \textit{B-child U-child}$
and *M-cap*: $\textit{M-of } l \leq \textit{cap}$
and *source-factor*: $\textit{Suc } h \leq A$
and *k-pos*: $0 < k$
shows $c \leq \textit{bmssp-amortized-cost-bound } A\ R\ h\ t\ (\textit{Suc } q)\ (\textit{Suc } L)\ B\ U$
⟨*proof*⟩

theorem *direct-insert-costed-nonbase-step-closes-amortized-bound-from-child-bound-with-invariants*:
assumes *loop*:
$\textit{direct-insert-costed-partition-loop-state } \Delta\ \textit{M-of } t\ h\ k\ \textit{cap } l\ d\ P\ B\ d'\ D\ a$
$\textit{betas } \textit{bs } B'\ \textit{Us } \textit{U-loop } \textit{c-loop } \textit{child-costs}$
and *sound*: $\textit{sound-label } d$
and *pre*: $\textit{bmssp-pre-full } d\ P\ B$
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow \textit{reachable } s\ x$
and *P-k-cap*: $k * \textit{card } P \leq \textit{cap}$
and *P-anti*: $\textit{tree-antichain } P$
and *child-bound*:
$\bigwedge \textit{c-child U-child S-child B-child d-child B-child}'.$
$[\![\textit{direct-insert-costed-bmssp } \Delta\ \textit{M-of } t\ h\ k\ \textit{cap } l\ d\ \textit{S-child B-child}$
$\textit{d-child B-child}'\ \textit{U-child c-child};$
$\textit{bmssp-pre-full } d\ \textit{S-child B-child};$

$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x;$
$card\ S\text{-}child \leq M\text{-}of\ l;$
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow below\text{-}bound\ (d\ x)\ B\text{-}child;$
$k * card\ S\text{-}child \leq cap;$
$tree\text{-}antichain\ S\text{-}child\rrbracket$
$\Longrightarrow c\text{-}child \leq$
$bmssp\text{-}amortized\text{-}cost\text{-}bound\ A\ R\ h\ t\ q\ L\ B\text{-}child\ U\text{-}child$
and *M-cap*: $M\text{-}of\ l \leq cap$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: $finite\ U$
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq$
$bmssp\text{-}amortized\text{-}cost\text{-}bound\ A\ R\ h\ t\ (Suc\ q)\ (Suc\ (Suc\ L))\ B\ U$
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-invariants*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.$
$find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq cap \Longrightarrow$
$tree\text{-}antichain\ S \Longrightarrow$
$fp\text{-}iter\text{-}capped\text{-}scan\text{-}cost\ k\ cap\ d\ S\ S\ B + c\text{-}insert \leq A * card\ U$

and *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)$ *B*
and *S-k-cap*: $k * \mathit{card}\ S \leq \mathit{cap}$
and *S-anti*: *tree-antichain S*
shows $c \leq$ *bmssp-amortized-cost-bound A R h t l* $(2 * l + 1)$ *B U*
⟨*proof*⟩

theorem *direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k * \mathit{card}\ P \leq \mathit{cap}$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* $(d\ x)$ *B-child*;
$k * \mathit{card}$ *S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of* $l \leq$ *cap*
and *source-factor*: *Suc* $h \leq A$
and *k-pos*: $0 < k$
and *edge-budget*:
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h *$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H *$ *card* (*outgoing-edges U*)
shows $c \leq A *$ *Suc L* $*$ *card U* $+ (R + H) *$ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *direct-insert-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U-loop c-loop child-costs

and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k * card\ P \leq cap$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child' U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
$k * card\ S\text{-}child \leq cap$;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
and *edge-budget*:
$t * sum\text{-}list$
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H * card$ (*outgoing-edges U-loop*)
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$(R + H) * card$ (*outgoing-edges U*)
⟨*proof*⟩

theorem *direct-insert-costed-bmssp-level-bound-from-local-budgets-with-invariants-and-edge-budget*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow$ *M-of i* $\leq$ *cap*
and *edge-budget*:
$\bigwedge$*l d P B d' D a betas bs B' Us U c child-costs*.
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a betas bs B' Us U c child-costs* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
($\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*) $\Longrightarrow$
$t * sum\text{-}list$
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +

$h * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ P\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq H * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ U)$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ \mathit{c\text{-}insert}\ d'\ a\ \mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ \mathit{U\text{-}loop}\ \mathit{c\text{-}loop}$
$\mathit{child\text{-}costs}\ U.$
$D = \mathit{label\text{-}partition\text{-}view}$
$(\mathit{find\text{-}pivots\text{-}label\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B)$
$(\mathit{find\text{-}pivots\text{-}pivots\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B) \Longrightarrow$
$\mathit{partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound}\ \mathit{c\text{-}insert}\ t$
$(\mathit{find\text{-}pivots\text{-}pivots\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B) \Longrightarrow$
$\mathit{direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l$
$(\mathit{find\text{-}pivots\text{-}label\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B)$
$(\mathit{find\text{-}pivots\text{-}pivots\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B)\ B\ d'\ D\ a\ \mathit{betas}\ \mathit{bs}\ B'$
$\mathit{Us}\ \mathit{U\text{-}loop}\ \mathit{c\text{-}loop}\ \mathit{child\text{-}costs} \Longrightarrow$
$\mathit{complete\text{-}on}\ d'$
$\{v \in \mathit{bound\text{-}tree}\ S\ B'.\ \mathit{find\text{-}pivots\text{-}label\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B\ v = \mathit{dist}\ s\ v\} \Longrightarrow$
$U = \mathit{U\text{-}loop} \cup$
$\{v \in \mathit{bound\text{-}tree}\ S\ B'.$
$\mathit{find\text{-}pivots\text{-}label\text{-}capped}\ k\ \mathit{cap}\ d\ S\ B\ v = \mathit{dist}\ s\ v\} \Longrightarrow$
$\mathit{sound\text{-}label}\ d \Longrightarrow$
$\mathit{bmssp\text{-}pre\text{-}full}\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \mathit{below\text{-}bound}\ (d\ x)\ B) \Longrightarrow$
$k * \mathit{card}\ S \leq \mathit{cap} \Longrightarrow$
$\mathit{tree\text{-}antichain}\ S \Longrightarrow$
$\mathit{fp\text{-}iter\text{-}capped\text{-}scan\text{-}cost}\ k\ \mathit{cap}\ d\ S\ S\ B + \mathit{c\text{-}insert} \leq A * \mathit{card}\ U$
and *run*:
$\mathit{direct\text{-}insert\text{-}costed\text{-}bmssp}\ \Delta\ \mathit{M\text{-}of}\ t\ h\ k\ \mathit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c$
and *sound*: $\mathit{sound\text{-}label}\ d$
and *pre*: $\mathit{bmssp\text{-}pre\text{-}full}\ d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{reachable}\ s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow \mathit{below\text{-}bound}\ (d\ x)\ B$
and *S-k-cap*: $k * \mathit{card}\ S \leq \mathit{cap}$
and *S-anti*: $\mathit{tree\text{-}antichain}\ S$
shows $c \leq \mathit{level\text{-}range\text{-}cost\text{-}bound}\ A\ (R + l * H)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-graph-bound-level-cap-from-local-budgets*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow \mathit{reachable}\ s\ v$
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $\mathit{Suc}\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ \mathit{betas}\ \mathit{bs}\ B'\ \mathit{Us}\ \mathit{U\text{-}loop}\ \mathit{c\text{-}loop}\ \mathit{child\text{-}costs}.$
$\mathit{direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state}\ \Delta\ (\mathit{bmssp\text{-}level\text{-}cap}\ k\ t)\ t\ h\ k$
$(\mathit{bmssp\text{-}level\text{-}cap}\ k\ t\ l)\ l'\ d\ P\ B\ d'\ D\ a\ \mathit{betas}\ \mathit{bs}\ B'$

Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h *$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H *$ *card* (*outgoing-edges U-loop*)
and *step-budget*:
$\bigwedge l'$ *d S B D c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l'*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d' D*
a betas bs B' Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
{*v* $\in$ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v} $\Longrightarrow$
U = *U-loop* $\cup$
{*v* $\in$ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v} $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*$) \Longrightarrow$
$k *$ *card S* $\leq$ *bmssp-level-cap k t l* $\Longrightarrow$
tree-antichain S $\Longrightarrow$
fp-iter-capped-scan-cost k (*bmssp-level-cap k t l*) *d S S B* +
c-insert $\leq 2 * A *$ *card U*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows *sssp-correct d'* $\wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * H) *$ *edge-count*
⟨*proof*⟩

corollary *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-from-local-budgets*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*

and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l' *d P B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$t * sum\text{-}list$
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H *$ *card* (*outgoing-edges U-loop*)
and *step-budget*:
$\bigwedge l'\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l'
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d' D*
a betas bs B' Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v$\}$ $\Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in$ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k *$ *card S* $\leq$ *bmssp-level-cap k t l* $\Longrightarrow$
tree-antichain S $\Longrightarrow$
fp-iter-capped-scan-cost k (*bmssp-level-cap k t l*) *d S S B* +
c-insert $\leq 2 * A *$ *card U*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*

finite-initial-label $\{s\}$ *Infinity* d' *Infinity* U c
shows *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. A) (λ-. R) (λ-. H)
(λ-. l) (λ-. t) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-graph-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A$-*insert* $* k$
and *seen-scaled-factor*: $k * \Delta + A$-*insert* $\leq 2 * A$
and *source-factor*: *Suc* $h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap* k t) t h k
(*bmssp-level-cap* k t l) l' d P B d' D a *betas* bs B'
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
($\bigwedge x.\ x \in P \Longrightarrow$ *reachable* s x) $\Longrightarrow$
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list* P B a *betas* bs)) $+$
$h *$ *sum-list*
(*map card* (*range-tree-child-edge-range-list* P a *betas* bs))
$\leq H *$ *card* (*outgoing-edges U-loop*)
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap* k t) t h k
(*bmssp-level-cap* k t l) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* U c
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * H) *$ *edge-count*
⟨*proof*⟩

corollary *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A$-*insert* $* k$
and *seen-scaled-factor*: $k * \Delta + A$-*insert* $\leq 2 * A$
and *source-factor*: *Suc* $h \leq 2 * A$
and *k-pos*: $0 < k$

and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l' *d P B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h *$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H *$ *card* (*outgoing-edges U-loop*)
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
shows *sssp-correct d'* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

lemma *direct-insert-costed-partition-loop-state-trivial-edge-budget*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
betas bs B' Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
shows $t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h *$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq (t + h) *$ *card* (*outgoing-edges U*)
⟨*proof*⟩

corollary *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-trivial-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*

finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t* + *h*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-amortized*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *insert-scaled-factor*: *t* ≤ *A-insert* ∗ *k*
and *seen-scaled-factor*: *k* ∗ Δ + *A-insert* ≤ *2* ∗ *A*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *h*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

The remaining exact split-range results repeat the same source-progress closure without the direct-insert edge split. They are kept because some clients of the development may want the simpler range-costed interface, while the bucketed headline uses the more refined direct-insert and amortised path. The proof shape is the same: extract child calls, use strict progress to dominate child sources by child outputs, and close the level recurrence.

Having both interfaces documented makes the proof stack easier to audit. The exact split-range family is the clean mathematical recurrence, and the direct-insert family is the implementation-aware recurrence that exposes the bucketed partition's local costs.

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-costs*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)

and *M-cap*: *M-of l* ≤ *cap*
and *source-factor*: *Suc h* ≤ *A*
and *k-pos*: *0* < *k*
shows *c* ≤ *A* ∗ *Suc L* ∗ *card U* + (*R* + *t*) ∗ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-costs-and-edge-ranges*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *M-cap*: *M-of l* ≤ *cap*
and *source-factor*: *Suc h* ≤ *A*
and *k-pos*: *0* < *k*
shows *c* ≤
A ∗ *Suc L* ∗ *card U* + *R* ∗ *card* (*outgoing-edges U*) +
t ∗ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* ≤ *cap*
and *source-factor*: *Suc h* ≤ *A*
and *k-pos*: *0* < *k*
shows *c* ≤ *A* ∗ *Suc L* ∗ *card U* + (*R* + *t*) ∗ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-and-edge-ranges*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: *0* $<$ *k*
shows *c* $\leq$
A $*$ *Suc L* $*$ *card U* $+$ *R* $*$ *card* (*outgoing-edges U*) $+$
t $*$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: *k* $*$ *card P* $\leq$ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
k $*$ *card S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: *0* $<$ *k*

shows $c \leq A * Suc\ L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound*:
assumes *loop*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x$;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow below\text{-}bound\ (d\ x)\ B\text{-}child$⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* $\leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$(R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-and-edge-ranges*:
assumes *loop*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x$;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow below\text{-}bound\ (d\ x)\ B\text{-}child$⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*

and *k-pos*: $0 < k$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$R * card\ (outgoing\text{-}edges\ U) +$
$t * sum\text{-}list\ (map\ card\ (range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list\ P\ a\ betas\ bs))$
⟨*proof*⟩

theorem *exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants*:
assumes *loop*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *P-k-cap*: $k * card\ P \leq cap$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge c\text{-}child\ U\text{-}child\ S\text{-}child\ B\text{-}child\ d\text{-}child\ B\text{-}child'.$
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x$;
$card\ S\text{-}child \leq M\text{-}of\ l$;
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow below\text{-}bound\ (d\ x)\ B\text{-}child$;
$k * card\ S\text{-}child \leq cap$;
tree-antichain S-child⟧
$\Longrightarrow c\text{-}child \leq level\text{-}range\text{-}cost\text{-}bound\ A\ R\ L\ U\text{-}child$
and *M-cap*: $M\text{-}of\ l \leq cap$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$(R + t) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *exact-split-range-costed-bmssp-level-bound-from-local-budgets*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$
and *step-budget*:

$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$exact\text{-}split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$fp\text{-}iter\text{-}capped\text{-}scan\text{-}cost\ k\ cap\ d\ S\ S\ B + c\text{-}insert \le A * card\ U$
and *run*:
$exact\text{-}split\text{-}range\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ S\ B\ d'\ B'\ U\ c$
and *sound*: $sound\text{-}label\ d$
and *pre*: $bmssp\text{-}pre\text{-}full\ d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B$
shows $c \le level\text{-}range\text{-}cost\text{-}bound\ A\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *exact-split-range-costed-bmssp-level-bound-from-local-budgets-with-invariants*:
assumes *base-budget*: $\Delta \le A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \le A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \le l \Longrightarrow M\text{-}of\ i \le cap$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$exact\text{-}split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'$

$\{v \in \textit{bound-tree } S\ B'.\ \textit{find-pivots-label-capped } k\ \textit{cap}\ d\ S\ B\ v = \textit{dist } s\ v\} \Longrightarrow$
$U = \textit{U-loop} \cup$
$\{v \in \textit{bound-tree } S\ B'.\ \textit{find-pivots-label-capped } k\ \textit{cap}\ d\ S\ B\ v = \textit{dist } s\ v\} \Longrightarrow$
$\textit{sound-label } d \Longrightarrow$
$\textit{bmssp-pre-full } d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{reachable } s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{below-bound } (d\ x)\ B) \Longrightarrow$
$k * \textit{card } S \leq \textit{cap} \Longrightarrow$
$\textit{tree-antichain } S \Longrightarrow$
$\textit{fp-iter-capped-scan-cost } k\ \textit{cap}\ d\ S\ S\ B + \textit{c-insert} \leq A * \textit{card } U$
and *run*:
$\textit{exact-split-range-costed-bmssp } \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l\ d\ S\ B\ d'\ B'\ U\ c$
and *sound*: $\textit{sound-label } d$
and *pre*: $\textit{bmssp-pre-full } d\ S\ B$
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow \textit{reachable } s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow \textit{below-bound } (d\ x)\ B$
and *S-k-cap*: $k * \textit{card } S \leq \textit{cap}$
and *S-anti*: $\textit{tree-antichain } S$
shows $c \leq \textit{level-range-cost-bound } A\ (R + l * t)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-local-budget-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow \textit{reachable } s\ v$
and *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $\textit{Suc } h \leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow \textit{M-of } i \leq \textit{cap}$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ \textit{c-insert}\ d'\ a\ \textit{betas}\ \textit{bs}\ B'\ \textit{Us}\ \textit{U-loop}\ \textit{c-loop}$
$\textit{child-costs}\ U.$
$D = \textit{label-partition-view}$
$(\textit{find-pivots-label-capped } k\ \textit{cap}\ d\ S\ B)$
$(\textit{find-pivots-pivots-capped } k\ \textit{cap}\ d\ S\ B) \Longrightarrow$
$\textit{partition-initial-insert-cost-bound c-insert } t$
$(\textit{find-pivots-pivots-capped } k\ \textit{cap}\ d\ S\ B) \Longrightarrow$
$\textit{exact-split-range-costed-partition-loop-state } \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l$
$(\textit{find-pivots-label-capped } k\ \textit{cap}\ d\ S\ B)$
$(\textit{find-pivots-pivots-capped } k\ \textit{cap}\ d\ S\ B)\ B\ d'\ D\ a\ \textit{betas}\ \textit{bs}\ B'$
$\textit{Us}\ \textit{U-loop}\ \textit{c-loop}\ \textit{child-costs} \Longrightarrow$
$\textit{complete-on } d'$
$\{v \in \textit{bound-tree } S\ B'.\ \textit{find-pivots-label-capped } k\ \textit{cap}\ d\ S\ B\ v = \textit{dist } s\ v\} \Longrightarrow$
$U = \textit{U-loop} \cup$
$\{v \in \textit{bound-tree } S\ B'.\ \textit{find-pivots-label-capped } k\ \textit{cap}\ d\ S\ B\ v = \textit{dist } s\ v\} \Longrightarrow$
$\textit{sound-label } d \Longrightarrow$
$\textit{bmssp-pre-full } d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{reachable } s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{below-bound } (d\ x)\ B) \Longrightarrow$

*fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U*
and *run*:
exact-split-range-costed-bmssp Δ M-of t h k cap l
finite-initial-label {s} Infinity d' Infinity U c
shows *sssp-correct d' ∧*
*c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U*
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-local-budget-graph-bound*:
assumes *all-reachable*: ⋀*v. v ∈ V ⟹ reachable s v*
and *base-budget*: *Δ ≤ A*
and *R-pos*: *0 < R*
and *source-factor*: *Suc h ≤ A*
and *k-pos*: *0 < k*
and *M-cap*: ⋀*i. i ≤ l ⟹ M-of i ≤ cap*
and *step-budget*:
⋀*l d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U.*
D = label-partition-view (find-pivots-label-capped k cap d S B) (find-pivots-pivots-capped k cap d S B) ⟹
partition-initial-insert-cost-bound c-insert t (find-pivots-pivots-capped k cap d S B) ⟹
exact-split-range-costed-partition-loop-state Δ M-of t h k cap l (find-pivots-label-capped k cap d S B) (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B' Us U-loop c-loop child-costs ⟹
complete-on d' {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
U = U-loop ∪ {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀x. x ∈ S ⟹ reachable s x) ⟹
(⋀x. x ∈ S ⟹ below-bound (d x) B) ⟹
*fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U*
and *run*:
exact-split-range-costed-bmssp Δ M-of t h k cap l
finite-initial-label {s} Infinity d' Infinity U c
shows *sssp-correct d' ∧*
*c ≤ A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count*
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-progress-graph-bound*:
assumes *all-reachable*: ⋀*v. v ∈ V ⟹ reachable s v*
and *degree*: *edge-outdegree-le Δ*
and *degree-factor*: *Δ ≤ A*

and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$
and *step-scaled-seen-progress*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$exact\text{-}split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq cap \wedge tree\text{-}antichain\ S \wedge$
$(B' = B \longrightarrow$
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ U)$
and *run*:
$exact\text{-}split\text{-}range\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$finite\text{-}initial\text{-}label\ \{s\}\ Infinity\ d'\ Infinity\ U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) * vertex\text{-}count + (R + l * t) * edge\text{-}count$
$\langle proof \rangle$

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-progress-graph-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow reachable\ s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *step-scaled-seen-progress*:

$\bigwedge l'\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B) \Longrightarrow$
$exact\text{-}split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ (bmssp\text{-}level\text{-}cap\ k\ t)\ t\ h\ k$
$(bmssp\text{-}level\text{-}cap\ k\ t\ l)\ l'$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)\ B\ d'\ D$
$a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.$
$find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B\ v =$
$dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.$
$find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B\ v =$
$dist\ s\ v\} \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq bmssp\text{-}level\text{-}cap\ k\ t\ l \wedge tree\text{-}antichain\ S \wedge$
$(B' = B \longrightarrow$
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B) \leq$
$card\ U)$
and *run*:
$exact\text{-}split\text{-}range\text{-}costed\text{-}bmssp\ \Delta\ (bmssp\text{-}level\text{-}cap\ k\ t)\ t\ h\ k$
$(bmssp\text{-}level\text{-}cap\ k\ t\ l)\ l$
$finite\text{-}initial\text{-}label\ \{s\}\ Infinity\ d'\ Infinity\ U\ c$
shows $sssp\text{-}correct\ d' \wedge$
$c \leq 2 * A * (2 * l + 1) * vertex\text{-}count + (R + l * t) * edge\text{-}count$
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-success-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow reachable\ s\ v$
and *degree*: $edge\text{-}outdegree\text{-}le\ \Delta$
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *top-cap*: $k \leq cap$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$

and *seen-success*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$exact\text{-}split\text{-}range\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq cap \Longrightarrow$
$tree\text{-}antichain\ S \Longrightarrow$
$B' = B \Longrightarrow$
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ U$
and *run*:
$exact\text{-}split\text{-}range\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$finite\text{-}initial\text{-}label\ \{s\}\ Infinity\ d'\ Infinity\ U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) * vertex\text{-}count + (R + l * t) * edge\text{-}count$
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-success-graph-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow reachable\ s\ v$
and *degree*: $edge\text{-}outdegree\text{-}le\ \Delta$
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *seen-success*:
$\bigwedge l'\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
$child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$

(*find-pivots-pivots-capped* k (*bmssp-level-cap* k t l) d S B) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap* k t) t h k
(*bmssp-level-cap* k t l) l'
(*find-pivots-label-capped* k (*bmssp-level-cap* k t l) d S B)
(*find-pivots-pivots-capped* k (*bmssp-level-cap* k t l) d S B) B d' D
a *betas bs* B' *Us U-loop c-loop child-costs* $\Longrightarrow$
complete-on d'
{$v \in$ *bound-tree* S B'.
find-pivots-label-capped k (*bmssp-level-cap* k t l) d S B v =
dist s v} $\Longrightarrow$
U = *U-loop* $\cup$
{$v \in$ *bound-tree* S B'.
find-pivots-label-capped k (*bmssp-level-cap* k t l) d S B v =
dist s v} $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.$ $x \in S \Longrightarrow$ *reachable* s x) $\Longrightarrow$
($\bigwedge x.$ $x \in S \Longrightarrow$ *below-bound* (d x) B) $\Longrightarrow$
$k * $ *card* $S \leq$ *bmssp-level-cap* k t l $\Longrightarrow$
tree-antichain S $\Longrightarrow$
$B' = B \Longrightarrow$
card (*find-pivots-seen-capped* k (*bmssp-level-cap* k t l) d S B) $\leq$
card U
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* k t) t h k
(*bmssp-level-cap* k t l) l
finite-initial-label {s} *Infinity* d' *Infinity* U c
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* + $(R + l * t) *$ *edge-count*
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-scaled-graph-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.$ $v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A$-*insert* $* k$
and *seen-scaled-factor*: $k * \Delta + A$-*insert* $\leq 2 * A$
and *source-factor*: *Suc* $h \leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* k t) t h k
(*bmssp-level-cap* k t l) l
finite-initial-label {s} *Infinity* d' *Infinity* U c
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* + $(R + l * t) *$ *edge-count*
⟨*proof*⟩

corollary *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-graph-time-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label $\{s\}$ *Infinity d′ Infinity U c*
shows *sssp-correct d′* $\wedge$
$c \leq$ *bmssp-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *l*) (λ-. *t*)
(λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

lemma *exact-split-range-costed-partition-loop-state-trivial-edge-budget*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
shows *t* $*$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq t *$ *card* (*outgoing-edges U*)
⟨*proof*⟩

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k *$ *card P* $\leq$ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;

$\bigwedge x.\ x \in S\text{-}child \Longrightarrow below\text{-}bound\ (d\ x)\ B\text{-}child;$
$k * card\ S\text{-}child \leq cap;$
$tree\text{-}antichain\ S\text{-}child\rrbracket$
$\Longrightarrow c\text{-}child \leq level\text{-}range\text{-}cost\text{-}bound\ A\ R\ L\ U\text{-}child$
and *M-cap*: $M\text{-}of\ l \leq cap$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *edge-budget*:
$t * sum\text{-}list\ (map\ card\ (range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list\ P\ a\ betas\ bs)) \leq$
$H * card\ (outgoing\text{-}edges\ U)$
shows $c \leq A * Suc\ L * card\ U + (R + H) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *loop*:
exact-split-range-costed-partition-loop-state $\Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
$betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs$
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x$
and *P-k-cap*: $k * card\ P \leq cap$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge c\text{-}child\ U\text{-}child\ S\text{-}child\ B\text{-}child\ d\text{-}child\ B\text{-}child'.$
$\llbracket exact\text{-}split\text{-}range\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ S\text{-}child\ B\text{-}child$
$d\text{-}child\ B\text{-}child'\ U\text{-}child\ c\text{-}child;$
$bmssp\text{-}pre\text{-}full\ d\ S\text{-}child\ B\text{-}child;$
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow reachable\ s\ x;$
$card\ S\text{-}child \leq M\text{-}of\ l;$
$\bigwedge x.\ x \in S\text{-}child \Longrightarrow below\text{-}bound\ (d\ x)\ B\text{-}child;$
$k * card\ S\text{-}child \leq cap;$
$tree\text{-}antichain\ S\text{-}child\rrbracket$
$\Longrightarrow c\text{-}child \leq level\text{-}range\text{-}cost\text{-}bound\ A\ R\ L\ U\text{-}child$
and *M-cap*: $M\text{-}of\ l \leq cap$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *edge-budget*:
$t * sum\text{-}list\ (map\ card\ (range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list\ P\ a\ betas\ bs)) \leq$
$H * card\ (outgoing\text{-}edges\ U\text{-}loop)$
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: $c\text{-}scan\text{-}insert \leq A * card\ U$
and *c-def*: $c = c\text{-}scan\text{-}insert + c\text{-}loop$
shows $c \leq A * Suc\ (Suc\ L) * card\ U + (R + H) * card\ (outgoing\text{-}edges\ U)$
⟨*proof*⟩

theorem *exact-split-range-costed-bmssp-level-bound-from-local-budgets-with-invariants-and-edge-budget*:

assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$
and *edge-budget*:
$\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a betas bs B' Us U c child-costs* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$t * sum\text{-}list\ (map\ card\ (range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list\ P\ a\ betas\ bs))$
$\leq H * card\ (outgoing\text{-}edges\ U)$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq cap \Longrightarrow$
tree-antichain S $\Longrightarrow$
fp-iter-capped-scan-cost k cap d S S B $+$ *c-insert* $\leq A * card\ U$
and *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B$
and *S-k-cap*: $k * card\ S \leq cap$
and *S-anti*: *tree-antichain S*
shows $c \leq level\text{-}range\text{-}cost\text{-}bound\ A\ (R + l * H)\ (2 * l + 1)\ U$
⟨*proof*⟩

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-*

refined-graph-bound-level-cap:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) $l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$t *$ *sum-list* (*map card* (*range-tree-child-edge-range-list* $P\ a\ betas\ bs$))
$\leq H *$ *card* (*outgoing-edges* $U\text{-}loop$)
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * H) *$ *edge-count*
⟨*proof*⟩

corollary *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) $l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$t *$ *sum-list* (*map card* (*range-tree-child-edge-range-list* $P\ a\ betas\ bs$))
$\leq H *$ *card* (*outgoing-edges* $U\text{-}loop$)

and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows *sssp-correct d'* $\wedge$
c $\leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

corollary *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-trivial-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: *0* $<$ *R*
and *insert-factor*: *t* $\leq$ *A* $*$ *k*
and *insert-scaled-factor*: *t* $\leq$ *A-insert* $*$ *k*
and *seen-scaled-factor*: *k* $*$ Δ + *A-insert* $\leq$ *2* $*$ *A*
and *source-factor*: *Suc h* $\leq$ *2* $*$ *A*
and *k-pos*: *0* $<$ *k*
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows *sssp-correct d'* $\wedge$
c $\leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

end

end
theory *BMSSP-Partition-Data-Structure*
imports *BMSSP-Partition-Pull-Bridge*
begin

34 Concrete Partition Data Structure

The abstract partition interface uses a set of active keys together with a value function. A concrete implementation must be careful around *Pull*: the abstract contract removes keys but leaves the value function unchanged. The state below therefore stores a sorted active list and a value memory. Removing a key only removes it from the active list; the remembered value is retained for the abstraction relation.

record *'k partition-state* =
entries-of :: (*'k* $\times$ *real*) *list*
values-of :: *'k* $\Rightarrow$ *real*

definition *entry-keys* :: $('k \times real)\ list \Rightarrow 'k\ set$ **where**
entry-keys xs = *fst* ‘ *set xs*

definition *partition-state-invar* :: $'k$ *partition-state* ⇒ *bool* **where**
partition-state-invar P ⟷
distinct (*map fst* (*entries-of P*)) ∧
sorted-wrt (λ*p q*. *snd p* ≤ *snd q*) (*entries-of P*) ∧
(∀ (*x*, *b*)∈*set* (*entries-of P*). *values-of P x* = *b*)

definition *partition-state-view* :: $'k$ *partition-state* ⇒ $'k$ *partition-view* **where**
partition-state-view P =
(| *keys-of* = *entry-keys* (*entries-of P*), *value-of* = *values-of P* |)

definition *partition-state-refines* ::
$'k$ *partition-state* ⇒ $'k$ *partition-view* ⇒ *bool* **where**
partition-state-refines P D ⟷
partition-state-invar P ∧ *partition-state-view P* = *D*

definition *empty-partition-state* :: $('k \Rightarrow real) \Rightarrow 'k$ *partition-state* **where**
empty-partition-state v = (| *entries-of* = [], *values-of* = *v* |)

definition *partition-state-from-keys* ::
$('k \Rightarrow real) \Rightarrow 'k\ list \Rightarrow 'k$ *partition-state* **where**
partition-state-from-keys d xs =
(| *entries-of* = *sort-key snd* (*map* (λ*x*. (*x*, *d x*)) *xs*),
values-of = *d* |)

definition *partition-state-insert* ::
$'k \Rightarrow real \Rightarrow 'k$ *partition-state* ⇒ $'k$ *partition-state* **where**
partition-state-insert x b P =
(*let b′* = (*if x* ∈ *entry-keys* (*entries-of P*)
then min (*values-of P x*) *b else b*);
rest = *filter* (λ*p*. *fst p* ≠ *x*) (*entries-of P*)
in (| *entries-of* = *sort-key snd* ((*x*, *b′*) # *rest*),
values-of = (*values-of P*)(*x* := *b′*) |))

definition *partition-state-batch-prepend* ::
$('k \times real)\ list \Rightarrow 'k$ *partition-state* ⇒ $'k$ *partition-state* **where**
partition-state-batch-prepend xs P =
fold (λ(*x*, *b*) *P′*. *partition-state-insert x b P′*) *xs P*

definition *partition-state-pull-prefix* ::
nat ⇒ $'k$ *partition-state* ⇒ $('k \times real)\ list$ **where**
partition-state-pull-prefix M P =
(*let xs* = *entries-of P in*
if length xs ≤ *M then xs*
else takeWhile (λ*p*. *snd p* < *snd* (*xs* ! *M*)) *xs*)

definition *partition-state-pull-set* ::

nat ⇒ *'k partition-state* ⇒ *'k set* **where**
partition-state-pull-set M P =
entry-keys (*partition-state-pull-prefix M P*)

definition *partition-state-pull-bound* ::
nat ⇒ *real* ⇒ *'k partition-state* ⇒ *real* **where**
partition-state-pull-bound M B P =
(*let xs* = *entries-of P in if length xs* ≤ *M then B else snd* (*xs* ! *M*))

definition *partition-state-delete-keys* ::
'k set ⇒ *'k partition-state* ⇒ *'k partition-state* **where**
partition-state-delete-keys S P =
(| *entries-of* = *filter* (λ*p*. *fst p* ∉ *S*) (*entries-of P*),
values-of = *values-of P* |)

definition *partition-state-pull* ::
nat ⇒ *real* ⇒ *'k partition-state* ⇒
'k set × *real* × *'k partition-state* **where**
partition-state-pull M B P =
(*let S* = *partition-state-pull-set M P*;
beta = *partition-state-pull-bound M B P*
in (*S*, *beta*, *partition-state-delete-keys S P*))

definition *costed-partition-state-insert* ::
nat ⇒ *'k partition-state* ⇒ *'k* ⇒ *real* ⇒ *nat* ⇒
'k partition-state ⇒ *bool* **where**
costed-partition-state-insert t P x b c P' ⟷
P' = *partition-state-insert x b P* ∧ *c* = *t*

definition *costed-partition-state-batch-prepend* ::
nat ⇒ *'k partition-state* ⇒ (*'k* × *real*) *list* ⇒ *nat* ⇒
'k partition-state ⇒ *bool* **where**
costed-partition-state-batch-prepend t P xs c P' ⟷
P' = *partition-state-batch-prepend xs P* ∧ *c* = *t* ∗ *length xs*

definition *costed-partition-state-pull* ::
nat ⇒ *real* ⇒ *'k partition-state* ⇒ *'k set* ⇒ *real* ⇒
'k partition-state ⇒ *nat* ⇒ *bool* **where**
costed-partition-state-pull M B P S beta P' c ⟷
partition-state-pull M B P = (*S*, *beta*, *P'*) ∧ *c* = *card S*

lemma *costed-partition-state-insert-exists*:
∃ *c P'*. *costed-partition-state-insert t P x b c P'*
⟨*proof*⟩

lemma *costed-partition-state-insert-deterministic*:
assumes *costed-partition-state-insert t P x b c P'*
and *costed-partition-state-insert t P x b c' P''*
shows *c* = *c'* ∧ *P'* = *P''*

⟨*proof*⟩

lemma *costed-partition-state-batch-prepend-exists*:
∃ *c P'*. *costed-partition-state-batch-prepend t P xs c P'*
⟨*proof*⟩

lemma *costed-partition-state-batch-prepend-deterministic*:
assumes *costed-partition-state-batch-prepend t P xs c P'*
and *costed-partition-state-batch-prepend t P xs c' P''*
shows $c = c' \wedge P' = P''$
⟨*proof*⟩

lemma *costed-partition-state-pull-exists*:
∃ *S beta P' c*. *costed-partition-state-pull M B P S beta P' c*
⟨*proof*⟩

lemma *costed-partition-state-pull-deterministic*:
assumes *costed-partition-state-pull M B P S beta P' c*
and *costed-partition-state-pull M B P S' beta' P'' c'*
shows $S = S' \wedge beta = beta' \wedge P' = P'' \wedge c = c'$
⟨*proof*⟩

lemma *entry-keys-simps* [*simp*]:
entry-keys [] = {}
entry-keys (*x* # *xs*) = *insert* (*fst x*) (*entry-keys xs*)
⟨*proof*⟩

lemma *entry-keys-filter-notin*:
entry-keys (*filter* (λ*p*. *fst p* ∉ *S*) *xs*) = *entry-keys xs* − *S*
⟨*proof*⟩

lemma *entry-keys-filter-neq*:
entry-keys (*filter* (λ*p*. *fst p* ≠ *x*) *xs*) = *entry-keys xs* − {*x*}
⟨*proof*⟩

lemma *entry-keys-sort-key* [*simp*]:
entry-keys (*sort-key f xs*) = *entry-keys xs*
⟨*proof*⟩

lemma *entry-keys-insort-key* [*simp*]:
entry-keys (*insort-key f x xs*) = *insert* (*fst x*) (*entry-keys xs*)
⟨*proof*⟩

lemma *sorted-wrt-snd-sort-key* [*simp*]:
sorted-wrt (λ*p q*. *snd p* ≤ *snd q*) (*sort-key snd xs*)
⟨*proof*⟩

lemma *distinct-map-fst-sort-key*:
assumes *distinct* (*map fst xs*)

shows *distinct* (*map fst* (*sort-key f xs*))
⟨*proof*⟩

lemma *empty-partition-state-invar* [*simp*]:
partition-state-invar (*empty-partition-state v*)
⟨*proof*⟩

lemma *empty-partition-state-view* [*simp*]:
partition-state-view (*empty-partition-state v*) =
(| *keys-of* = {}, *value-of* = *v* |)
⟨*proof*⟩

lemma *partition-state-from-keys-invar*:
assumes *distinct xs*
shows *partition-state-invar* (*partition-state-from-keys d xs*)
⟨*proof*⟩

lemma *partition-state-from-keys-view*:
partition-state-view (*partition-state-from-keys d xs*) =
(| *keys-of* = *set xs*, *value-of* = *d* |)
⟨*proof*⟩

lemma *partition-state-from-keys-refines*:
assumes *distinct xs*
shows *partition-state-refines* (*partition-state-from-keys d xs*)
(| *keys-of* = *set xs*, *value-of* = *d* |)
⟨*proof*⟩

lemma *partition-state-insert-invar*:
assumes *inv*: *partition-state-invar P*
shows *partition-state-invar* (*partition-state-insert x b P*)
⟨*proof*⟩

theorem *partition-state-insert-refines-min-update*:
assumes *partition-state-invar P*
shows *partition-state-view* (*partition-state-insert x b P*) =
min-update (*partition-state-view P*) *x b*
⟨*proof*⟩

theorem *partition-state-insert-refines-insert-spec*:
assumes *inv*: *partition-state-invar P*
shows *insert-spec* (*partition-state-view P*) *x b*
(*partition-state-view* (*partition-state-insert x b P*))
⟨*proof*⟩

theorem *partition-state-insert-refines-view*:
assumes *ref*: *partition-state-refines P D*
shows *partition-state-refines* (*partition-state-insert x b P*)
(*min-update D x b*)

⟨*proof*⟩

lemma *partition-state-batch-prepend-invar*:
assumes *partition-state-invar P*
shows *partition-state-invar* (*partition-state-batch-prepend xs P*)
⟨*proof*⟩

theorem *partition-state-batch-prepend-refines-batch-min-update*:
assumes *partition-state-invar P*
shows *partition-state-view* (*partition-state-batch-prepend xs P*) =
batch-min-update (*partition-state-view P*) *xs*
⟨*proof*⟩

theorem *partition-state-batch-prepend-refines-view*:
assumes *ref*: *partition-state-refines P D*
shows *partition-state-refines* (*partition-state-batch-prepend xs P*)
(*batch-min-update D xs*)
⟨*proof*⟩

lemma *partition-state-delete-keys-invar*:
assumes *partition-state-invar P*
shows *partition-state-invar* (*partition-state-delete-keys S P*)
⟨*proof*⟩

lemma *partition-state-delete-keys-view*:
partition-state-view (*partition-state-delete-keys S P*) =
(| *keys-of* = *keys-of* (*partition-state-view P*) − *S*,
value-of = *value-of* (*partition-state-view P*) |)
⟨*proof*⟩

lemma *partition-state-pull-prefix-subset*:
set (*partition-state-pull-prefix M P*) ⊆ *set* (*entries-of P*)
⟨*proof*⟩

lemma *partition-state-pull-set-subset*:
partition-state-pull-set M P ⊆ *keys-of* (*partition-state-view P*)
⟨*proof*⟩

lemma *partition-state-pull-prefix-card-le*:
assumes *inv*: *partition-state-invar P*
shows *card* (*partition-state-pull-set M P*) ≤ *M*
⟨*proof*⟩

lemma *sorted-wrt-takeWhile-less-eq-filter*:
fixes *f* :: *'a* ⇒ *'b*::*linorder*
assumes *sorted*: *sorted-wrt* (λ*x y*. *f x* ≤ *f y*) *xs*
shows *takeWhile* (λ*x*. *f x* < *t*) *xs* = *filter* (λ*x*. *f x* < *t*) *xs*
⟨*proof*⟩

lemma *partition-state-pull-set-exact*:
  assumes *inv*: *partition-state-invar P*
    and *upper*: $\bigwedge u.\ u \in$ *keys-of* (*partition-state-view P*) $\Longrightarrow$
      value-of (*partition-state-view P*) *u* $<$ *B*
  shows *partition-state-pull-set M P* =
    {*u* $\in$ *keys-of* (*partition-state-view P*).
      value-of (*partition-state-view P*) *u* $<$ *partition-state-pull-bound M B P*}
⟨*proof*⟩

theorem *partition-state-pull-refines-pull-separates*:
  assumes *inv*: *partition-state-invar P*
    and *upper*: $\bigwedge u.\ u \in$ *keys-of* (*partition-state-view P*) $\Longrightarrow$
      value-of (*partition-state-view P*) *u* $<$ *B*
    and *pull*: *partition-state-pull M B P* = (*S*, *beta*, *P'*)
  shows *pull-separates* (*partition-state-view P*) *M B S beta*
    (*partition-state-view P'*)
⟨*proof*⟩

theorem *partition-state-pull-refines-view*:
  assumes *ref*: *partition-state-refines P D*
    and *upper*: $\bigwedge u.\ u \in$ *keys-of D* $\Longrightarrow$ *value-of D u* $<$ *B*
    and *pull*: *partition-state-pull M B P* = (*S*, *beta*, *P'*)
  defines *D'* $\equiv$ ⦇ *keys-of* = *keys-of D* − *S*, *value-of* = *value-of D* ⦈
  shows *partition-state-refines P' D'* $\wedge$ *pull-separates D M B S beta D'*
⟨*proof*⟩

theorem *partition-state-pull-invar*:
  assumes *inv*: *partition-state-invar P*
    and *pull*: *partition-state-pull M B P* = (*S*, *beta*, *P'*)
  shows *partition-state-invar P'*
  ⟨*proof*⟩

theorem *costed-partition-state-insert-refines*:
  assumes *inv*: *partition-state-invar P*
    and *op*: *costed-partition-state-insert t P x b c P'*
  shows *partition-state-invar P'* $\wedge$
    insert-spec (*partition-state-view P*) *x b* (*partition-state-view P'*) $\wedge$
    partition-insert-cost-bound c t
⟨*proof*⟩

theorem *costed-partition-state-batch-prepend-refines*:
  assumes *inv*: *partition-state-invar P*
    and *op*: *costed-partition-state-batch-prepend t P xs c P'*
  shows *partition-state-invar P'* $\wedge$
    partition-state-view P' = *batch-min-update* (*partition-state-view P*) *xs* $\wedge$
    partition-batch-cost-bound c t xs
⟨*proof*⟩

theorem *costed-partition-state-pull-refines*:

assumes *inv*: *partition-state-invar P*
and *upper*: $\bigwedge u.\ u \in$ *keys-of* (*partition-state-view P*) $\Longrightarrow$
value-of (*partition-state-view P*) $u < B$
and *op*: *costed-partition-state-pull M B P S beta P′ c*
shows *partition-state-invar P′* $\wedge$
pull-separates (*partition-state-view P*) *M B S beta*
(*partition-state-view P′*) $\wedge$
partition-pull-cost-bound c S
⟨*proof*⟩

context *unique-shortest-digraph*
begin

lemma *partition-state-from-keys-label-view*:
partition-state-view (*partition-state-from-keys d xs*) =
label-partition-view d (*set xs*)
⟨*proof*⟩

theorem *partition-state-pull-label-set-eq-split-below*:
assumes *inv*: *partition-state-invar* P_D
and *view*: *partition-state-view* P_D = *label-partition-view d S*
and *upper*: $\bigwedge u.\ u \in S \Longrightarrow d\ u < B$
and *pull*: *partition-state-pull M B* P_D = (*S-pull*, *beta*, $P_D{}'$)
shows *S-pull* = *split-below d S beta*
⟨*proof*⟩

theorem *partition-state-pull-establishes-lower-pre*:
assumes *inv*: *partition-state-invar* P_D
and *view*: *partition-state-view* P_D = *label-partition-view d S*
and *pre*: *bmssp-pre-full d S* (*Fin B*)
and *upper*: $\bigwedge u.\ u \in S \Longrightarrow d\ u < B$
and *pull*: *partition-state-pull M B* P_D = (*S-pull*, *beta*, $P_D{}'$)
shows *bmssp-pre-full d S-pull* (*Fin beta*)
⟨*proof*⟩

end

end
theory *BMSSP-Exact-Concrete-Cost*
imports *BMSSP-Direct-Insert-Costed BMSSP-Partition-Data-Structure*
begin

35 Exact Concrete Cost Runs

The existing direct-insert costed run relation records data-structure costs by upper-bound predicates. The relation below is a concrete-cost refinement layer: pulls and batch-prepends are performed by the concrete partition state operations, whose costs are exact. Its main theorem shows that every exact

concrete run is accepted by the existing direct-insert costed relation, so all already-proved graph-time bounds apply to such runs.

Initial insertion is kept as a separate exact initial-state predicate. It records an exact insertion trace and the resulting abstract view; future totality work can instantiate it with a concrete enumeration of the pivot set.

This theory is not the final executable bucketed structure. It is the bridge from abstract cost predicates to concrete partition-state operations. The direct-insert relation says "there exists a pull cost satisfying the pull predicate" and similarly for batch costs. Here those costs are the values returned by *costed-partition-state-pull* and *costed-partition-state-batch-prepend*. The refinement theorem then proves that these exact operations satisfy the abstract cost relation.

The file also contains witness lemmas. They show that an initial partition can be built by inserting the pivot set, that terminal loop cases can be produced directly, and that nonterminal loop cases can be assembled from one concrete pull, one recursive child call, and three concrete batch prepends. These witnesses make the concrete-cost layer a useful target for later totality or executable-search work, even though the main Track 2 deliverable uses the bucketed amortised bounds.

inductive *exact-partition-state-insert-list* ::
  nat ⇒ *'k partition-state* ⇒ (*'k* × *real*) *list* ⇒ *nat* ⇒
    'k partition-state ⇒ *bool* **where**
  Exact-Insert-List-Nil:
    exact-partition-state-insert-list t P [] *0 P*
| *Exact-Insert-List-Cons*:
    costed-partition-state-insert t P x b c-x P_1 ⟹
     exact-partition-state-insert-list t P_1 *xs c-tail P′* ⟹
     c = *c-x* + *c-tail* ⟹
     exact-partition-state-insert-list t P ((*x*, *b*) # *xs*) *c P′*

Exact initial insertion is represented as a list trace. The trace is small enough to reason about directly: each element costs exactly t, preserves the partition invariant, and refines to a minimum-update on the abstract partition view. Building an initial partition from a distinct enumeration of the pivot set therefore gives the abstract label partition view expected by the BMSSP loop.

lemma *batch-min-update-append* [*simp*]:
  batch-min-update (*batch-min-update D xs*) *ys* = *batch-min-update D* (*xs* @ *ys*)
  ⟨*proof*⟩

lemma *exact-partition-state-insert-list-cost*:
  assumes *exact-partition-state-insert-list t P xs c P′*
  shows *c* = *t* ∗ *length xs*
  ⟨*proof*⟩

lemma *exact-partition-state-insert-list-invar*:

assumes *inv*: *partition-state-invar P*
and *inserts*: *exact-partition-state-insert-list t P xs c P'*
shows *partition-state-invar P'*
⟨*proof*⟩

lemma *exact-partition-state-insert-list-batch-prepend*:
exact-partition-state-insert-list t P xs (*t* ∗ *length xs*)
(*partition-state-batch-prepend xs P*)
⟨*proof*⟩

context *unique-shortest-digraph*
begin

lemma *batch-min-update-label-pairs-value-disjoint*:
assumes *distinct*: *distinct xs*
and *disjoint*: *set xs* ∩ *keys-of D* = {}
shows *value-of* (*batch-min-update D* (*map* (λ*x*. (*x*, *d x*)) *xs*)) *y* =
(*if y* ∈ *set xs then d y else value-of D y*)
⟨*proof*⟩

lemma *min-update-label-partition-view-new*:
assumes *x* ∉ *S*
shows *min-update* (*label-partition-view d S*) *x* (*d x*) =
label-partition-view d (*insert x S*)
⟨*proof*⟩

lemma *batch-min-update-label-partition-view-disjoint*:
assumes *distinct*: *distinct xs*
and *disjoint*: *set xs* ∩ *S* = {}
shows *batch-min-update* (*label-partition-view d S*)
(*map* (λ*x*. (*x*, *d x*)) *xs*) =
label-partition-view d (*S* ∪ *set xs*)
⟨*proof*⟩

lemma *batch-min-update-empty-label-view*:
assumes *distinct*: *distinct xs*
shows *batch-min-update* (*partition-state-view* (*empty-partition-state d*))
(*map* (λ*x*. (*x*, *d x*)) *xs*) =
label-partition-view d (*set xs*)
⟨*proof*⟩

definition *exact-partition-initial-state* **where**
exact-partition-initial-state t d P P$_D$ *c* ⟷
partition-state-invar P$_D$ ∧
partition-state-view P$_D$ = *label-partition-view d P* ∧
c = *t* ∗ *card P* ∧
(∃ *xs*. *distinct xs* ∧ *set xs* = *P* ∧
exact-partition-state-insert-list t (*empty-partition-state d*)
(*map* (λ*x*. (*x*, *d x*)) *xs*) *c P*$_D$)

The predicate *exact-partition-initial-state* packages the initial build phase for a non-base BMSSP call. It records three facts at once: the concrete partition state is well formed, its abstract view is exactly the label view of the pivot set, and the build cost is exactly $t * \mathit{card}\ P$. The existence lemmas below use finite pivot sets to construct such a state by a batch of concrete inserts.

lemma *exact-partition-initial-state-invar*:
  assumes *exact-partition-initial-state t d P* P_D *c*
  shows *partition-state-invar* P_D
  ⟨*proof*⟩

lemma *exact-partition-initial-state-view*:
  assumes *exact-partition-initial-state t d P* P_D *c*
  shows *partition-state-view* P_D = *label-partition-view d P*
  ⟨*proof*⟩

lemma *exact-partition-initial-state-cost-bound*:
  assumes *exact-partition-initial-state t d P* P_D *c*
  shows *partition-initial-insert-cost-bound c t P*
  ⟨*proof*⟩

lemma *exact-partition-initial-state-cost*:
  assumes *exact-partition-initial-state t d P* P_D *c*
  shows $c = t * \mathit{card}\ P$
  ⟨*proof*⟩

lemma *exact-partition-initial-state-exists*:
  assumes *finite-P*: *finite P*
  shows ∃ P_D *c*. *exact-partition-initial-state t d P* P_D *c*
⟨*proof*⟩

lemma *exact-partition-initial-state-exists-for-capped-pivots*:
  assumes *S-subset*: $S \subseteq V$
  shows ∃ P_D *c*. *exact-partition-initial-state t*
    (*find-pivots-label-capped k cap d S B*)
    (*find-pivots-pivots-capped k cap d S B*) P_D *c*
⟨*proof*⟩

lemma *exact-partition-initial-state-exists-for-capped-pivots-with-cost*:
  assumes *S-subset*: $S \subseteq V$
  shows ∃ P_D. *exact-partition-initial-state t*
    (*find-pivots-label-capped k cap d S B*)
    (*find-pivots-pivots-capped k cap d S B*) P_D
    (*t* ∗ *card* (*find-pivots-pivots-capped k cap d S B*))
⟨*proof*⟩

lemma *find-pivots-pivots-capped-reaches*:
  assumes *reaches*: ∀ $x \in S$. *reachable s x*

shows $\forall\, x{\in}$*find-pivots-pivots-capped k cap d S B. reachable s x*
⟨*proof*⟩

lemma *find-pivots-pivots-capped-label-less-finite*:
assumes *below*: $\bigwedge y.\ y \in S \Longrightarrow d\ y < Bmax$
and *xP*: $x \in$ *find-pivots-pivots-capped k cap d S* (*Fin Bmax*)
shows *find-pivots-label-capped k cap d S* (*Fin Bmax*) $x < Bmax$
⟨*proof*⟩

The main exact concrete run relation follows the same tree as the direct-insert costed relation, but it keeps concrete partition states in the loop. A step performs a concrete pull, recursively solves the pulled lower split, then performs three concrete batch prepends: direct edge relaxations, lower edge relaxations, and source labels. The accumulated cost is not merely bounded by predicates; it is the exact sum of the concrete operation costs, the child cost, and the tail-loop cost.

This exactness is useful for refinement. The direct-insert relation is the mature cost interface used by the graph-time theorems. By proving that every exact concrete run refines to that relation, all existing correctness and runtime results become available for the concrete partition-state execution.

inductive *exact-concrete-partition-loop-state*
and *exact-concrete-bmssp* **where**
Exact-Concrete-State-Done:
keys-of (*partition-state-view D*) = {} ⟹
bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P B*) ⟹
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
[] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a B*])) *0* []
| *Exact-Concrete-State-Stop*:
bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ⟹
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
[] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)])) *0* []
| *Exact-Concrete-State-Step*:
partition-state-invar D ⟹
partition-upper-bound (*partition-state-view D*) *Bmax* ⟹
costed-partition-state-pull (*M-of l*) *Bmax D S-pull beta D-pull c-pull* ⟹
bound-le (*Fin beta*) *B* ⟹
bmssp-pre-full d S-pull (*Fin beta*) ⟹
S-pull = *split-below d P beta* ⟹
(∀ *x*∈*S-pull. reachable s x*) ⟹
a ≤ *b* ⟹
bound-le (*Fin a*) *B′* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ⟹

exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child* $\Longrightarrow$
U-child = *range-tree P a* (*Fin b*) $\Longrightarrow$
complete-preserved d-child d' U-child $\Longrightarrow$
direct-edge-batch = *edge-relaxation-pairs-in-bound d-child U-child beta B* $\Longrightarrow$
lower-edge-batch = *edge-relaxation-pairs-between d-child U-child b beta* $\Longrightarrow$
source-batch = *label-pairs-between d S-pull b beta* $\Longrightarrow$
costed-partition-state-batch-prepend t D-pull direct-edge-batch c-direct D-direct
$\Longrightarrow$
costed-partition-state-batch-prepend h D-direct lower-edge-batch c-lower D-lower
$\Longrightarrow$
costed-partition-state-batch-prepend h D-lower source-batch c-sources D-next
$\Longrightarrow$
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d'*
D-next b betas bs B' Us-tail U-tail c-tail child-costs-tail $\Longrightarrow$
c = *c-pull* + *c-direct* + *c-lower* + *c-sources* + *c-child* + *c-tail* $\Longrightarrow$
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
(*beta* # *betas*) (*b* # *bs*) *B'*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) $\cup$
$\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c*
(*c-child* # *child-costs-tail*)
| *Exact-Concrete-Base*:
S = {*x*} $\Longrightarrow$
exact-concrete-bmssp Δ *M-of t h k cap 0 d S B*
(λ*v. if v* $\in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
(*base-case-scan-cost* Δ *k x B*)
| *Exact-Concrete-Step*:
exact-partition-initial-state t
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *D c-insert* $\Longrightarrow$
exact-concrete-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a*
betas bs B' Us U-loop c-loop child-costs-loop $\Longrightarrow$
complete-on d'
{*v* $\in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v* = *dist s v*} $\Longrightarrow$
U = *U-loop* $\cup$
{*v* $\in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v* = *dist s v*} $\Longrightarrow$
c = *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* + *c-loop* $\Longrightarrow$
exact-concrete-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d' B' U c*

inductive-cases *exact-concrete-bmssp-zeroE*:
exact-concrete-bmssp Δ *M-of t h k cap 0 d S B d' B' U c*

inductive-cases *exact-concrete-bmssp-SucE*:
exact-concrete-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d' B' U c*

lemma *exact-partition-initial-state-cost-unique*:
assumes *exact-partition-initial-state t d P* P_D *c*
and *exact-partition-initial-state t d P* $P_D{}'$ *c'*
shows $c = c'$
⟨*proof*⟩

lemma *exact-partition-initial-state-upper-bound*:
assumes *init*: *exact-partition-initial-state t d P* P_D *c*
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < B$
shows *partition-upper-bound* (*partition-state-view* P_D) *B*
⟨*proof*⟩

lemma *exact-partition-initial-state-pull-split*:
assumes *init*: *exact-partition-initial-state t d P* P_D *c*
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < B$
and *pull*: *partition-state-pull M B* P_D = (*S-pull*, *beta*, $P_D{}'$)
shows *S-pull* = *split-below d P beta*
⟨*proof*⟩

lemma *exact-partition-initial-state-pull-exact-data*:
assumes *init*: *exact-partition-initial-state t d P* P_D *c*
and *pre*: *bmssp-pre-full d P* (*Fin B*)
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < B$
and *pull*: *partition-state-pull M B* P_D = (*S-pull*, *beta*, $P_D{}'$)
shows *costed-partition-state-pull M B* P_D *S-pull beta* $P_D{}'$ (*card S-pull*) ∧
S-pull = *split-below d P beta* ∧
bmssp-pre-full d S-pull (*Fin beta*) ∧
partition-state-invar $P_D{}'$
⟨*proof*⟩

lemma *exact-partition-initial-state-pull-reaches*:
assumes *init*: *exact-partition-initial-state t d P* P_D *c*
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < B$
and *pull*: *partition-state-pull M B* P_D = (*S-pull*, *beta*, $P_D{}'$)
and *P-reaches*: $\forall\, x{\in}P.$ *reachable s x*
shows $\forall\, x{\in}$*S-pull*. *reachable s x*
⟨*proof*⟩

lemma *exact-partition-initial-state-pull-beta-bound*:
assumes *init*: *exact-partition-initial-state t d P* P_D *c*
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < B$
and *pull*: *partition-state-pull M B* P_D = (*S-pull*, *beta*, $P_D{}'$)
shows *bound-le* (*Fin beta*) (*Fin B*)
⟨*proof*⟩

The pull lemmas are the local bridge from a concrete partition state to the BMSSP child problem. Starting from an exact initial state, a concrete pull refines to *pull-separates* on the abstract label partition view. Therefore the

pulled source set is exactly *split-below d P beta*, it inherits the child BMSSP precondition, and its vertices remain reachable. These are the premises required by the recursive exact concrete step.

lemma *exact-partition-initial-state-exists-with-cost*:
assumes *finite-P*: *finite P*
shows $\exists\, P_D.$ *exact-partition-initial-state t d P* P_D $(t * \mathit{card}\ P)$
⟨*proof*⟩

lemma *exact-concrete-bmssp-base-exists*:
assumes $S = \{x\}$
shows *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B*
(λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
(*base-case-scan-cost* Δ *k x B*)
⟨*proof*⟩

lemma *exact-concrete-bmssp-zero-singletonD*:
assumes *run*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d* $\{x\}$ *B d′ B′ U c*
shows *d′* =
(λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*) ∧
B′ = *base-case-bound k x B* ∧
U = *base-case-vertices k x B* ∧
c = *base-case-scan-cost* Δ *k x B*
⟨*proof*⟩

lemma *exact-concrete-bmssp-zero-singleton-cost-unique*:
assumes *run*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d* $\{x\}$ *B d′ B′ U c*
and *run′*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d* $\{x\}$ *B d″ B″ U′ c′*
shows $c = c'$
⟨*proof*⟩

lemma *exact-concrete-bmssp-zeroD*:
assumes *run*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d′ B′ U c*
shows $\exists\, x.\ S = \{x\} \wedge$
d′ = (λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*) ∧
B′ = *base-case-bound k x B* ∧
U = *base-case-vertices k x B* ∧
c = *base-case-scan-cost* Δ *k x B*
⟨*proof*⟩

lemma *exact-concrete-bmssp-zero-deterministic*:
assumes *run*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d′ B′ U c*
and *run′*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d″ B″ U′ c′*
shows $d' = d'' \wedge B' = B'' \wedge U = U' \wedge c = c'$
⟨*proof*⟩

lemma *exact-concrete-bmssp-zero-exists-iff-singleton*:
(∃ *d′ B′ U c*. *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d′ B′ U c*)

$\longleftrightarrow (\exists\, x.\ S = \{x\})$
⟨*proof*⟩

lemma *complete-on-dist*:
complete-on (*dist s*) *U*
⟨*proof*⟩

lemma *exact-concrete-partition-loop-state-done-witness*:
assumes *empty*: *keys-of* (*partition-state-view D*) = {}
and *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d′* (*bound-tree P B*)
shows *exact-concrete-partition-loop-state* Δ *M-of t h k cap l d P B d′*
D a [] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a B*])) *0* []
⟨*proof*⟩

lemma *exact-concrete-partition-loop-state-stop-witness*:
assumes *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d′* (*bound-tree P* (*Fin a*))
and *threshold*: $k * cap \leq$ *card* (*bound-tree P* (*Fin a*))
shows *exact-concrete-partition-loop-state* Δ *M-of t h k cap l d P B d′*
D a [] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)])) *0* []
⟨*proof*⟩

lemma *exact-concrete-partition-loop-state-done-exists*:
assumes *empty*: *keys-of* (*partition-state-view D*) = {}
and *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d′* (*bound-tree P B*)
shows ∃ *betas bs B′ Us U c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D a betas bs B′ Us U c child-costs
⟨*proof*⟩

lemma *exact-concrete-partition-loop-state-stop-exists*:
assumes *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d′* (*bound-tree P* (*Fin a*))
and *threshold*: $k * cap \leq$ *card* (*bound-tree P* (*Fin a*))
shows ∃ *betas bs B′ Us U c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D a betas bs B′ Us U c child-costs
⟨*proof*⟩

lemma *exact-concrete-partition-loop-state-step-from-concrete-ops*:
assumes *inv-D*: *partition-state-invar D*
and *upper-D*: *partition-upper-bound* (*partition-state-view D*) *Bmax*
and *pull*:
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*)
and *beta-bound*: *bound-le* (*Fin beta*) *B*

```
    and child-pre: bmssp-pre-full d S-pull (Fin beta)
    and S-pull-def: S-pull = split-below d P beta
    and S-pull-reaches: ∀x∈S-pull. reachable s x
    and a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
    and complete-prefix: complete-on d' (bound-tree P (Fin a))
    and below-threshold: card (bound-tree P (Fin a)) < k * cap
    and child:
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) U-child c-child
    and U-child: U-child = range-tree P a (Fin b)
    and preserved: complete-preserved d-child d' U-child
    and tail:
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child U-child b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child U-child beta B)
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail
  shows exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
    (beta # betas) (b # bs) B'
    (range-tree P a (Fin b) # Us-tail)
    (bound-tree P (Fin a) ∪
      ⋃(set (range-tree P a (Fin b) # Us-tail)))
    (card S-pull +
      t * length (edge-relaxation-pairs-in-bound d-child U-child beta B) +
      h * length (edge-relaxation-pairs-between d-child U-child b beta) +
      h * length (label-pairs-between d S-pull b beta) +
      c-child + c-tail)
    (c-child # child-costs-tail)
⟨proof⟩
```

The witness lemmas turn concrete operations into one exact loop step. The most explicit version assumes the concrete pull and the concrete tail state are already known, then constructs the exact step and computes its cost. The initialized version derives the pull split and child precondition from *exact-partition-initial-state*. Later existence lemmas use these builders to cover the terminal, threshold, and recursive cases of a non-base call.

```
lemma exact-concrete-initialized-partition-loop-state-step:
  assumes init: exact-partition-initial-state t d P D c-insert
    and pre: bmssp-pre-full d P (Fin Bmax)
    and upper: ⋀u. u ∈ P ⟹ d u < Bmax
    and pull: partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
    and beta-bound: bound-le (Fin beta) B
    and S-pull-reaches: ∀x∈S-pull. reachable s x
    and a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
```

and *complete-prefix*: *complete-on* d' (*bound-tree* P (*Fin* a))
and *below-threshold*: *card* (*bound-tree* P (*Fin* a)) $<$ $k * cap$
and *child*:
exact-concrete-bmssp Δ *M-of* t h k *cap* l d *S-pull* (*Fin* *beta*)
d-child (*Fin* b) *U-child* *c-child*
and *U-child*: *U-child* = *range-tree* P a (*Fin* b)
and *preserved*: *complete-preserved* *d-child* d' *U-child*
and *tail*:
exact-concrete-partition-loop-state Δ *M-of* t h k *cap* l d P B d'
(*partition-state-batch-prepend* (*label-pairs-between* d *S-pull* b *beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between* *d-child* *U-child* b *beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound* *d-child* *U-child* *beta* B)
D-pull)))
b *betas* *bs* B' *Us-tail* *U-tail* *c-tail* *child-costs-tail*
shows *exact-concrete-partition-loop-state* Δ *M-of* t h k *cap* l d P B d' D a
(*beta* # *betas*) (b # *bs*) B'
(*range-tree* P a (*Fin* b) # *Us-tail*)
(*bound-tree* P (*Fin* a) $\cup$
$\bigcup$(*set* (*range-tree* P a (*Fin* b) # *Us-tail*)))
(*card* *S-pull* +
$t * length$ (*edge-relaxation-pairs-in-bound* *d-child* *U-child* *beta* B) +
$h * length$ (*edge-relaxation-pairs-between* *d-child* *U-child* b *beta*) +
$h * length$ (*label-pairs-between* d *S-pull* b *beta*) +
c-child + *c-tail*)
(*c-child* # *child-costs-tail*)
⟨*proof*⟩

lemma *partition-state-pull-three-batch-prepend-invar*:
assumes *inv*: *partition-state-invar* D
and *pull*: *partition-state-pull* M B D = (*S-pull*, *beta*, *D-pull*)
shows *partition-state-invar*
(*partition-state-batch-prepend* *zs*
(*partition-state-batch-prepend* *ys*
(*partition-state-batch-prepend* *xs* *D-pull*)))
⟨*proof*⟩

lemma *exact-concrete-initialized-partition-loop-state-step-exists*:
assumes *init*: *exact-partition-initial-state* t d P D *c-insert*
and *pre*: *bmssp-pre-full* d P (*Fin* *Bmax*)
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < Bmax$
and *pull*: *partition-state-pull* (*M-of* l) *Bmax* D = (*S-pull*, *beta*, *D-pull*)
and *beta-bound*: *bound-le* (*Fin* *beta*) B
and *S-pull-reaches*: $\forall x \in$ *S-pull*. *reachable* s x
and *a-le-b*: $a \leq b$
and *a-bound*: *bound-le* (*Fin* a) B'
and *complete-prefix*: *complete-on* d' (*bound-tree* P (*Fin* a))
and *below-threshold*: *card* (*bound-tree* P (*Fin* a)) $<$ $k * cap$

and *child*:
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child*
and *U-child*: *U-child* = *range-tree P a* (*Fin b*)
and *preserved*: *complete-preserved d-child d′ U-child*
and *tail*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child U-child b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child U-child beta B*)
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail
shows $\exists$ *c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
(*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) $\cup$
$\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c child-costs*
⟨*proof*⟩

lemma *exact-concrete-initialized-partition-loop-state-step-exists-finite-bound*:
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < Bmax$
and *pull*: *partition-state-pull* (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*)
and *P-reaches*: $\forall x{\in}P.$ *reachable s x*
and *a-le-b*: $a \leq b$
and *a-bound*: *bound-le* (*Fin a*) *B′*
and *complete-prefix*: *complete-on d′* (*bound-tree P* (*Fin a*))
and *below-threshold*: *card* (*bound-tree P* (*Fin a*)) < *k* ∗ *cap*
and *child*:
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child*
and *preserved*:
complete-preserved d-child d′ (*range-tree P a* (*Fin b*))
and *tail*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*
(*range-tree P a* (*Fin b*)) *b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta* (*Fin Bmax*))
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail

shows $\exists\, c\ child\text{-}costs.$
$exact\text{-}concrete\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P$
$(Fin\ Bmax)\ d'\ D\ a\ (beta\ \#\ betas)\ (b\ \#\ bs)\ B'$
$(range\text{-}tree\ P\ a\ (Fin\ b)\ \#\ Us\text{-}tail)$
$(bound\text{-}tree\ P\ (Fin\ a)\ \cup$
$\bigcup(set\ (range\text{-}tree\ P\ a\ (Fin\ b)\ \#\ Us\text{-}tail)))\ c\ child\text{-}costs$
⟨*proof*⟩

lemma *exact-concrete-initialized-partition-loop-state-nonterminal-exists*:
assumes *init*: $exact\text{-}partition\text{-}initial\text{-}state\ t\ d\ P\ D\ c\text{-}insert$
and *pre*: $bmssp\text{-}pre\text{-}full\ d\ P\ (Fin\ Bmax)$
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < Bmax$
and *pull*: $partition\text{-}state\text{-}pull\ (M\text{-}of\ l)\ Bmax\ D = (S\text{-}pull,\ beta,\ D\text{-}pull)$
and *beta-bound*: $bound\text{-}le\ (Fin\ beta)\ B$
and *P-reaches*: $\forall\, x{\in}P.\ reachable\ s\ x$
and *complete-prefix*: $complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a))$
and *below-threshold*: $card\ (bound\text{-}tree\ P\ (Fin\ a)) < k * cap$
and *step*:
$\exists\, d\text{-}child\ b\ c\text{-}child\ betas\ bs\ B'\ Us\text{-}tail\ U\text{-}tail\ c\text{-}tail$
$child\text{-}costs\text{-}tail.$
$a \le b \wedge$
$bound\text{-}le\ (Fin\ a)\ B' \wedge$
$exact\text{-}concrete\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ S\text{-}pull\ (Fin\ beta)$
$d\text{-}child\ (Fin\ b)\ (range\text{-}tree\ P\ a\ (Fin\ b))\ c\text{-}child \wedge$
$complete\text{-}preserved\ d\text{-}child\ d'\ (range\text{-}tree\ P\ a\ (Fin\ b)) \wedge$
$exact\text{-}concrete\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P\ B\ d'$
$(partition\text{-}state\text{-}batch\text{-}prepend\ (label\text{-}pairs\text{-}between\ d\ S\text{-}pull\ b\ beta)$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(edge\text{-}relaxation\text{-}pairs\text{-}between\ d\text{-}child$
$(range\text{-}tree\ P\ a\ (Fin\ b))\ b\ beta)$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(edge\text{-}relaxation\text{-}pairs\text{-}in\text{-}bound\ d\text{-}child$
$(range\text{-}tree\ P\ a\ (Fin\ b))\ beta\ B)$
$D\text{-}pull)))$
$b\ betas\ bs\ B'\ Us\text{-}tail\ U\text{-}tail\ c\text{-}tail\ child\text{-}costs\text{-}tail$
shows $\exists\, betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
$exact\text{-}concrete\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P\ B\ d'$
$D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs$
⟨*proof*⟩

lemma *exact-concrete-initialized-partition-loop-state-nonterminal-exists-finite-bound*:
assumes *init*: $exact\text{-}partition\text{-}initial\text{-}state\ t\ d\ P\ D\ c\text{-}insert$
and *pre*: $bmssp\text{-}pre\text{-}full\ d\ P\ (Fin\ Bmax)$
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < Bmax$
and *pull*: $partition\text{-}state\text{-}pull\ (M\text{-}of\ l)\ Bmax\ D = (S\text{-}pull,\ beta,\ D\text{-}pull)$
and *P-reaches*: $\forall\, x{\in}P.\ reachable\ s\ x$
and *complete-prefix*: $complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a))$
and *below-threshold*: $card\ (bound\text{-}tree\ P\ (Fin\ a)) < k * cap$

and *step*:
$\exists$ *d-child b c-child betas bs B′ Us-tail U-tail c-tail*
child-costs-tail.
$a \leq b \wedge$
bound-le (*Fin a*) *B′* $\wedge$
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child* $\wedge$
complete-preserved d-child d′ (*range-tree P a* (*Fin b*)) $\wedge$
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*
(*range-tree P a* (*Fin b*)) *b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta* (*Fin Bmax*))
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail
shows $\exists$ *betas bs B′ Us U c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′ D a betas bs B′ Us U c child-costs*
⟨*proof*⟩

lemma *exact-concrete-initialized-partition-loop-state-cases-exists-finite-bound*:
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: $\bigwedge u.\ u \in P \Longrightarrow d\ u < Bmax$
and *P-reaches*: $\forall x \in P.$ *reachable s x*
and *cases*:
(*keys-of* (*partition-state-view D*) = {} $\wedge$
bound-le (*Fin a*) (*Fin Bmax*) $\wedge$
complete-on d′ (*bound-tree P* (*Fin Bmax*))) $\vee$
(*bound-le* (*Fin a*) (*Fin Bmax*) $\wedge$
complete-on d′ (*bound-tree P* (*Fin a*)) $\wedge$
$k * cap \leq$ *card* (*bound-tree P* (*Fin a*))) $\vee$
($\exists$ *S-pull beta D-pull*.
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*) $\wedge$
complete-on d′ (*bound-tree P* (*Fin a*)) $\wedge$
card (*bound-tree P* (*Fin a*)) $< k * cap \wedge$
($\exists$ *d-child b c-child betas bs B′ Us-tail U-tail c-tail*
child-costs-tail.
$a \leq b \wedge$
bound-le (*Fin a*) *B′* $\wedge$
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child* $\wedge$
complete-preserved d-child d′ (*range-tree P a* (*Fin b*)) $\wedge$
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′*

(*partition-state-batch-prepend*
(*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*
(*range-tree P a* (*Fin b*)) *b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta* (*Fin Bmax*))
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail))
shows ∃ *betas bs B′ Us U c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′ D a betas bs B′ Us U c child-costs*
⟨*proof*⟩

lemma *exact-concrete-initialized-partition-loop-state-cases-exists-finite-bound-with-complete*:
fixes *W* :: *bound* ⇒ *′a set*
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: ⋀*u*. *u* ∈ *P* ⟹ *d u* < *Bmax*
and *P-reaches*: ∀ *x*∈*P*. *reachable s x*
and *cases*:
(*keys-of* (*partition-state-view D*) = {} ∧
bound-le (*Fin a*) (*Fin Bmax*) ∧
complete-on d′ (*bound-tree P* (*Fin Bmax*)) ∧
complete-on d′ (*W* (*Fin Bmax*))) ∨
(*bound-le* (*Fin a*) (*Fin Bmax*) ∧
complete-on d′ (*bound-tree P* (*Fin a*)) ∧
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ∧
complete-on d′ (*W* (*Fin a*))) ∨
(∃ *S-pull beta D-pull B′*.
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*) ∧
complete-on d′ (*bound-tree P* (*Fin a*)) ∧
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ∧
complete-on d′ (*W B′*) ∧
(∃ *d-child b c-child betas bs Us-tail U-tail c-tail*
child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B′* ∧
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child* ∧
complete-preserved d-child d′ (*range-tree P a* (*Fin b*)) ∧
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′*
(*partition-state-batch-prepend*
(*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*

```
        (range-tree P a (Fin b)) b beta)
      (partition-state-batch-prepend
        (edge-relaxation-pairs-in-bound d-child
          (range-tree P a (Fin b)) beta (Fin Bmax))
        D-pull)))
    b betas bs B' Us-tail U-tail c-tail child-costs-tail))
 shows ∃ betas bs B' Us U c child-costs.
  exact-concrete-partition-loop-state Δ M-of t h k cap l d P
   (Fin Bmax) d' D a betas bs B' Us U c child-costs ∧
  complete-on d' (W B')
⟨proof⟩

lemma exact-concrete-initialized-partition-loop-state-cases-exists-with-complete:
 fixes W :: bound ⇒ 'a set
 assumes init: exact-partition-initial-state t d P D c-insert
  and pre: bmssp-pre-full d P (Fin Bmax)
  and upper: ⋀u. u ∈ P ⟹ d u < Bmax
  and P-reaches: ∀ x∈P. reachable s x
  and cases:
   (keys-of (partition-state-view D) = {} ∧
    bound-le (Fin a) B ∧
    complete-on d' (bound-tree P B) ∧
    complete-on d' (W B)) ∨
   (bound-le (Fin a) B ∧
    complete-on d' (bound-tree P (Fin a)) ∧
    k * cap ≤ card (bound-tree P (Fin a)) ∧
    complete-on d' (W (Fin a))) ∨
   (∃ S-pull beta D-pull B'.
    partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull) ∧
    bound-le (Fin beta) B ∧
    complete-on d' (bound-tree P (Fin a)) ∧
    card (bound-tree P (Fin a)) < k * cap ∧
    complete-on d' (W B') ∧
    (∃ d-child b c-child betas bs Us-tail U-tail c-tail
       child-costs-tail.
     a ≤ b ∧
     bound-le (Fin a) B' ∧
     exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
       d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
     complete-preserved d-child d' (range-tree P a (Fin b)) ∧
     exact-concrete-partition-loop-state Δ M-of t h k cap l d P
       B d'
       (partition-state-batch-prepend
         (label-pairs-between d S-pull b beta)
         (partition-state-batch-prepend
           (edge-relaxation-pairs-between d-child
             (range-tree P a (Fin b)) b beta)
           (partition-state-batch-prepend
             (edge-relaxation-pairs-in-bound d-child
```

```
                (range-tree P a (Fin b)) beta B)
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  shows ∃ betas bs B' Us U c child-costs.
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
      B d' D a betas bs B' Us U c child-costs ∧
    complete-on d' (W B')
⟨proof⟩

lemma exact-concrete-partition-loop-state-step-exists-from-concrete-ops:
  assumes inv-D: partition-state-invar D
    and upper-D: partition-upper-bound (partition-state-view D) Bmax
    and pull:
      partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
    and beta-bound: bound-le (Fin beta) B
    and child-pre: bmssp-pre-full d S-pull (Fin beta)
    and S-pull-def: S-pull = split-below d P beta
    and S-pull-reaches: ∀ x∈S-pull. reachable s x
    and a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
    and complete-prefix: complete-on d' (bound-tree P (Fin a))
    and below-threshold: card (bound-tree P (Fin a)) < k * cap
    and child:
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) U-child c-child
    and U-child: U-child = range-tree P a (Fin b)
    and preserved: complete-preserved d-child d' U-child
    and tail:
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child U-child b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child U-child beta B)
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail
  shows ∃ c child-costs.
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
      (beta # betas) (b # bs) B'
      (range-tree P a (Fin b) # Us-tail)
      (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # Us-tail))) c child-costs
⟨proof⟩

lemma exact-concrete-bmssp-step-with-exact-insert-cost:
  assumes init: exact-partition-initial-state t
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) D c-insert
    and loop:
```

exact-concrete-partition-loop-state Δ *M-of t h k cap l*
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) B d′ D a
betas bs B′ Us U-loop c-loop child-costs-loop
and *complete*:
complete-on d′
{v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v}
and *U*:
U = U-loop ∪
{v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v}
shows *exact-concrete-bmssp* Δ *M-of t h k cap (Suc l) d S B d′ B′ U*
(fp-iter-capped-scan-cost k cap d S S B +
t ∗ card (find-pivots-pivots-capped k cap d S B) + c-loop)
⟨*proof*⟩

lemma *exact-concrete-bmssp-Suc-exists-from-initial-loop*:
assumes *S-subset*: *S ⊆ V*
and *loop*:
⋀*D. exact-partition-initial-state t*
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) D
(t ∗ card (find-pivots-pivots-capped k cap d S B)) ⟹
∃ d′ a betas bs B′ Us U-loop c-loop child-costs-loop.
exact-concrete-partition-loop-state Δ *M-of t h k cap l*
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) B d′ D a
betas bs B′ Us U-loop c-loop child-costs-loop ∧
complete-on d′
{v ∈ bound-tree S B′.
find-pivots-label-capped k cap d S B v = dist s v}
shows *∃ d′ B′ U c.*
exact-concrete-bmssp Δ *M-of t h k cap (Suc l) d S B d′ B′ U c*
⟨*proof*⟩

The non-base existence lemmas are phrased as case combinators. They do not decide which branch an implementation should take; instead they say that if a client can provide one of the expected witnesses (empty partition, threshold, or a recursive child plus tail loop), then a full exact concrete BMSSP step exists. This keeps totality concerns separate from the correctness and cost refinement already proved for any exact run.

lemma *exact-concrete-bmssp-Suc-exists-from-initial-loop-cases*:
fixes *d* :: *'a ⇒ real*
and *S* :: *'a set*
and *B* :: *bound*
and *Bmax* :: *real*
and *k cap* :: *nat*
and *d-fp* :: *'a ⇒ real*

and $P :: 'a\ set$
defines *d-fp-def*:
$d\text{-}fp \equiv find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B$
and *P-def*:
$P \equiv find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B$
assumes *S-subset*: $S \subseteq V$
and *pre-fp*: $bmssp\text{-}pre\text{-}full\ d\text{-}fp\ P\ (Fin\ Bmax)$
and *upper*: $\bigwedge x.\ x \in P \Longrightarrow d\text{-}fp\ x < Bmax$
and *P-reaches*: $\forall x{\in}P.\ reachable\ s\ x$
and *cases*:
$\bigwedge D.\ exact\text{-}partition\text{-}initial\text{-}state\ t\ d\text{-}fp\ P\ D\ (t * card\ P) \Longrightarrow$
$\exists d'\ a.$
$(keys\text{-}of\ (partition\text{-}state\text{-}view\ D) = \{\} \wedge$
$bound\text{-}le\ (Fin\ a)\ B \wedge$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ B) \wedge$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B.\ d\text{-}fp\ v = dist\ s\ v\}) \vee$
$(bound\text{-}le\ (Fin\ a)\ B \wedge$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a)) \wedge$
$k * cap \leq card\ (bound\text{-}tree\ P\ (Fin\ a)) \wedge$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ (Fin\ a).\ d\text{-}fp\ v = dist\ s\ v\}) \vee$
$(\exists S\text{-}pull\ beta\ D\text{-}pull\ B'.$
$partition\text{-}state\text{-}pull\ (M\text{-}of\ l)\ Bmax\ D = (S\text{-}pull,\ beta,\ D\text{-}pull) \wedge$
$bound\text{-}le\ (Fin\ beta)\ B \wedge$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a)) \wedge$
$card\ (bound\text{-}tree\ P\ (Fin\ a)) < k * cap \wedge$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ d\text{-}fp\ v = dist\ s\ v\} \wedge$
$(\exists d\text{-}child\ b\ c\text{-}child\ betas\ bs\ Us\text{-}tail\ U\text{-}tail\ c\text{-}tail$
$child\text{-}costs\text{-}tail.$
$a \leq b \wedge$
$bound\text{-}le\ (Fin\ a)\ B' \wedge$
$exact\text{-}concrete\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\text{-}fp\ S\text{-}pull$
$(Fin\ beta)\ d\text{-}child\ (Fin\ b)\ (range\text{-}tree\ P\ a\ (Fin\ b))$
$c\text{-}child \wedge$
$complete\text{-}preserved\ d\text{-}child\ d'\ (range\text{-}tree\ P\ a\ (Fin\ b)) \wedge$
$exact\text{-}concrete\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$d\text{-}fp\ P\ B\ d'$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(label\text{-}pairs\text{-}between\ d\text{-}fp\ S\text{-}pull\ b\ beta)$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(edge\text{-}relaxation\text{-}pairs\text{-}between\ d\text{-}child$
$(range\text{-}tree\ P\ a\ (Fin\ b))\ b\ beta)$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(edge\text{-}relaxation\text{-}pairs\text{-}in\text{-}bound\ d\text{-}child$
$(range\text{-}tree\ P\ a\ (Fin\ b))\ beta\ B)$
$D\text{-}pull)))$
$b\ betas\ bs\ B'\ Us\text{-}tail\ U\text{-}tail\ c\text{-}tail\ child\text{-}costs\text{-}tail))$

shows $\exists\, d'\ B'\ U\ c.$
$\quad$ *exact-concrete-bmssp* Δ *M-of t h k cap* (*Suc l*) *d S B d' B' U c*
⟨*proof*⟩

lemma *exact-concrete-bmssp-Suc-exists-from-finite-initial-loop-cases*:
fixes $d :: 'a \Rightarrow real$
and $S :: 'a\ set$
and $Bmax :: real$
and $k\ cap :: nat$
and $d\text{-}fp :: 'a \Rightarrow real$
and $P :: 'a\ set$
defines *d-fp-def*:
$d\text{-}fp \equiv find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ (Fin\ Bmax)$
and *P-def*:
$P \equiv find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ (Fin\ Bmax)$
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S* (*Fin Bmax*)
and *S-reaches*: $\forall\, x{\in}S.\ reachable\ s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow d\ x < Bmax$
and *cases*:
$\bigwedge D.\ exact\text{-}partition\text{-}initial\text{-}state\ t\ d\text{-}fp\ P\ D\ (t * card\ P) \Longrightarrow$
$\exists\, d'\ a.$
(*keys-of* (*partition-state-view D*) = {} ∧
bound-le (*Fin a*) (*Fin Bmax*) ∧
complete-on d' (*bound-tree P* (*Fin Bmax*)) ∧
complete-on d'
{*v* ∈ *bound-tree S* (*Fin Bmax*). *d-fp v* = *dist s v*}) ∨
(*bound-le* (*Fin a*) (*Fin Bmax*) ∧
complete-on d' (*bound-tree P* (*Fin a*)) ∧
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ∧
complete-on d'
{*v* ∈ *bound-tree S* (*Fin a*). *d-fp v* = *dist s v*}) ∨
(∃ *S-pull beta D-pull B'*.
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*) ∧
complete-on d' (*bound-tree P* (*Fin a*)) ∧
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ∧
complete-on d'
{*v* ∈ *bound-tree S B'*. *d-fp v* = *dist s v*} ∧
(∃ *d-child b c-child betas bs Us-tail U-tail c-tail*
child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B'* ∧
exact-concrete-bmssp Δ *M-of t h k cap l d-fp S-pull*
(*Fin beta*) *d-child* (*Fin b*) (*range-tree P a* (*Fin b*))
c-child ∧
complete-preserved d-child d' (*range-tree P a* (*Fin b*)) ∧
exact-concrete-partition-loop-state Δ *M-of t h k cap l*
d-fp P (*Fin Bmax*) *d'*
(*partition-state-batch-prepend*

```
                (label-pairs-between d-fp S-pull b beta)
                (partition-state-batch-prepend
                  (edge-relaxation-pairs-between d-child
                    (range-tree P a (Fin b)) b beta)
                  (partition-state-batch-prepend
                    (edge-relaxation-pairs-in-bound d-child
                      (range-tree P a (Fin b)) beta (Fin Bmax))
                    D-pull)))
              b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S (Fin Bmax)
      d' B' U c
⟨proof⟩

lemma exact-concrete-bmssp-Suc-exists-if-pivots-empty:
  assumes S-subset: S ⊆ V
    and a-bound: bound-le (Fin a) B
    and pivots-empty:
      find-pivots-pivots-capped k cap d S B = {}
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
⟨proof⟩

lemma exact-concrete-bmssp-Suc-exists-if-initial-threshold:
  assumes S-subset: S ⊆ V
    and a-bound: bound-le (Fin a) B
    and threshold:
      k * cap ≤
        card (bound-tree (find-pivots-pivots-capped k cap d S B) (Fin a))
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
⟨proof⟩

lemma exact-concrete-bmssp-Suc-exists-if-pivots-empty-same-bound:
  assumes S-subset: S ⊆ V
    and a-bound: bound-le (Fin a) B
    and pivots-empty:
      find-pivots-pivots-capped k cap d S B = {}
  shows ∃ d' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B U c
⟨proof⟩

lemma exact-concrete-bmssp-Suc-exists-if-initial-threshold-bound:
  assumes S-subset: S ⊆ V
    and a-bound: bound-le (Fin a) B
    and threshold:
      k * cap ≤
        card (bound-tree (find-pivots-pivots-capped k cap d S B) (Fin a))
  shows ∃ d' U c.
```

exact-concrete-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B*
d′ (*Fin a*) *U c*
⟨*proof*⟩

The central refinement theorem forgets the concrete partition state and keeps only its abstract view. Concrete pulls refine to *pull-separates* and concrete batch prepends refine to *batch-min-update* with the expected cost predicates. After this projection, the exact run is a direct-insert costed run with the same labels, bounds, output sets, and total cost.

The final two theorems are immediate consequences of that projection: exact concrete BMSSP runs inherit the success-or-threshold property and the full BMSSP correctness theorem from the direct-insert layer.

theorem *exact-concrete-refines-direct-insert*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′*
(*partition-state-view D*) *a betas bs B′ Us U c child-costs*
and *exact-concrete-bmssp-refines-direct-insert*:
exact-concrete-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
⟨*proof*⟩

theorem *exact-concrete-bmssp-Suc-success-or-threshold*:
assumes *run*:
exact-concrete-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $B' = B \vee k * cap \leq$ *card U*
⟨*proof*⟩

theorem *exact-concrete-bmssp-correct*:
assumes *run*:
exact-concrete-bmssp Δ *M-of t h k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *bmssp-post-full d S B d′ B′ U*
⟨*proof*⟩

end

end
theory *BMSSP-Top-Level-Bounds*
imports *BMSSP-Strict-Tie-Breaking BMSSP-Exact-Concrete-Cost*
begin

36 Top-Level BMSSP Theorems

This theory is the entry point for the checked correctness result for the top-level BMSSP algorithm modelled in this development. Everything before this point proves local correctness and local cost statements: FindPivots preserves the complete-source invariant, the partition loop realizes the recursive BMSSP step, and the concrete partition operations account for Insert, BatchPrepend, and Pull. This file is where those ingredients are closed over a root call and converted into the single-source shortest-path headline.

The algorithmic run relations used here are the strongest ones in the project. They record the capped FindPivots step, the exact pulled child source, the range-synchronised recursive calls, and the separated edge and source batch costs. The early theorems therefore look a little verbose: their purpose is to expose a precise bridge from the executable recurrence to the abstract correctness theorem while retaining enough cost structure for the asymptotic proof.

The locale assumption is the paper's tie-breaking consequence: shortest paths form a strict tree order. Under that assumption, the proof instantiates the BMSSP parameters with *sssp-log-one-third-param* and *sssp-log-two-thirds-param*. These correspond to the paper's $log^{\frown}\{1/3\}\ n$ and $log^{\frown}\{2/3\}\ n$ scales. The level capacity is computed by *bmssp-level-cap*; the point of the schedule is that the recursive tree depth, pivot growth, and bucketed partition cost telescope into the target $O(m * log^{\frown}\{2/3\}\ n)$ envelope.

The file proceeds in four stages. First, it proves root-level correctness for operational and costed runs. Second, it specializes the refined graph bound to the logarithmic parameter schedule. Third, it packages the least top-level execution cost as a function and proves Big-O bounds for it. Finally, it exposes locale-level headline theorems that downstream theories can cite without reopening the cost recurrence.

context *strict-tie-breaking-digraph*
begin

theorem *operational-range-split-algorithm-correct*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *run*:
      full-operational-bmssp k cap l
        finite-initial-label $\{s\}$ *Infinity d′ Infinity U*
  shows $U = V \wedge$ *sssp-correct d′*
⟨*proof*⟩

theorem *operational-range-split-algorithm-reachable-correct*:
  assumes *run*:
      full-operational-bmssp k cap l
        finite-initial-label $\{s\}$ *Infinity d′ Infinity U*
  shows *sssp-correct d′*

⟨*proof*⟩

theorem *range-split-algorithm-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows $U = V \wedge$ *sssp-correct* d'
⟨*proof*⟩

theorem *range-split-algorithm-reachable-correct*:
assumes *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows *sssp-correct* d'
⟨*proof*⟩

theorem *direct-insert-algorithm-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows $U = V \wedge$ *sssp-correct* d'
⟨*proof*⟩

theorem *direct-insert-algorithm-reachable-correct*:
assumes *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows *sssp-correct* d'
⟨*proof*⟩

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-under-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$

(*bmssp-level-cap k t l*) *l'* *d* *P* *B* *d'* *D* *a* *betas* *bs* *B'*
Us *U-loop* *c-loop* *child-costs* ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *H* ∗ *card* (*outgoing-edges U-loop*)
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d'* *Infinity* *U c*
shows *U* = *V* ∧ *sssp-correct d'* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-trivial-edge-budget*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *insert-scaled-factor*: *t* ≤ *A-insert* ∗ *k*
and *seen-scaled-factor*: *k* ∗ Δ + *A-insert* ≤ *2* ∗ *A*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d'* *Infinity* *U c*
shows *U* = *V* ∧ *sssp-correct d'* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t* + *h*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-amortized*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *insert-scaled-factor*: *t* ≤ *A-insert* ∗ *k*
and *seen-scaled-factor*: *k* ∗ Δ + *A-insert* ≤ *2* ∗ *A*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*

and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *h*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

The preceding theorems keep the parameters abstract. They are useful because they show exactly which inequalities are required of a costed BMSSP run: outdegree must be bounded, Insert must be charged at the chosen parameter scale, and the source and seen-set terms must fit into the refined graph bound *bmssp-refined-graph-time-bound*. The amortized variant is the one that matches the bucketed partition analysis: the split work is not charged to each operation eagerly, but through the aggregate accounting used by the partition interface.

The next group substitutes the logarithmic schedule. The one-third parameter controls the number of levels and source growth, while the two-thirds parameter controls the Insert scale supplied to the partition layer. The simple arithmetic lemmas in *sssp-log-one-third-param* and *sssp-log-two-thirds-param* prove that these choices satisfy the abstract side conditions.

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-log-params-bounded-degree*:
fixes *N p q* :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap p q*) *q p p*
(*bmssp-level-cap p q p*) *p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc* $\Delta * p$) (λ-. *q*) (λ-. *p*)
(λ-. *p*) (λ-. *q*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*:
fixes *N D p q* :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*:
direct-insert-costed-bmssp D (*bmssp-level-cap p q*) *q p p*

```
      (bmssp-level-cap p q p) p
      finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc D * p) (λ-. q) (λ-. p)
      (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
⟨proof⟩

theorem exact-concrete-algorithm-correct-and-refined-graph-bound-log-params-fixed-
degree:
  fixes N D p q :: nat
  defines p-def: p ≡ sssp-log-one-third-param N
    and q-def: q ≡ sssp-log-two-thirds-param N
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le D
    and run:
      exact-concrete-bmssp D (bmssp-level-cap p q) q p p
        (bmssp-level-cap p q p) p
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc D * p) (λ-. q) (λ-. p)
      (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
⟨proof⟩
```

The definitions below package the root execution as an ordinary total cost predicate. *exact-concrete-top-level-run* fixes the root source set to $\{s\}$ and the input bound to *Infinity*; the auxiliary *exact-concrete-root-run* keeps the returned bound visible for existence proofs that reason about threshold cases. The least-cost wrapper *exact-concrete-top-level-time* is not an algorithm in its own right; it is a mathematical way to speak about the cost of any concrete run satisfying the checked run relation.

```
definition exact-concrete-top-level-run ::
  nat ⇒ nat ⇒ ('a ⇒ real) ⇒ 'a set ⇒ nat ⇒ bool where
  exact-concrete-top-level-run D N d' U c ⟷
    (let p = sssp-log-one-third-param N;
         q = sssp-log-two-thirds-param N
     in exact-concrete-bmssp D (bmssp-level-cap p q) q p p
       (bmssp-level-cap p q p) p
       finite-initial-label {s} Infinity d' Infinity U c)

definition exact-concrete-root-run ::
  nat ⇒ nat ⇒ ('a ⇒ real) ⇒ bound ⇒ 'a set ⇒ nat ⇒ bool where
  exact-concrete-root-run D N d' B' U c ⟷
    (let p = sssp-log-one-third-param N;
         q = sssp-log-two-thirds-param N
     in exact-concrete-bmssp D (bmssp-level-cap p q) q p p
       (bmssp-level-cap p q p) p
       finite-initial-label {s} Infinity d' B' U c)
```

lemma *exact-concrete-top-level-run-root-run-iff*:
exact-concrete-top-level-run D N d′ U c ⟷
exact-concrete-root-run D N d′ Infinity U c
⟨*proof*⟩

definition *exact-concrete-top-level-cost* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
exact-concrete-top-level-cost D N c ⟷
(∃ *d′ U*. *exact-concrete-top-level-run D N d′ U c*)

definition *exact-concrete-top-level-time* :: *nat* ⇒ *nat* ⇒ *nat* **where**
exact-concrete-top-level-time D N =
(*LEAST c*. *exact-concrete-top-level-cost D N c*)

lemma *exact-concrete-top-level-cost-exists-if-top-pivots-empty*:
assumes *pivots-empty*:
find-pivots-pivots-capped (*sssp-log-one-third-param N*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
finite-initial-label {*s*} *Infinity* = {}
shows ∃ *c*. *exact-concrete-top-level-cost D N c*
⟨*proof*⟩

lemma *exact-concrete-top-level-cost-exists-from-initial-loop*:
fixes *D N p q cap l* :: *nat*
and *d-fp* :: *′a* ⇒ *real*
and *P* :: *′a set*
defines *p-def*: *p* ≡ *sssp-log-one-third-param N*
and *q-def*: *q* ≡ *sssp-log-two-thirds-param N*
and *cap-def*: *cap* ≡ *bmssp-level-cap p q p*
and *l-def*: *l* ≡ *p* − *1*
and *d-fp-def*:
d-fp ≡ *find-pivots-label-capped p cap finite-initial-label* {*s*}
Infinity
and *P-def*:
P ≡ *find-pivots-pivots-capped p cap finite-initial-label* {*s*}
Infinity
assumes *loop*:
⋀*D-part*. *exact-partition-initial-state q d-fp P D-part*
(*q* ∗ *card P*) ⟹
∃ *d′ a betas bs Us U-loop c-loop child-costs-loop*.
exact-concrete-partition-loop-state D (*bmssp-level-cap p q*)
q p p cap l d-fp P Infinity d′ D-part a betas bs Infinity
Us U-loop c-loop child-costs-loop ∧
complete-on d′
{*v* ∈ *bound-tree* {*s*} *Infinity*. *d-fp v* = *dist s v*}
shows ∃ *c*. *exact-concrete-top-level-cost D N c*
⟨*proof*⟩

lemma *exact-concrete-top-level-cost-exists-from-root-run-below-threshold*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *below-threshold*:
vertex-count $<$
sssp-log-one-third-param $N *$
bmssp-level-cap (*sssp-log-one-third-param* N)
(*sssp-log-two-thirds-param* N) (*sssp-log-one-third-param* N)
and *run*: *exact-concrete-root-run* $D\ N\ d'\ B'\ U\ c$
shows $\exists c.$ *exact-concrete-top-level-cost* $D\ N\ c$
⟨*proof*⟩

lemma *eventually-exact-concrete-top-level-cost-if-root-runs-exist-below-threshold*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *below-threshold*:
eventually
($\lambda N.$ *vertex-count* $<$
sssp-log-one-third-param $N *$
bmssp-level-cap (*sssp-log-one-third-param* N)
(*sssp-log-two-thirds-param* N) (*sssp-log-one-third-param* N))
at-top
and *root-runs*:
eventually
($\lambda N.\ \exists d'\ B'\ U\ c.$ *exact-concrete-root-run* $D\ N\ d'\ B'\ U\ c$)
at-top
shows *eventually* ($\lambda N.\ \exists c.$ *exact-concrete-top-level-cost* $D\ N\ c$) *at-top*
⟨*proof*⟩

lemma *eventually-exact-concrete-top-level-cost-from-empty-top-pivots*:
eventually ($\lambda N.\ \exists c.$ *exact-concrete-top-level-cost* $D\ N\ c$) *at-top*
⟨*proof*⟩

lemma *exact-concrete-top-level-time-witness*:
assumes *exact-concrete-top-level-cost* $D\ N\ c$
shows *exact-concrete-top-level-cost* $D\ N$
(*exact-concrete-top-level-time* $D\ N$)
⟨*proof*⟩

lemma *exact-concrete-top-level-time-le*:
assumes *exact-concrete-top-level-cost* $D\ N\ c$
shows *exact-concrete-top-level-time* $D\ N \leq c$
⟨*proof*⟩

lemma *eventually-exact-concrete-top-level-cost-if-all*:
assumes $\bigwedge N.\ \exists c.$ *exact-concrete-top-level-cost* $D\ N\ c$
shows *eventually* ($\lambda N.\ \exists c.$ *exact-concrete-top-level-cost* $D\ N\ c$) *at-top*
⟨*proof*⟩

lemma *eventually-exact-concrete-top-level-cost-square-if-all*:
assumes $\bigwedge N.\ \exists c.$ *exact-concrete-top-level-cost* $D\ N\ c$

shows *eventually*
(λ*N*. ∃ *c*. *exact-concrete-top-level-cost D* (*N* ∗ *N*) *c*) *at-top*
⟨*proof*⟩

theorem *exact-concrete-top-level-run-correct-and-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*: *exact-concrete-top-level-run D N d′ U c*
shows *U* = *V* ∧ *sssp-correct d′* ∧
c ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *exact-concrete-top-level-cost-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *cost*: *exact-concrete-top-level-cost D N c*
shows *c* ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *exact-concrete-top-level-time-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run-exists*: *exact-concrete-top-level-cost D N c*
shows *exact-concrete-top-level-time D N* ≤
bmssp-refined-graph-time-bound
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

From this point onward, the file is purely asymptotic. The function *T-bmssp* abbreviates the least top-level concrete cost, and the following theorems compare it with increasingly convenient targets. The central comparison

target is *sssp-time-target*, which expands to an edge-count term multiplied by the two-thirds logarithmic factor. The proofs reuse the complexity lemmas for *bmssp-refined-graph-time-bound*; no algorithmic invariant is reproved in this section.

definition *T-bmssp* :: *nat* ⇒ *nat* ⇒ *nat* **where**
T-bmssp D N = *exact-concrete-top-level-time D N*

theorem *T-bmssp-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run-exists*: *exact-concrete-top-level-cost D N c*
shows *T-bmssp D N* ≤
bmssp-refined-graph-time-bound
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *T-bmssp-bigo-sssp-time-target-log-params-fixed-degree-if-exact-runs-exist*:
fixes *m* :: *nat* ⇒ *nat*
and *Cn Cm* :: *real*
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *Cn-pos*: *0* < *Cn*
and *Cm-pos*: *0* < *Cm*
and *vertex-count-dominated*:
eventually (λ*n*. *real vertex-count* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *edge-count-dominated*:
eventually (λ*n*. *real edge-count* ≤ *Cm* ∗ *real* (*m n*)) *at-top*
and *exact-runs*:
eventually (λ*n*. ∃ *c*. *exact-concrete-top-level-cost D n c*) *at-top*
shows (λ*n*. *real* (*T-bmssp D n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
⟨*proof*⟩

theorem *T-bmssp-bigo-sssp-time-target-square-size-params-fixed-degree-if-exact-runs-exist*:
fixes *m* :: *nat* ⇒ *nat*
and *Cn Cm* :: *real*
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *Cn-pos*: *0* < *Cn*
and *Cm-pos*: *0* < *Cm*
and *vertex-count-dominated*:
eventually (λ*n*. *real vertex-count* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *edge-count-dominated*:
eventually (λ*n*. *real edge-count* ≤ *Cm* ∗ *real* (*m n*)) *at-top*

and *exact-runs*:
eventually ($\lambda n.$ $\exists\, c.$ *exact-concrete-top-level-cost* D $(n * n)$ c) *at-top*
shows ($\lambda n.$ *real* (*T-bmssp* D $(n * n)$)) $\in$
$O(\lambda n.$ *sssp-time-target* m $n)$
⟨*proof*⟩

theorem *T-bmssp-bigo-edge-count-target-log-params-fixed-degree-if-exact-runs-exist*:
assumes *all-reachable*: $\bigwedge v.$ $v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* D
and *edge-count-pos*: $0 <$ *edge-count*
and *exact-runs*:
eventually ($\lambda n.$ $\exists\, c.$ *exact-concrete-top-level-cost* D n c) *at-top*
shows ($\lambda n.$ *real* (*T-bmssp* D n)) $\in$
$O(\lambda n.$ *sssp-time-target* (λ-. *edge-count*) $n)$
⟨*proof*⟩

theorem *T-bmssp-bigo-edge-count-target-log-params-fixed-degree*:
assumes *all-reachable*: $\bigwedge v.$ $v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* D
and *edge-count-pos*: $0 <$ *edge-count*
shows ($\lambda n.$ *real* (*T-bmssp* D n)) $\in$
$O(\lambda n.$ *sssp-time-target* (λ-. *edge-count*) $n)$
⟨*proof*⟩

theorem *T-bmssp-bigo-size-target-log-params-fixed-degree*:
assumes *all-reachable*: $\bigwedge v.$ $v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* D
shows ($\lambda n.$ *real* (*T-bmssp* D n)) $\in$
$O(\lambda n.$ *sssp-time-target* (λ-. *Suc edge-count*) $n)$
⟨*proof*⟩

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-exact-runs-exist*:
assumes *all-reachable*: $\bigwedge v.$ $v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* D
and *edge-count-pos*: $0 <$ *edge-count*
and *exact-runs*:
eventually ($\lambda n.$ $\exists\, c.$ *exact-concrete-top-level-cost* D n c) *at-top*
shows ($\lambda n.$ *real* (*T-bmssp* D n)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real* n + 2)) *powr* (2 / 3))
⟨*proof*⟩

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-root-runs-exist-below-threshold*:
assumes *all-reachable*: $\bigwedge v.$ $v \in V \Longrightarrow$ *reachable* s v
and *degree*: *edge-outdegree-le* D
and *edge-count-pos*: $0 <$ *edge-count*
and *below-threshold*:
eventually
($\lambda N.$ *vertex-count* $<$

sssp-log-one-third-param N ∗
bmssp-level-cap (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
at-top
and *root-runs*:
eventually
(λ*N*. ∃ *d′ B′ U c*. *exact-concrete-root-run D N d′ B′ U c*)
at-top
shows (λ*n*. *real* (*T-bmssp D n*)) ∈
O(λ*n*. *real edge-count* ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-root-runs-exist*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* < *edge-count*
and *root-runs*:
eventually
(λ*N*. ∃ *d′ B′ U c*. *exact-concrete-root-run D N d′ B′ U c*)
at-top
shows (λ*n*. *real* (*T-bmssp D n*)) ∈
O(λ*n*. *real edge-count* ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

theorem *T-bmssp-bigo-target-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* < *edge-count*
shows (λ*n*. *real* (*T-bmssp D n*)) ∈
O(λ*n*. *real edge-count* ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

theorem *bmssp-runtime-bigo-target*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* < *edge-count*
shows (λ*n*. *real* (*T-bmssp D n*)) ∈
O(λ*n*. *real edge-count* ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-total*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* < *edge-count*
and *total*: ⋀*N*. ∃ *c*. *exact-concrete-top-level-cost D N c*
shows (λ*n*. *real* (*T-bmssp D n*)) ∈
O(λ*n*. *real edge-count* ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

theorem *T-bmssp-bigo-edge-count-target-square-size-params-fixed-degree-if-exact-runs-exist*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *edge-count-pos*: *0 < edge-count*
    and *exact-runs*:
      eventually ($\lambda n.\ \exists c.$ *exact-concrete-top-level-cost D* $(n * n)$ *c*) *at-top*
  shows ($\lambda n.$ *real* (*T-bmssp D* $(n * n)$)) $\in$
    $O(\lambda n.$ *sssp-time-target* (λ-. *edge-count*) *n*)
⟨*proof*⟩

theorem *T-bmssp-bigo-target-square-size-params-fixed-degree-if-exact-runs-exist*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *edge-count-pos*: *0 < edge-count*
    and *exact-runs*:
      eventually ($\lambda n.\ \exists c.$ *exact-concrete-top-level-cost D* $(n * n)$ *c*) *at-top*
  shows ($\lambda n.$ *real* (*T-bmssp D* $(n * n)$)) $\in$
    $O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
  ⟨*proof*⟩

theorem *T-bmssp-bigo-target-square-size-params-fixed-degree-if-total*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *edge-count-pos*: *0 < edge-count*
    and *total*: $\bigwedge N.\ \exists c.$ *exact-concrete-top-level-cost D N c*
  shows ($\lambda n.$ *real* (*T-bmssp D* $(n * n)$)) $\in$
    $O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
  ⟨*proof*⟩

theorem *range-split-algorithm-correct-and-closed-graph-bound*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le* Δ
    and *degree-factor*: $\Delta \leq A$
    and *R-pos*: *0 < R*
    and *insert-factor*: $t \leq A * k$
    and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
    and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
    and *source-factor*: *Suc h* $\leq 2 * A$
    and *k-pos*: *0 < k*
    and *run*:
      exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
        (*bmssp-level-cap k t l*) *l*
        finite-initial-label {*s*} *Infinity d′ Infinity U c*
  shows $U = V \wedge$ *sssp-correct* $d' \wedge$
    $c \leq$ *bmssp-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *l*) (λ-. *t*)
      (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *range-split-algorithm-correct-and-refined-graph-bound-under-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) $l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'$
$Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$t *$ *sum-list* (*map card* (*range-tree-child-edge-range-list* $P\ a\ betas\ bs$))
$\leq H *$ *card* (*outgoing-edges* *U-loop*)
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows $U = V \wedge$ *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. A) (λ-. R) (λ-. H)
(λ-. l) (λ-. t) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *range-split-algorithm-correct-and-refined-graph-bound-trivial-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows $U = V \wedge$ *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. A) (λ-. R) (λ-. t)
(λ-. l) (λ-. t) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

end

context *positive-unique-shortest-digraph*
begin

theorem *positive-weight-direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*:
  fixes *N D p q* :: *nat*
  defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
    and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *run*:
      direct-insert-costed-bmssp D (*bmssp-level-cap p q*) *q p p*
        (*bmssp-level-cap p q p*) *p*
        finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
  shows $U = V \wedge$ *sssp-correct d'* $\wedge$
    $c \leq$ *bmssp-refined-graph-time-bound*
      (λ-. *Suc D* $*$ *p*) (λ-. *q*) (λ-. *p*)
      (λ-. *p*) (λ-. *q*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *positive-weight-exact-concrete-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*:
  fixes *N D p q* :: *nat*
  defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
    and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *run*:
      exact-concrete-bmssp D (*bmssp-level-cap p q*) *q p p*
        (*bmssp-level-cap p q p*) *p*
        finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
  shows $U = V \wedge$ *sssp-correct d'* $\wedge$
    $c \leq$ *bmssp-refined-graph-time-bound*
      (λ-. *Suc D* $*$ *p*) (λ-. *q*) (λ-. *p*)
      (λ-. *p*) (λ-. *q*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

end

locale *bounded-reduced-positive-instance* = *positive-unique-shortest-digraph* +
  fixes *D* :: *nat*
  assumes *all-vertices-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *bounded-edge-outdegree*: *edge-outdegree-le D*
begin

theorem *bounded-reduced-direct-insert-algorithm-correct-and-refined-graph-bound-log-params*:
  fixes *N p q* :: *nat*

defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *run*:
direct-insert-costed-bmssp D (*bmssp-level-cap p q*) *q p p*
(*bmssp-level-cap p q p*) *p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct d′* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* $*$ *p*) (λ-. *q*) (λ-. *p*)
(λ-. *p*) (λ-. *q*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *bounded-reduced-exact-concrete-algorithm-correct-and-refined-graph-bound-log-params*:
fixes *N p q* :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *run*:
exact-concrete-bmssp D (*bmssp-level-cap p q*) *q p p*
(*bmssp-level-cap p q p*) *p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct d′* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* $*$ *p*) (λ-. *q*) (λ-. *p*)
(λ-. *p*) (λ-. *q*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

theorem *bounded-reduced-bmssp-runtime-bigo-target*:
assumes *edge-count-pos*: *0* $<$ *edge-count*
shows (λ*n*. *real* (*T-bmssp D n*)) $\in$
O(λ*n*. *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

end

locale *bmssp-runtime-headline-instance* =
bounded: *bounded-reduced-positive-instance V E w s D*
for *V* :: *′v set*
and *E* :: (*′v* × *′v*) *set*
and *w* :: *′v* ⇒ *′v* ⇒ *real*
and *s* :: *′v*
and *D* :: *nat*
begin

This locale is the public asymptotic headline for the top-level theory. It hides the reduced positive instance, the fixed outdegree bound, and the least-cost definition behind two simple functions, *T-bmssp* for the checked BMSSP running time and *m* for the edge-count scale. The theorem below is deliberately assumption-free inside the locale: all graph hypotheses are

already part of the locale interpretation.

definition *T-bmssp* :: *nat* ⇒ *nat* **where**
T-bmssp N = *bounded.T-bmssp D N*

definition *m* :: *nat* ⇒ *nat* **where**
m N = *Suc bounded.edge-count*

theorem *bmssp-runtime-bigo-target*:
shows (λ*n*. *real* (*T-bmssp n*)) ∈
O(λ*n*. *real* (*m n*) ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

The theorem (λ*n*. *real* (*T-bmssp n*)) ∈ *O*(λ*n*. *real* (*m n*) ∗ *ln* (*real n* + *2*) *powr* (*2* / *3*)) is the headline statement used by the rest of the AFP entry. Its right-hand side is the deterministic BMSSP target: graph-size work times *log*^{*2/3*}. The proof is short here because the recurrence has already been solved in the preceding fixed-degree theorem and the locale definitions only rename the time and size functions.

end

sublocale *bounded-reduced-positive-instance* <
runtime-headline: *bmssp-runtime-headline-instance V E w s D*
⟨*proof*⟩

context *bounded-reduced-positive-instance*
begin

theorem *bounded-reduced-runtime-headline-bmssp-runtime-bigo-target*:
shows (λ*n*. *real* (*runtime-headline.T-bmssp n*)) ∈
O(λ*n*. *real* (*runtime-headline.m n*) ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

end

locale *reduction-correctness-contract* =
original: *finite-weighted-digraph V E w s* +
reduced: *bounded-reduced-positive-instance V' E' w' s' D*
for *V* :: *'v set*
and *E* :: (*'v* × *'v*) *set*
and *w* :: *'v* ⇒ *'v* ⇒ *real*
and *s* :: *'v*
and *V'* :: *'r set*
and *E'* :: (*'r* × *'r*) *set*
and *w'* :: *'r* ⇒ *'r* ⇒ *real*
and *s'* :: *'r*
and *D* :: *nat* +
fixes *decode-label* :: (*'r* ⇒ *real*) ⇒ *'v* ⇒ *real*
assumes *transfer-sssp-correct*:

reduced.sssp-correct $d' \Longrightarrow$ *original.sssp-correct* (*decode-label* d')
begin

theorem *reduced-direct-insert-run-transfers-correctness-and-refined-bound*:
fixes N p q :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param* N
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param* N
assumes *run*:
reduced.direct-insert-costed-bmssp D (*bmssp-level-cap* p q) q p p
(*bmssp-level-cap* p q p) p
reduced.finite-initial-label $\{s'\}$ *Infinity* d' *Infinity* U c
shows *original.sssp-correct* (*decode-label* d') $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc* $D * p$) (λ-. q) (λ-. p)
(λ-. p) (λ-. q) (λ-. *reduced.edge-count*)
reduced.vertex-count
⟨*proof*⟩

theorem *reduced-exact-concrete-run-transfers-correctness-and-refined-bound*:
fixes N p q :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param* N
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param* N
assumes *run*:
reduced.exact-concrete-bmssp D (*bmssp-level-cap* p q) q p p
(*bmssp-level-cap* p q p) p
reduced.finite-initial-label $\{s'\}$ *Infinity* d' *Infinity* U c
shows *original.sssp-correct* (*decode-label* d') $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc* $D * p$) (λ-. q) (λ-. p)
(λ-. p) (λ-. q) (λ-. *reduced.edge-count*)
reduced.vertex-count
⟨*proof*⟩

theorem *reduced-exact-concrete-top-level-time-transfers-correctness-and-refined-bound*:
assumes *run-exists*: *reduced.exact-concrete-top-level-cost* D N c
shows $\exists\, d'\ U.$ *reduced.exact-concrete-top-level-run* D N d' U
(*reduced.exact-concrete-top-level-time* D N) $\wedge$
original.sssp-correct (*decode-label* d') $\wedge$
reduced.exact-concrete-top-level-time D N $\leq$
bmssp-refined-graph-time-bound
(λ-. *Suc* $D *$ *sssp-log-one-third-param* N)
(λ-. *sssp-log-two-thirds-param* N)
(λ-. *sssp-log-one-third-param* N)
(λ-. *sssp-log-one-third-param* N)
(λ-. *sssp-log-two-thirds-param* N)
(λ-. *reduced.edge-count*) *reduced.vertex-count*
⟨*proof*⟩

theorem *reduced-runtime-headline-bmssp-runtime-bigo-target*:
shows $(\lambda n.\ real\ (reduced.runtime\text{-}headline.T\text{-}bmssp\ n)) \in$
$O(\lambda n.\ real\ (reduced.runtime\text{-}headline.m\ n) *$
$(ln\ (real\ n + 2))\ powr\ (2\ /\ 3))$
⟨*proof*⟩

end

37 Reduction Contract for the Asymptotic Bound

The theorem below is the checked asymptotic contract needed from the graph reduction layer. Once a family of reduced instances supplies a fixed outdegree bound D, a linear edge-count domination for the vertex term, and the per-instance cost bound produced by the locale theorem above, the final SSSP target bound follows.

theorem *reduction-contract-bigo-sssp-time-target*:
fixes $D :: nat$
assumes *Cn-pos*: $0 < Cn$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \le Cn * real\ (m\ n))$ *at-top*
and *reduced-instance-cost-bound*:
eventually
$(\lambda n.\ T\ n \le$
bmssp-refined-graph-time-bound
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param m n) *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *reduction-contract-bigo-sssp-time-target-with-constant-slack*:
fixes $D :: nat$
assumes *Cn-pos*: $0 < Cn$
and *Ccost-pos*: $0 < Ccost$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \le Cn * real\ (m\ n))$ *at-top*
and *reduced-instance-cost-bound*:
eventually
$(\lambda n.\ real\ (T\ n) \le Ccost *$
real (*bmssp-refined-graph-time-bound*
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param m n)) *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
⟨*proof*⟩

theorem *reduction-contract-bigo-sssp-time-target-square-size-params*:
fixes $D :: nat$

```
  assumes Cn-pos: 0 < Cn
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and reduced-instance-cost-bound:
      eventually
        (λn. T n ≤
          bmssp-refined-graph-time-bound
            (λn. Suc D * sssp-log-one-third-param (n * n))
            (λn. sssp-log-two-thirds-param (n * n))
            (λn. sssp-log-one-third-param (n * n))
            (λn. sssp-log-one-third-param (n * n))
            (λn. sssp-log-two-thirds-param (n * n)) m n) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
  ⟨proof⟩

theorem reduction-contract-bigo-sssp-time-target-square-size-params-with-constant-
slack:
  fixes D :: nat
  assumes Cn-pos: 0 < Cn
    and Ccost-pos: 0 < Ccost
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and reduced-instance-cost-bound:
      eventually
        (λn. real (T n) ≤ Ccost *
          real (bmssp-refined-graph-time-bound
            (λn. Suc D * sssp-log-one-third-param (n * n))
            (λn. sssp-log-two-thirds-param (n * n))
            (λn. sssp-log-one-third-param (n * n))
            (λn. sssp-log-one-third-param (n * n))
            (λn. sssp-log-two-thirds-param (n * n)) m n)) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
  ⟨proof⟩

end
theory BMSSP-Executable-Refinement-Internal
  imports BMSSP-Executable-Base-Case BMSSP-Top-Level-Bounds
begin
```

38 Executable Graph Refinement Bridge

This theory starts the refinement bridge between the concrete executable entry point *bmssp-distances* and the locale-based BMSSP correctness stack. The executable graph is a list of natural-number edge triples; the abstract semantics expects a finite weighted digraph locale. The definitions below name the carrier, edge set, and weight function induced by the list representation, then package the assumptions needed before the executable loop can be related to the direct-insert BMSSP relation.

fun *nat-edge-source* :: *nat-edge* ⇒ *nat* **where**
nat-edge-source (*u*, *v*, *w*) = *u*

fun *nat-edge-target* :: *nat-edge* ⇒ *nat* **where**
nat-edge-target (*u*, *v*, *w*) = *v*

fun *nat-edge-weight* :: *nat-edge* ⇒ *nat* **where**
nat-edge-weight (*u*, *v*, *w*) = *w*

definition *nat-graph-edge-list* :: *nat-graph* ⇒ (*nat* × *nat*) *list* **where**
nat-graph-edge-list G = *map* (λ*e*. (*nat-edge-source e*, *nat-edge-target e*)) *G*

definition *nat-graph-edges* :: *nat-graph* ⇒ (*nat* × *nat*) *set* **where**
nat-graph-edges G = *set* (*nat-graph-edge-list G*)

definition *nat-graph-vertices* :: *nat-graph* ⇒ *nat set* **where**
nat-graph-vertices G = *set* (*map nat-edge-source G*) ∪ *set* (*map nat-edge-target G*)

definition *nat-graph-vertex-list* :: *nat-graph* ⇒ *nat list* **where**
nat-graph-vertex-list G =
sort (*remdups* (*map nat-edge-source G* @ *map nat-edge-target G*))

definition *nat-graph-weight* :: *nat-graph* ⇒ *nat* ⇒ *nat* ⇒ *real* **where**
nat-graph-weight G u v =
(*case map-of*
(*map* (λ*e*. ((*nat-edge-source e*, *nat-edge-target e*),
real (*nat-edge-weight e*))) *G*) (*u*, *v*) *of*
None ⇒ *0*
| *Some c* ⇒ *c*)

definition *nat-graph-total-weight* :: *nat-graph* ⇒ *nat* **where**
nat-graph-total-weight G = *sum-list* (*map nat-edge-weight G*)

definition *nat-graph-well-formed* :: *nat-graph* ⇒ *bool* **where**
nat-graph-well-formed G ⟷
distinct (*nat-graph-edge-list G*) ∧
nat-graph-total-weight G < *bmssp-infinity*

lemma *nat-graph-well-formed-distinct-edge-list*:
assumes *nat-graph-well-formed G*
shows *distinct* (*nat-graph-edge-list G*)
⟨*proof*⟩

definition *nat-graph-reachable* :: *nat-graph* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
nat-graph-reachable G a b ⟷
finite-weighted-digraph.reachable
(*nat-graph-vertices G*) (*nat-graph-edges G*) *a b*

definition *nat-graph-dist* :: *nat-graph* ⇒ *nat* ⇒ *nat* ⇒ *real* **where**
nat-graph-dist G a b =
finite-weighted-digraph.dist
(*nat-graph-vertices G*) (*nat-graph-edges G*) (*nat-graph-weight G*) *a b*

definition *executable-label-of* :: *nat-dist* ⇒ *nat* ⇒ *real* **where**
executable-label-of ds v =
(*case bmssp-lookup-dist ds v of*
None ⇒ *bmssp-bound*
| *Some d* ⇒ *real d*)

lemma *bmssp-lookup-dist-mem-key*:
assumes *bmssp-lookup-dist ds v* = *Some d*
shows *v* ∈ *set* (*map fst ds*)
⟨*proof*⟩

lemma *bmssp-lookup-dist-Some-pair-mem*:
assumes *bmssp-lookup-dist ds v* = *Some d*
shows (*v*, *d*) ∈ *set ds*
⟨*proof*⟩

lemma *bmssp-lookup-dist-Some-if-distinct-mem*:
assumes *distinct*: *distinct* (*map fst ds*)
and *mem*: (*v*, *d*) ∈ *set ds*
shows *bmssp-lookup-dist ds v* = *Some d*
⟨*proof*⟩

lemma *bmssp-lookup-dist-None-if-distinct-not-mem*:
assumes *distinct*: *distinct* (*map fst ds*)
and *not-mem*: *v* ∉ *set* (*map fst ds*)
shows *bmssp-lookup-dist ds v* = *None*
⟨*proof*⟩

lemma *bmssp-set-dist-keys*:
set (*map fst* (*bmssp-set-dist v d ds*)) = *insert v* (*set* (*map fst ds*))
⟨*proof*⟩

lemma *bmssp-set-dist-fst-image* [*simp*]:
fst ‘ *set* (*bmssp-set-dist v d ds*) = *insert v* (*fst* ‘ *set ds*)
⟨*proof*⟩

lemma *bmssp-set-dist-preserves-distinct*:
assumes *distinct* (*map fst ds*)
shows *distinct* (*map fst* (*bmssp-set-dist v d ds*))
⟨*proof*⟩

lemma *bmssp-lookup-dist-set-dist-other*:
assumes *v* ≠ *x*
shows *bmssp-lookup-dist* (*bmssp-set-dist x dx ds*) *v* =

bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-lookup-dist-set-dist-same*:
bmssp-lookup-dist (*bmssp-set-dist x dx ds*) *x* = *Some dx*
⟨*proof*⟩

lemma *bmssp-normalize-dist-key-set* [*simp*]:
set (*map fst* (*bmssp-normalize-dist ds*)) = *set* (*map fst ds*)
⟨*proof*⟩

lemma *bmssp-normalize-dist-fst-image* [*simp*]:
fst ‘ *set* (*bmssp-normalize-dist ds*) = *fst* ‘ *set ds*
⟨*proof*⟩

lemma *bmssp-lookup-dist-None-iff-not-key*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *bmssp-lookup-dist ds v* = *None* ⟷ *v* ∉ *set* (*map fst ds*)
⟨*proof*⟩

lemma *bmssp-lookup-dist-normalize-dist*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *bmssp-lookup-dist* (*bmssp-normalize-dist ds*) *v* =
bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-relax-edges-preserves-distinct-dist*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *distinct* (*map fst* (*snd* (*bmssp-relax-edges G settled u du ds*)))
⟨*proof*⟩

lemma *bmssp-relax-vertices-preserves-distinct-dist*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *distinct* (*map fst* (*snd* (*bmssp-relax-vertices G settled xs ds*)))
⟨*proof*⟩

lemma *bmssp-relax-edges-dist-keys-subset*:
assumes *keys*: *set* (*map fst ds*) ⊆ *set vertices*
and *uV*: *u* ∈ *set vertices*
and *edge-targets*:
⋀*a b w*. (*a*, *b*, *w*) ∈ *set G* ⟹ *b* ∈ *set vertices*
shows *set* (*map fst* (*snd* (*bmssp-relax-edges G settled u du ds*)))
⊆ *set vertices*
⟨*proof*⟩

lemma *bmssp-relax-vertices-dist-keys-subset*:
assumes *keys*: *set* (*map fst ds*) ⊆ *set vertices*
and *xs-subset*: *set xs* ⊆ *set vertices*
and *edge-targets*:

$\bigwedge a\ b\ w.\ (a,\ b,\ w) \in set\ G \Longrightarrow b \in set\ vertices$
shows *set* (*map fst* (*snd* (*bmssp-relax-vertices G settled xs ds*)))
$\subseteq$ *set vertices*
⟨*proof*⟩

definition *real-label-integral-on* :: *nat set* ⇒ (*nat* ⇒ *real*) ⇒ *bool* **where**
real-label-integral-on U d ⟷
$(\forall\, v{\in}U.\ real\ (nat\ (floor\ (d\ v))) = d\ v)$

definition *encode-dist-assoc-list* :: *nat list* ⇒ (*nat* ⇒ *real*) ⇒ *nat-dist* **where**
encode-dist-assoc-list xs d =
map (λ*v*. (*v*, *nat* (*floor* (*d v*)))) (*sort* (*remdups xs*))

lemma *encode-dist-assoc-list-memI*:
assumes *v* ∈ *set xs*
shows ∃ *d*. (*v*, *d*) ∈ *set* (*encode-dist-assoc-list xs f*)
⟨*proof*⟩

lemma *encode-dist-assoc-list-memD*:
assumes (*v*, *d*) ∈ *set* (*encode-dist-assoc-list xs f*)
shows *v* ∈ *set xs*
⟨*proof*⟩

lemma *sort-remdups-eq-if-set-eq*:
assumes *set xs* = *set ys*
shows *sort* (*remdups xs*) = *sort* (*remdups ys*)
⟨*proof*⟩

lemma *encode-dist-assoc-list-cong-set*:
assumes *set xs* = *set ys*
shows *encode-dist-assoc-list xs f* = *encode-dist-assoc-list ys f*
⟨*proof*⟩

lemma *encode-dist-assoc-list-cong-set-floor*:
assumes *set-eq*: *set xs* = *set ys*
and *floors*:
$\bigwedge v.\ v \in set\ xs \Longrightarrow$
nat (*floor* (*f v*)) = *nat* (*floor* (*g v*))
shows *encode-dist-assoc-list xs f* = *encode-dist-assoc-list ys g*
⟨*proof*⟩

lemma *encode-dist-assoc-list-partition-key*:
encode-dist-assoc-list xs (λ*v*. *bmssp-partition-key v* (*f v*)) =
map (λ*v*. (*v*, *f v*)) (*sort* (*remdups xs*))
⟨*proof*⟩

lemma *map-fst-pair-map* [*simp*]:
map fst (*map* (λ*x*. (*x*, *f x*)) *xs*) = *xs*
⟨*proof*⟩

lemma *bmssp-normalize-dist-encode*:
  assumes *distinct*: *distinct* (*map fst ds*)
  shows *bmssp-normalize-dist ds* =
    encode-dist-assoc-list (*map fst ds*) (*executable-label-of ds*)
⟨*proof*⟩

lemma *bmssp-loop-zero-encode*:
  assumes *distinct* (*map fst ds*)
  shows *bmssp-loop 0 G vertices settled ds P* =
    encode-dist-assoc-list (*map fst ds*) (*executable-label-of ds*)
  ⟨*proof*⟩

lemma *bmssp-loop-empty-pull-encode*:
  assumes *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
    and *empty*: *filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices* = []
    and *distinct*: *distinct* (*map fst ds*)
  shows *bmssp-loop* (*Suc fuel*) *G vertices settled ds P* =
    encode-dist-assoc-list (*map fst ds*) (*executable-label-of ds*)
  ⟨*proof*⟩

lemma *bmssp-loop-preserves-distinct-output*:
  assumes *distinct*: *distinct* (*map fst ds*)
  shows *distinct* (*map fst* (*bmssp-loop fuel G vertices settled ds P*))
  ⟨*proof*⟩

lemma *bmssp-loop-output-sorted-keys*:
  sorted (*map fst* (*bmssp-loop fuel G vertices settled ds P*))
  ⟨*proof*⟩

lemma *distinct-sorted-dist-encode*:
  assumes *distinct*: *distinct* (*map fst ds*)
    and *sorted*: *sorted* (*map fst ds*)
  shows *ds* = *encode-dist-assoc-list* (*map fst ds*) (*executable-label-of ds*)
⟨*proof*⟩

lemma *bmssp-loop-output-encode*:
  assumes *distinct*: *distinct* (*map fst ds*)
  shows *bmssp-loop fuel G vertices settled ds P* =
    encode-dist-assoc-list
      (*map fst* (*bmssp-loop fuel G vertices settled ds P*))
      (*executable-label-of* (*bmssp-loop fuel G vertices settled ds P*))
⟨*proof*⟩

lemma *bmssp-loop-output-keys-subset-vertices*:
  assumes *keys*: *set* (*map fst ds*) ⊆ *set vertices*
    and *edge-targets*:
      ⋀*a b w*. (*a*, *b*, *w*) ∈ *set G* ⟹ *b* ∈ *set vertices*
  shows *set* (*map fst* (*bmssp-loop fuel G vertices settled ds P*))

$\subseteq$ *set vertices*
⟨*proof*⟩

lemma *executable-label-integral-on-keys*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *real-label-integral-on* (*set* (*map fst ds*)) (*executable-label-of ds*)
⟨*proof*⟩

lemma *nat-graph-vertex-list-set* [*simp*]:
set (*nat-graph-vertex-list G*) = *nat-graph-vertices G*
⟨*proof*⟩

lemma *nat-graph-edge-list-set* [*simp*]:
set (*nat-graph-edge-list G*) = *nat-graph-edges G*
⟨*proof*⟩

lemma *bmssp-relax-edges-update-edge*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
shows (*u*, *v*) ∈ *set* (*nat-graph-edge-list G*)
⟨*proof*⟩

lemma *bmssp-relax-edges-update-partition-key*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
shows ∃ *d*. *b* = *bmssp-partition-key v d*
⟨*proof*⟩

lemma *bmssp-relax-vertices-update-partition-key*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
shows ∃ *d*. *b* = *bmssp-partition-key v d*
⟨*proof*⟩

lemma *bmssp-relax-vertices-update-floor*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
shows ∃ *d*. *b* = *bmssp-partition-key v d* ∧ *nat* (*floor b*) = *d*
⟨*proof*⟩

lemma *bmssp-relax-edges-update-floor*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
shows ∃ *d*. *b* = *bmssp-partition-key v d* ∧ *nat* (*floor b*) = *d*
⟨*proof*⟩

lemma *bmssp-relax-edges-update-not-settled*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
shows *v* ∉ *set settled*
⟨*proof*⟩

lemma *bmssp-relax-vertices-update-not-settled*:
assumes (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
shows *v* ∉ *set settled*

⟨*proof*⟩

lemma *bmssp-relax-edges-dist-keys*:
set (*map fst* (*snd* (*bmssp-relax-edges G settled u du ds*))) =
set (*map fst ds*) ∪ *fst* ‘ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
⟨*proof*⟩

lemma *bmssp-relax-vertices-dist-keys*:
set (*map fst* (*snd* (*bmssp-relax-vertices G settled xs ds*))) =
set (*map fst ds*) ∪ *fst* ‘ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
⟨*proof*⟩

lemma *bmssp-relax-edges-lookup-not-updated*:
assumes *v* ∉ *fst* ‘ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-edges G settled u du ds*)) *v* =
bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-relax-vertices-lookup-not-updated*:
assumes *v* ∉ *fst* ‘ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-vertices G settled xs ds*)) *v* =
bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-relax-edges-update-lookup-floor*:
assumes *distinct-updates*:
distinct (*map fst* (*fst* (*bmssp-relax-edges G settled u du ds*)))
and *update*: (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-edges G settled u du ds*))
shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-edges G settled u du ds*)) *v* =
Some (*nat* (*floor b*))
⟨*proof*⟩

lemma *bmssp-relax-vertices-update-lookup-floor*:
assumes *distinct-updates*:
distinct (*map fst* (*fst* (*bmssp-relax-vertices G settled xs ds*)))
and *update*: (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-vertices G settled xs ds*)) *v* =
Some (*nat* (*floor b*))
⟨*proof*⟩

lemma *bmssp-relax-edges-lookup-Some-preserved*:
assumes *distinct*: *distinct* (*map fst ds*)
and *lookup*: *bmssp-lookup-dist ds v* = *Some d*
shows ∃ *d′*. *bmssp-lookup-dist*
(*snd* (*bmssp-relax-edges G settled u du ds*)) *v* = *Some d′*
⟨*proof*⟩

lemma *bmssp-relax-vertices-lookup-Some-preserved*:
assumes *distinct*: *distinct* (*map fst ds*)

and *lookup*: *bmssp-lookup-dist ds v = Some d*
shows $\exists\, d'.$ *bmssp-lookup-dist*
(snd (bmssp-relax-vertices G settled xs ds)) v = Some d'
⟨*proof*⟩

lemma *bmssp-relax-edges-lookup-le*:
assumes *distinct*: *distinct (map fst ds)*
and *old*: *bmssp-lookup-dist ds v = Some old*
and *new*:
bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
Some new
shows *new ≤ old*
⟨*proof*⟩

lemma *bmssp-relax-vertices-lookup-le*:
assumes *distinct*: *distinct (map fst ds)*
and *old*: *bmssp-lookup-dist ds v = Some old*
and *new*:
bmssp-lookup-dist (snd (bmssp-relax-vertices G settled xs ds)) v =
Some new
shows *new ≤ old*
⟨*proof*⟩

lemma *bmssp-relax-edges-edge-lookup-le-candidate*:
assumes *edge*: *(u, v, w) ∈ set G*
and *v-unsettled*: *v ∉ set settled*
shows $\exists\, dv.$
bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
Some dv ∧
dv ≤ du + w
⟨*proof*⟩

lemma *bmssp-relax-edges-update-improves-lookup*:
assumes *distinct*: *distinct (map fst ds)*
and *update*: *(v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))*
and *old*: *bmssp-lookup-dist ds v = Some old*
shows *nat (floor b) < old*
⟨*proof*⟩

lemma *bmssp-relax-vertices-update-improves-lookup*:
assumes *distinct*: *distinct (map fst ds)*
and *update*: *(v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))*
and *old*: *bmssp-lookup-dist ds v = Some old*
shows *nat (floor b) < old*
⟨*proof*⟩

lemma *bmssp-relax-edges-updates-distinct*:
assumes *distinct-edges*: *distinct (nat-graph-edge-list G)*
shows *distinct (map fst (fst (bmssp-relax-edges G settled u du ds)))*

⟨*proof*⟩

lemma *bmssp-relax-vertices-singleton-updates-distinct*:
assumes *distinct-edges*: *distinct* (*nat-graph-edge-list G*)
shows *distinct*
(*map fst* (*fst* (*bmssp-relax-vertices G settled* [*u*] *ds*)))
⟨*proof*⟩

lemma *bmssp-relax-vertices-updates-distinct-if-length-le-one*:
assumes *distinct-edges*: *distinct* (*nat-graph-edge-list G*)
and *len*: *length xs* ≤ *1*
shows *distinct* (*map fst* (*fst* (*bmssp-relax-vertices G settled xs ds*)))
⟨*proof*⟩

lemma *bmssp-relax-vertices-pulled-updates-distinct*:
assumes *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D′*
shows *distinct*
(*map fst*
(*fst* (*bmssp-relax-vertices G settled*
(*filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*) *ds*)))
⟨*proof*⟩

lemma *bmssp-apply-updates-ordered-after-pull*:
fixes *old-settled settled* :: *nat list*
assumes *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bp-ordered-invariant*
(*bmssp-apply-updates*
(*fst* (*bmssp-relax-vertices G settled*
(*filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*) *ds*))
P1)
⟨*proof*⟩

lemma *bmssp-partition-key-zero-lt-bound* [*simp*]:
bmssp-partition-key s 0 < *bmssp-bound*
⟨*proof*⟩

lemma *bmssp-partition-key-lt-bound-if-distance-lt*:
assumes *d* < *bmssp-infinity*
shows *bmssp-partition-key v d* < *bmssp-bound*
⟨*proof*⟩

lemma *bmssp-initial-partition-bridge*:
shows *bp-ordered-invariant*

```
    (bp-result-of
      (c-bp-regularized-local-insert s (bmssp-partition-key s 0)
        (bp-empty bmssp-block-size bmssp-bound)))
  and partition-upper-bound
    (bp-view
      (bp-result-of
        (c-bp-regularized-local-insert s (bmssp-partition-key s 0)
          (bp-empty bmssp-block-size bmssp-bound))))
    bmssp-bound
  and bp-view
      (bp-result-of
        (c-bp-regularized-local-insert s (bmssp-partition-key s 0)
          (bp-empty bmssp-block-size bmssp-bound))) =
    min-update (bp-view (bp-empty bmssp-block-size bmssp-bound))
      s (bmssp-partition-key s 0)
⟨proof⟩

lemma bmssp-loop-partition-step-bridge:
  fixes old-settled settled vertices :: nat list
    and pulled :: nat list
    and updates :: (nat × real) list
  assumes pulled-def:
      pulled =
        filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
    and updates-def: updates = fst (bmssp-relax-vertices G settled pulled ds)
    and P2-def: P2 = bmssp-apply-updates updates P1
    and wf: nat-graph-well-formed G
    and distinct-vertices: distinct vertices
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
  shows pull-separates (bp-view P) bmssp-block-size B S beta (bp-view P1)
    and bp-ordered-invariant P1
    and partition-upper-bound (bp-view P1) B
    and length pulled ≤ 1
    and distinct (map fst updates)
    and (⋀x b. (x, b) ∈ set updates ⟹
      ∃ d. b = bmssp-partition-key x d ∧ nat (floor b) = d)
    and bp-view P2 = batch-min-update (bp-view P1) updates
    and bp-ordered-invariant P2
    and (⋀x b. (x, b) ∈ set updates ⟹ b < B) ⟹
      partition-upper-bound (bp-view P2) B
⟨proof⟩

definition bmssp-partition-keys-match ::
  nat list ⇒ nat-dist ⇒ nat bucketed-partition ⇒ bool where
  bmssp-partition-keys-match settled ds P ⟷
    keys-of (bp-view P) = set (map fst ds) − set settled
```

lemma *bmssp-partition-keys-match-initial*:
bmssp-partition-keys-match [] [(*src*, *0*)]
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*)))
⟨*proof*⟩

lemma *bmssp-filter-pulled-set-eq*:
assumes *S-vertices*: $S \subseteq$ *set vertices*
and *S-unsettled*: $S \cap$ *set settled* $= \{\}$
shows *set* (*filter* ($\lambda x.\ x \in S \wedge x \notin$ *set settled*) *vertices*) $= S$
⟨*proof*⟩

lemma *bmssp-partition-keys-match-step*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *match*: *bmssp-partition-keys-match old-settled ds P*
and *keys-vertices*: *keys-of* (*bp-view P*) $\subseteq$ *set vertices*
and *pulled-def*:
pulled =
filter ($\lambda x.\ x \in S \wedge x \notin$ *set old-settled*) *vertices*
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds′*)
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta*
(*bp-view P1*)
and *view*: *bp-view P2* = *batch-min-update* (*bp-view P1*) *updates*
shows *bmssp-partition-keys-match settled ds′ P2*
⟨*proof*⟩

lemma *bmssp-partition-keys-match-loop-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *match*: *bmssp-partition-keys-match old-settled ds P*
and *keys-vertices*: *keys-of* (*bp-view P*) $\subseteq$ *set vertices*
and *pulled-def*:
pulled =
filter ($\lambda x.\ x \in S \wedge x \notin$ *set old-settled*) *vertices*
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds′*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bmssp-partition-keys-match settled ds′ P2*
⟨*proof*⟩

lemma *batch-min-update-value-not-updated*:
assumes $x \notin fst$ ‘ $set\ xs$
shows $value\text{-}of\ (batch\text{-}min\text{-}update\ D\ xs)\ x = value\text{-}of\ D\ x$
⟨*proof*⟩

lemma *batch-min-update-value-pair-less-old*:
assumes *distinct*: $distinct\ (map\ fst\ xs)$
and *pair*: $(x, b) \in set\ xs$
and *better*: $x \in keys\text{-}of\ D \Longrightarrow b < value\text{-}of\ D\ x$
shows $value\text{-}of\ (batch\text{-}min\text{-}update\ D\ xs)\ x = b$
⟨*proof*⟩

definition *bmssp-partition-values-match* ::
$nat\ list \Rightarrow nat\text{-}dist \Rightarrow nat\ partition\text{-}view \Rightarrow bool$ **where**
bmssp-partition-values-match settled ds D ⟷
$(\forall\, v\ d.\ bmssp\text{-}lookup\text{-}dist\ ds\ v = Some\ d \longrightarrow$
$v \notin set\ settled \longrightarrow value\text{-}of\ D\ v = bmssp\text{-}partition\text{-}key\ v\ d)$

lemma *bmssp-partition-values-match-initial*:
bmssp-partition-values-match [] $[(src, 0)]$
(*bp-view*
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*))))
⟨*proof*⟩

lemma *bmssp-partition-values-match-pull*:
bmssp-partition-values-match settled ds D $\Longrightarrow$
$value\text{-}of\ Dp = value\text{-}of\ D \Longrightarrow$
bmssp-partition-values-match settled ds Dp
⟨*proof*⟩

lemma *bmssp-partition-values-match-step*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: $(nat \times real)\ list$
assumes *vals*: *bmssp-partition-values-match old-settled ds* (*bp-view P*)
and *keys*: *bmssp-partition-keys-match old-settled ds P*
and *distinct-ds*: $distinct\ (map\ fst\ ds)$
and *pulled-def*:
pulled =
filter $(\lambda x.\ x \in S \wedge x \notin set\ old\text{-}settled)\ vertices$
and *settled-def*: $settled = pulled\ @\ old\text{-}settled$
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* $= (updates, ds')$
and *distinct-updates*: $distinct\ (map\ fst\ updates)$
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta*
(*bp-view P1*)
and *view*: *bp-view P2* = *batch-min-update* (*bp-view P1*) *updates*

shows *bmssp-partition-values-match settled ds′ (bp-view P2)*
⟨*proof*⟩

lemma *bmssp-partition-values-match-loop-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *vals*: *bmssp-partition-values-match old-settled ds (bp-view P)*
and *keys*: *bmssp-partition-keys-match old-settled ds P*
and *distinct-ds*: *distinct (map fst ds)*
and *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds′*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound (bp-view P) B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bmssp-partition-values-match settled ds′ (bp-view P2)*
⟨*proof*⟩

lemma *bmssp-partition-key-le-imp-distance-le*:
assumes *bmssp-partition-key u du* ≤ *bmssp-partition-key v dv*
shows *du* ≤ *dv*
⟨*proof*⟩

lemma *bmssp-pull-residual-label-le*:
assumes *vals*: *bmssp-partition-values-match old-settled ds (bp-view P)*
and *keys*: *bmssp-partition-keys-match old-settled ds P*
and *pull*: *pull-separates (bp-view P) bmssp-block-size B S beta*
(bp-view P1)
and *uS*: *u* ∈ *S*
and *u-lookup*: *bmssp-lookup-dist ds u* = *Some du*
and *v-lookup*: *bmssp-lookup-dist ds v* = *Some dv*
and *v-unsettled*: *v* ∉ *set old-settled*
and *v-not-S*: *v* ∉ *S*
shows *du* ≤ *dv*
⟨*proof*⟩

definition *bmssp-partition-state-match* ::
nat list ⇒ *nat list* ⇒ *nat-dist* ⇒ *nat bucketed-partition* ⇒ *bool* **where**
bmssp-partition-state-match vertices settled ds P ⟷
distinct (map fst ds) ∧
set (map fst ds) ⊆ *set vertices* ∧
bmssp-partition-keys-match settled ds P ∧
bmssp-partition-values-match settled ds (bp-view P)

lemma *bmssp-partition-state-match-initial*:
assumes *src-vertices*: *src* ∈ *set vertices*
shows *bmssp-partition-state-match vertices* [] [(*src*, *0*)]
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*)))
⟨*proof*⟩

lemma *bmssp-partition-state-match-keys-subset*:
assumes *bmssp-partition-state-match vertices settled ds P*
shows *keys-of* (*bp-view P*) ⊆ *set vertices*
⟨*proof*⟩

lemma *bmssp-partition-state-match-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *match*: *bmssp-partition-state-match vertices old-settled ds P*
and *edge-targets*:
⋀*a b w*. (*a*, *b*, *w*) ∈ *set G* ⟹ *b* ∈ *set vertices*
and *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bmssp-partition-state-match vertices settled ds' P2*
⟨*proof*⟩

lemma *bmssp-partition-state-pull-residual-label-le*:
assumes *match*: *bmssp-partition-state-match vertices old-settled ds P*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta*
(*bp-view P1*)
and *uS*: *u* ∈ *S*
and *u-lookup*: *bmssp-lookup-dist ds u* = *Some du*
and *v-lookup*: *bmssp-lookup-dist ds v* = *Some dv*
and *v-unsettled*: *v* ∉ *set old-settled*
and *v-not-S*: *v* ∉ *S*
shows *du* ≤ *dv*
⟨*proof*⟩

lemma *bmssp-partition-state-pulled-not-settled*:
assumes *match*: *bmssp-partition-state-match vertices settled ds P*

and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D'*
and *uS*: $u \in S$
shows $u \notin$ *set settled*
⟨*proof*⟩

lemma *bmssp-partition-state-pulled-lookup*:
assumes *match*: *bmssp-partition-state-match vertices settled ds P*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D'*
and *uS*: $u \in S$
obtains *d* **where** *bmssp-lookup-dist ds u* = *Some d*
⟨*proof*⟩

lemma *bmssp-partition-state-pulled-list-lookup*:
assumes *match*: *bmssp-partition-state-match vertices old-settled ds P*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D'*
and *pulled-def*:
pulled =
filter ($\lambda x.\ x \in S \wedge x \notin$ *set old-settled*) *vertices*
and *u-pulled*: $u \in$ *set pulled*
obtains *d* **where** *bmssp-lookup-dist ds u* = *Some d*
⟨*proof*⟩

lemma *nat-graph-edge-in-vertices*:
assumes (*u*, *v*) ∈ *nat-graph-edges G*
shows *u* ∈ *nat-graph-vertices G v* ∈ *nat-graph-vertices G*
⟨*proof*⟩

lemma *nat-graph-weight-nonneg* [*simp*]:
0 ≤ *nat-graph-weight G u v*
⟨*proof*⟩

lemma *bmssp-edge-vertices-set*:
set (*concat* (*map bmssp-edge-vertices G*)) = *nat-graph-vertices G*
⟨*proof*⟩

lemma *bmssp-vertices-set*:
set (*bmssp-vertices G src*) = *insert src* (*nat-graph-vertices G*)
⟨*proof*⟩

lemma *bmssp-vertices-set-if-source*:
assumes *src* ∈ *nat-graph-vertices G*
shows *set* (*bmssp-vertices G src*) = *nat-graph-vertices G*
⟨*proof*⟩

lemma *bmssp-edge-target-in-vertices*:
assumes (*a*, *b*, *w*) ∈ *set G*
shows *b* ∈ *set* (*bmssp-vertices G src*)
⟨*proof*⟩

lemma *bmssp-lookup-dist-Some-mem*:
assumes *bmssp-lookup-dist ds v* = *Some d*
shows (*v*, *d*) ∈ *set ds*
⟨*proof*⟩

lemma *bmssp-set-dist-mem-cases*:
assumes (*v*, *d*) ∈ *set* (*bmssp-set-dist x dx ds*)
shows (*v* = *x* ∧ *d* = *dx*) ∨ (*v*, *d*) ∈ *set ds*
⟨*proof*⟩

lemma *nat-graph-weight-of-edge*:
assumes *wf*: *nat-graph-well-formed G*
and *edge*: (*u*, *v*, *w*) ∈ *set G*
shows *nat-graph-weight G u v* = *real w*
⟨*proof*⟩

lemma *exec-walk-append-edge*:
assumes *walk*: *exec-walk vs es p*
and *p-ne*: *p* ≠ []
and *last-p*: *last p* = *u*
and *vV*: *v* ∈ *set vs*
and *edge*: (*u*, *v*) ∈ *set es*
shows *exec-walk vs es* (*p* @ [*v*])
⟨*proof*⟩

lemma *exec-walk-weight-append-edge*:
assumes *p* ≠ []
shows *exec-walk-weight W* (*p* @ [*v*]) =
exec-walk-weight W p + *W* (*last p*) *v*
⟨*proof*⟩

fun *exec-path-edges* **where**
exec-path-edges [] = []
| *exec-path-edges* [*x*] = []
| *exec-path-edges* (*x* # *y* # *xs*) = (*x*, *y*) # *exec-path-edges* (*y* # *xs*)

lemma *exec-walk-path-edges-subset*:
assumes *exec-walk vs es p*
shows *set* (*exec-path-edges p*) ⊆ *set es*
⟨*proof*⟩

lemma *exec-path-edges-vertices*:
assumes *e* ∈ *set* (*exec-path-edges p*)
shows *fst e* ∈ *set p snd e* ∈ *set p*
⟨*proof*⟩

lemma *exec-path-edges-distinct-if-distinct*:
assumes *distinct p*
shows *distinct* (*exec-path-edges p*)

⟨*proof*⟩

lemma *exec-walk-weight-sum-path-edges*:
exec-walk-weight W p =
sum-list (*map* (λ*e*. *W* (*fst e*) (*snd e*)) (*exec-path-edges p*))
⟨*proof*⟩

lemma *nat-graph-weight-edge-list-sum*:
assumes *wf*: *nat-graph-well-formed G*
shows *sum-list*
(*map* (λ*e*. *nat-graph-weight G* (*fst e*) (*snd e*))
(*nat-graph-edge-list G*)) =
real (*nat-graph-total-weight G*)
⟨*proof*⟩

lemma *nat-graph-weight-nonnegative*:
0 ≤ *nat-graph-weight G u v*
⟨*proof*⟩

lemma *exec-walk-weight-nat-graph-le-total*:
assumes *wf*: *nat-graph-well-formed G*
and *walk*: *exec-walk vs* (*nat-graph-edge-list G*) *p*
and *distinct*: *distinct p*
shows *exec-walk-weight* (*nat-graph-weight G*) *p* ≤
real (*nat-graph-total-weight G*)
⟨*proof*⟩

definition *bmssp-label-witnesses* ::
nat-graph ⇒ *nat* ⇒ *nat list* ⇒ *nat-dist* ⇒ *bool* **where**
bmssp-label-witnesses G src settled ds ⟷
(∀ *v d*. (*v*, *d*) ∈ *set ds* ⟶
(∃ *p*. *p* ≠ [] ∧ *hd p* = *src* ∧ *last p* = *v* ∧ *distinct p* ∧
exec-walk (*bmssp-vertices G src*) (*nat-graph-edge-list G*) *p* ∧
exec-walk-weight (*nat-graph-weight G*) *p* = *real d* ∧
set (*butlast p*) ⊆ *set settled*))

lemma *bmssp-label-witnesses-mono-settled*:
assumes *witnesses*: *bmssp-label-witnesses G src settled ds*
and *subset*: *set settled* ⊆ *set settled′*
shows *bmssp-label-witnesses G src settled′ ds*
⟨*proof*⟩

lemma *bmssp-label-witnesses-initial*:
bmssp-label-witnesses G src [] [(*src*, 0)]
⟨*proof*⟩

lemma *bmssp-label-witnesses-set-dist*:
assumes *witnesses*: *bmssp-label-witnesses G src settled ds*
and *p*: *p* ≠ [] *hd p* = *src last p* = *x distinct p*

exec-walk (*bmssp-vertices G src*) (*nat-graph-edge-list G*) *p*
exec-walk-weight (*nat-graph-weight G*) *p* = *real dx*
set (*butlast p*) ⊆ *set settled*
shows *bmssp-label-witnesses G src settled* (*bmssp-set-dist x dx ds*)
⟨*proof*⟩

lemma *bmssp-relax-edges-lookup-settled*:
assumes *v* ∈ *set settled*
shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-edges G settled u du ds*)) *v* =
bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-relax-vertices-lookup-settled*:
assumes *v* ∈ *set settled*
shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-vertices G settled xs ds*)) *v* =
bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-relax-vertices-edge-lookup-le-candidate*:
assumes *distinct*: *distinct* (*map fst ds*)
and *xs-subset*: *set xs* ⊆ *set settled*
and *u-xs*: *u* ∈ *set xs*
and *lookup-u*: *bmssp-lookup-dist ds u* = *Some du*
and *edge*: (*u*, *v*, *w*) ∈ *set G*
and *v-unsettled*: *v* ∉ *set settled*
shows ∃ *dv*.
bmssp-lookup-dist (*snd* (*bmssp-relax-vertices G settled xs ds*)) *v* =
Some dv ∧
dv ≤ *du* + *w*
⟨*proof*⟩

definition *bmssp-settled-exact* ::
nat-graph ⇒ *nat* ⇒ *nat list* ⇒ *nat-dist* ⇒ *bool* **where**
bmssp-settled-exact G src settled ds ⟷
(∀ *v d*. *v* ∈ *set settled* ⟶
bmssp-lookup-dist ds v = *Some d* ⟶
real d = *nat-graph-dist G src v*)

lemma *bmssp-settled-exact-initial*:
bmssp-settled-exact G src [] *ds*
⟨*proof*⟩

lemma *bmssp-settled-exact-step*:
assumes *exact-old*: *bmssp-settled-exact G src old-settled ds*
and *pulled-exact*:
⋀*v d*. ⟦*v* ∈ *set pulled*; *bmssp-lookup-dist ds v* = *Some d*⟧
⟹ *real d* = *nat-graph-dist G src v*
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)

shows *bmssp-settled-exact G src settled ds'*
⟨*proof*⟩

definition *bmssp-frontier-relaxed* ::
nat-graph ⇒ *nat* ⇒ *nat list* ⇒ *nat-dist* ⇒ *bool* **where**
bmssp-frontier-relaxed G src settled ds ⟷
(∀ *u v w du*. (*u*, *v*, *w*) ∈ *set G* ⟶
u ∈ *set settled* ⟶
v ∉ *set settled* ⟶
bmssp-lookup-dist ds u = *Some du* ⟶
real du = *nat-graph-dist G src u* ⟶
(∃ *dv*. *bmssp-lookup-dist ds v* = *Some dv* ∧
real dv ≤ *real du* + *real w*))

lemma *bmssp-frontier-relaxed-initial*:
bmssp-frontier-relaxed G src [] *ds*
⟨*proof*⟩

lemma *bmssp-frontier-relaxed-step*:
assumes *frontier-old*: *bmssp-frontier-relaxed G src old-settled ds*
and *distinct*: *distinct* (*map fst ds*)
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
shows *bmssp-frontier-relaxed G src settled ds'*
⟨*proof*⟩

definition *bmssp-source-zero* :: *nat* ⇒ *nat-dist* ⇒ *bool* **where**
bmssp-source-zero src ds ⟷ *bmssp-lookup-dist ds src* = *Some 0*

lemma *bmssp-source-zero-initial*:
bmssp-source-zero src [(*src*, *0*)]
⟨*proof*⟩

lemma *bmssp-source-zero-step*:
assumes *zero*: *bmssp-source-zero src ds*
and *distinct*: *distinct* (*map fst ds*)
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
shows *bmssp-source-zero src ds'*
⟨*proof*⟩

definition *bmssp-settled-have-lookups* ::
nat list ⇒ *nat-dist* ⇒ *bool* **where**
bmssp-settled-have-lookups settled ds ⟷
(∀ *v*∈*set settled*. ∃ *d*. *bmssp-lookup-dist ds v* = *Some d*)

lemma *bmssp-settled-have-lookups-initial*:
bmssp-settled-have-lookups [] *ds*
⟨*proof*⟩

lemma *bmssp-settled-have-lookups-step*:
  assumes *have-old*: *bmssp-settled-have-lookups old-settled ds*
    and *match*: *bmssp-partition-state-match vertices old-settled ds P*
    and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D′*
    and *pulled-def*:
      pulled = *filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*
    and *settled-def*: *settled* = *pulled* @ *old-settled*
    and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds′*)
  shows *bmssp-settled-have-lookups settled ds′*
  ⟨*proof*⟩

lemma *finite-card-le-one-member-eq*:
  assumes *finite*: *finite A*
    and *card*: *card A* ≤ *1*
    and *x*: *x* ∈ *A*
    and *y*: *y* ∈ *A*
  shows *x* = *y*
⟨*proof*⟩

definition *bmssp-dijkstra-state* ::
  nat-graph ⇒ *nat* ⇒ *nat list* ⇒ *nat list* ⇒
    nat-dist ⇒ *nat bucketed-partition* ⇒ *bool* **where**
  bmssp-dijkstra-state G src vertices settled ds P ⟷
      bmssp-partition-state-match vertices settled ds P ∧
      bmssp-source-zero src ds ∧
      bmssp-settled-have-lookups settled ds ∧
      bmssp-label-witnesses G src settled ds ∧
      bmssp-settled-exact G src settled ds ∧
      bmssp-frontier-relaxed G src settled ds

lemma *bmssp-dijkstra-state-initial*:
  assumes *src-vertices*: *src* ∈ *set vertices*
  shows *bmssp-dijkstra-state G src vertices* [] [(*src*, *0*)]
    (*bp-result-of*
      (*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
        (*bp-empty bmssp-block-size bmssp-bound*)))
⟨*proof*⟩

lemma *bmssp-relax-edges-preserves-label-witnesses-aux*:
  assumes *wf*: *nat-graph-well-formed Gfull*
    and *edge-subset*: *set Erel* ⊆ *set Gfull*
    and *witnesses*: *bmssp-label-witnesses Gfull src settled ds*
    and *u-settled*: *u* ∈ *set settled*
    and *lookup-u*: *bmssp-lookup-dist ds u* = *Some du*
  shows *bmssp-label-witnesses Gfull src settled*
    (*snd* (*bmssp-relax-edges Erel settled u du ds*))
  ⟨*proof*⟩

lemma *bmssp-relax-edges-preserves-label-witnesses*:

assumes *wf*: *nat-graph-well-formed G*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
and *u-settled*: *u* ∈ *set settled*
and *lookup-u*: *bmssp-lookup-dist ds u* = *Some du*
shows *bmssp-label-witnesses G src settled*
(*snd* (*bmssp-relax-edges G settled u du ds*))
⟨*proof*⟩

lemma *bmssp-relax-vertices-preserves-label-witnesses*:
assumes *wf*: *nat-graph-well-formed G*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
and *xs-subset*: *set xs* ⊆ *set settled*
shows *bmssp-label-witnesses G src settled*
(*snd* (*bmssp-relax-vertices G settled xs ds*))
⟨*proof*⟩

lemma *bmssp-label-witnesses-lookup-lt-infinity*:
assumes *wf*: *nat-graph-well-formed G*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
and *lookup*: *bmssp-lookup-dist ds v* = *Some d*
shows *d* < *bmssp-infinity*
⟨*proof*⟩

lemma *bmssp-relax-vertices-update-lt-bound*:
assumes *wf*: *nat-graph-well-formed G*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
and *xs-subset*: *set xs* ⊆ *set settled*
and *distinct-updates*:
distinct (*map fst* (*fst* (*bmssp-relax-vertices G settled xs ds*)))
and *update*: (*v*, *b*) ∈ *set* (*fst* (*bmssp-relax-vertices G settled xs ds*))
shows *b* < *bmssp-bound*
⟨*proof*⟩

lemma *bmssp-loop-lookup-settled*:
assumes *distinct*: *distinct* (*map fst ds*)
and *v-settled*: *v* ∈ *set settled*
shows *bmssp-lookup-dist* (*bmssp-loop fuel G vertices settled ds P*) *v* =
bmssp-lookup-dist ds v
⟨*proof*⟩

lemma *bmssp-loop-preserves-label-witnesses*:
assumes *wf*: *nat-graph-well-formed G*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
shows ∃ *settled′*. *set settled* ⊆ *set settled′* ∧
bmssp-label-witnesses G src settled′
(*bmssp-loop fuel G vertices settled ds P*)
⟨*proof*⟩

lemma *bmssp-distances-label-witnesses*:

assumes *wf*: *nat-graph-well-formed G*
shows ∃ *settled*. *bmssp-label-witnesses G src settled*
(*bmssp-distances G src*)
⟨*proof*⟩

lemma *bmssp-distances-output-simple-walk*:
assumes *wf*: *nat-graph-well-formed G*
and *mem*: (*v*, *d*) ∈ *set* (*bmssp-distances G src*)
obtains *p* **where**
p ≠ []
hd p = *src*
last p = *v*
distinct p
exec-walk (*bmssp-vertices G src*) (*nat-graph-edge-list G*) *p*
exec-walk-weight (*nat-graph-weight G*) *p* = *real d*
⟨*proof*⟩

lemma *bmssp-distances-distinct-keys*:
distinct (*map fst* (*bmssp-distances G src*))
⟨*proof*⟩

lemma *bmssp-distances-keys-subset-bmssp-vertices*:
set (*map fst* (*bmssp-distances G src*)) ⊆ *set* (*bmssp-vertices G src*)
⟨*proof*⟩

lemma *bmssp-distances-encode-own-keys*:
bmssp-distances G src =
encode-dist-assoc-list (*map fst* (*bmssp-distances G src*))
(*executable-label-of* (*bmssp-distances G src*))
⟨*proof*⟩

lemma *nat-graph-finite-weighted-digraph*:
assumes *wf*: *nat-graph-well-formed G*
and *src*: *src* ∈ *nat-graph-vertices G*
shows *finite-weighted-digraph*
(*nat-graph-vertices G*) (*nat-graph-edges G*) (*nat-graph-weight G*) *src*
⟨*proof*⟩

lemma *nat-graph-reachable-iff-exec-reachable*:
assumes *wf*: *nat-graph-well-formed G*
and *src*: *src* ∈ *nat-graph-vertices G*
shows *exec-reachable* (*nat-graph-vertex-list G*) (*nat-graph-edge-list G*) *a b*
⟷ *nat-graph-reachable G a b*
⟨*proof*⟩

lemma *nat-graph-exec-dist-eq-dist*:
assumes *wf*: *nat-graph-well-formed G*
and *src*: *src* ∈ *nat-graph-vertices G*
and *reach*: *nat-graph-reachable G a b*

shows *exec-dist* (*nat-graph-vertex-list G*) (*nat-graph-edge-list G*)
(*nat-graph-weight G*) *a b* =
nat-graph-dist G a b
⟨*proof*⟩

lemma *nat-graph-weight-integral*:
∃ *n*. *nat-graph-weight G u v* = *real n*
⟨*proof*⟩

lemma *exec-walk-weight-nat-graph-integral*:
∃ *n*. *exec-walk-weight* (*nat-graph-weight G*) *p* = *real n*
⟨*proof*⟩

lemma *min-list-integral-nonempty*:
assumes *xs* ≠ []
and ⋀*x*. *x* ∈ *set xs* ⟹ ∃ *n*. *x* = *real n*
shows ∃ *n*. *min-list xs* = *real n*
⟨*proof*⟩

lemma *exec-dist-nat-graph-integral*:
∃ *n*. *exec-dist vs es* (*nat-graph-weight G*) *a b* = *real n*
⟨*proof*⟩

lemma *nat-graph-dist-integral*:
assumes *wf*: *nat-graph-well-formed G*
and *src*: *src* ∈ *nat-graph-vertices G*
and *reach*: *nat-graph-reachable G a b*
shows ∃ *n*. *nat-graph-dist G a b* = *real n*
⟨*proof*⟩

lemma *real-nat-floor-integral*:
assumes ∃ *n*::*nat*. *x* = *real n*
shows *real* (*nat* (*floor x*)) = *x*
⟨*proof*⟩

locale *nat-graph-instance* =
fixes *G* :: *nat-graph*
and *src* :: *nat*
assumes *wf*: *nat-graph-well-formed G*
and *src-in*: *src* ∈ *nat-graph-vertices G*
begin

interpretation *concrete*: *finite-weighted-digraph*
nat-graph-vertices G nat-graph-edges G nat-graph-weight G src
⟨*proof*⟩

lemma *exec-reachable-iff-reachable*:
exec-reachable (*nat-graph-vertex-list G*) (*nat-graph-edge-list G*) *a b*
⟷ *concrete.reachable a b*

⟨*proof*⟩

lemma *exec-dist-eq-dist*:
assumes *concrete.reachable a b*
shows *exec-dist* (*nat-graph-vertex-list G*) (*nat-graph-edge-list G*)
(*nat-graph-weight G*) *a b* =
concrete.dist a b
⟨*proof*⟩

lemma *bmssp-vertices-carrier*:
set (*bmssp-vertices G src*) = *nat-graph-vertices G*
⟨*proof*⟩

lemma *bmssp-distances-keys-subset-carrier*:
set (*map fst* (*bmssp-distances G src*)) ⊆ *nat-graph-vertices G*
⟨*proof*⟩

lemma *bmssp-distances-encode-filtered-vertex-list*:
bmssp-distances G src =
encode-dist-assoc-list
(*filter* (λ*v*. *v* ∈ *set* (*map fst* (*bmssp-distances G src*)))
(*nat-graph-vertex-list G*))
(*executable-label-of* (*bmssp-distances G src*))
⟨*proof*⟩

lemma *bmssp-distances-output-abstract-simple-walk*:
assumes *mem*: (*v*, *d*) ∈ *set* (*bmssp-distances G src*)
obtains *p* **where**
concrete.simple-walk-betw src p v
concrete.walk-weight p = *real d*
⟨*proof*⟩

lemma *bmssp-distances-output-reachable*:
assumes (*v*, *d*) ∈ *set* (*bmssp-distances G src*)
shows *nat-graph-reachable G src v*
⟨*proof*⟩

lemma *bmssp-distances-output-dist-le*:
assumes *mem*: (*v*, *d*) ∈ *set* (*bmssp-distances G src*)
shows *nat-graph-dist G src v* ≤ *real d*
⟨*proof*⟩

lemma *bmssp-label-witnesses-lookup-abstract-simple-walk*:
assumes *witnesses*: *bmssp-label-witnesses G src settled ds*
and *lookup*: *bmssp-lookup-dist ds v* = *Some d*
obtains *p* **where**
concrete.simple-walk-betw src p v
concrete.walk-weight p = *real d*
⟨*proof*⟩

lemma *bmssp-label-witnesses-lookup-reachable*:
assumes *witnesses*: *bmssp-label-witnesses G src settled ds*
and *lookup*: *bmssp-lookup-dist ds v = Some d*
shows *nat-graph-reachable G src v*
⟨*proof*⟩

lemma *bmssp-label-witnesses-lookup-dist-le*:
assumes *witnesses*: *bmssp-label-witnesses G src settled ds*
and *lookup*: *bmssp-lookup-dist ds v = Some d*
shows *nat-graph-dist G src v* ≤ *real d*
⟨*proof*⟩

lemma *bmssp-shortest-walk-first-unsettled-lookup-le-dist*:
assumes *sp*: *concrete.shortest-walk src p u*
and *u-unsettled*: *u* ∉ *set settled*
and *source-zero*: *bmssp-source-zero src ds*
and *settled-lookup*: *bmssp-settled-have-lookups settled ds*
and *exact*: *bmssp-settled-exact G src settled ds*
and *frontier*: *bmssp-frontier-relaxed G src settled ds*
obtains *y dy* **where**
y ∈ *set p*
y ∉ *set settled*
bmssp-lookup-dist ds y = Some dy
real dy ≤ *nat-graph-dist G src y*
nat-graph-dist G src y ≤ *nat-graph-dist G src u*
⟨*proof*⟩

lemma *bmssp-partition-state-pulled-label-le-dist*:
assumes *match*: *bmssp-partition-state-match vertices settled ds P*
and *source-zero*: *bmssp-source-zero src ds*
and *settled-lookup*: *bmssp-settled-have-lookups settled ds*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
and *exact*: *bmssp-settled-exact G src settled ds*
and *frontier*: *bmssp-frontier-relaxed G src settled ds*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta*
(*bp-view P1*)
and *uS*: *u* ∈ *S*
and *lookup-u*: *bmssp-lookup-dist ds u = Some du*
shows *real du* ≤ *nat-graph-dist G src u*
⟨*proof*⟩

lemma *bmssp-partition-state-pulled-label-exact*:
assumes *match*: *bmssp-partition-state-match vertices settled ds P*
and *source-zero*: *bmssp-source-zero src ds*
and *settled-lookup*: *bmssp-settled-have-lookups settled ds*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
and *exact*: *bmssp-settled-exact G src settled ds*
and *frontier*: *bmssp-frontier-relaxed G src settled ds*

and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta*
(*bp-view P1*)
and *uS*: $u \in S$
and *lookup-u*: *bmssp-lookup-dist ds u* = *Some du*
shows *real du* = *nat-graph-dist G src u*
⟨*proof*⟩

lemma *bmssp-settled-exact-partition-step*:
assumes *match*: *bmssp-partition-state-match vertices old-settled ds P*
and *source-zero*: *bmssp-source-zero src ds*
and *settled-lookup*: *bmssp-settled-have-lookups old-settled ds*
and *witnesses*: *bmssp-label-witnesses G src old-settled ds*
and *exact*: *bmssp-settled-exact G src old-settled ds*
and *frontier*: *bmssp-frontier-relaxed G src old-settled ds*
and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta*
(*bp-view P1*)
and *pulled-def*:
pulled = *filter* (λ*x*. $x \in S \wedge x \notin$ *set old-settled*)
vertices
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds′*)
shows *bmssp-settled-exact G src settled ds′*
⟨*proof*⟩

lemma *bmssp-dijkstra-state-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *state*: *bmssp-dijkstra-state G src vertices old-settled ds P*
and *edge-targets*:
$\bigwedge a\ b\ w.\ (a, b, w) \in$ *set G* $\Longrightarrow b \in$ *set vertices*
and *pulled-def*:
pulled = *filter* (λ*x*. $x \in S \wedge x \notin$ *set old-settled*)
vertices
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds′*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bmssp-dijkstra-state G src vertices settled ds′ P2*
⟨*proof*⟩

lemma *bmssp-dijkstra-state-step-upper-bound*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *state*: *bmssp-dijkstra-state G src vertices old-settled ds P*

and *wf*: *nat-graph-well-formed G*
and *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*)
vertices
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *bmssp-bound*
and *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
shows *partition-upper-bound* (*bp-view P2*) *bmssp-bound*
⟨*proof*⟩

definition *bmssp-singleton-bucket-shape* ::
nat bucketed-partition ⇒ *bool* **where**
bmssp-singleton-bucket-shape P ⟷
bp-block-size P = *bmssp-block-size* ∧
(∀ *b*∈*set* (*bp-buckets P*).
∃ *p*. *bp-bucket-entries b* = [*p*] ∧ *bp-marker b* = *snd p*)

lemma *bmssp-bucketize-sorted-entries-aux-singleton-shape*:
∀ *b*∈*set* (*bp-bucketize-sorted-entries-aux fuel 1 xs*).
∃ *p*. *bp-bucket-entries b* = [*p*] ∧ *bp-marker b* = *snd p*
⟨*proof*⟩

lemma *bmssp-bucketize-entries-singleton-shape*:
∀ *b*∈*set* (*bp-bucketize-entries bmssp-block-size xs*).
∃ *p*. *bp-bucket-entries b* = [*p*] ∧ *bp-marker b* = *snd p*
⟨*proof*⟩

lemma *bmssp-rebucket-singleton-shape*:
assumes *block*: *bp-block-size P* = *bmssp-block-size*
shows *bmssp-singleton-bucket-shape* (*bp-rebucket P*)
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-drop-empty-prefix-id*:
assumes *shape*:
∀ *b*∈*set bs*. ∃ *p*. *bp-bucket-entries b* = [*p*] ∧ *bp-marker b* = *snd p*
shows *bp-drop-empty-prefix bs* = *bs*
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-flat-length*:
assumes *shape*:
∀ *b*∈*set bs*. ∃ *p*. *bp-bucket-entries b* = [*p*] ∧ *bp-marker b* = *snd p*
shows *length* (*bp-bucket-entries-flat bs*) = *length bs*
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-entries-empty-buckets*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
and *empty*: *bp-entries P* = []
shows *bp-buckets P* = []
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-trim*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
shows *bmssp-trim-empty-prefix P* = *P*
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-delete-first-bucket*:
assumes *buckets*: *bp-buckets P* = *b* # *bs*
and *shape*: *bmssp-singleton-bucket-shape P*
and *distinct*: *distinct* (*map fst* (*bp-entries P*))
shows *bmssp-singleton-bucket-shape*
(*bmssp-trim-empty-prefix* (*bp-delete-keys* (*bp-bucket-keys b*) *P*))
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-delete-all-small*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
and *small*: *length* (*bp-entries P*) ≤ *bmssp-block-size*
shows *bmssp-singleton-bucket-shape*
(*bmssp-trim-empty-prefix*
(*bp-delete-keys* (*bp-entry-keys* (*bp-entries P*)) *P*))
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-pull-trim*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
and *ord*: *bp-ordered-invariant P*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bmssp-singleton-bucket-shape* (*bmssp-trim-empty-prefix P1*)
⟨*proof*⟩

lemma *bmssp-singleton-bucket-shape-regularized-insert*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
shows *bmssp-singleton-bucket-shape*
(*bp-result-of* (*c-bp-regularized-local-insert x b P*))
⟨*proof*⟩

lemma *bmssp-insert-updates-singleton-shape*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
shows *bmssp-singleton-bucket-shape* (*bmssp-insert-updates xs P*)
⟨*proof*⟩

lemma *bmssp-bucketed-batch-prepend-empty-singleton-shape*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
and *empty*: *bp-entries P* = []
shows *bmssp-singleton-bucket-shape*

(*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
⟨*proof*⟩

lemma *bmssp-apply-updates-singleton-shape*:
assumes *shape*: *bmssp-singleton-bucket-shape* (*bmssp-trim-empty-prefix P*)
shows *bmssp-singleton-bucket-shape* (*bmssp-apply-updates xs P*)
⟨*proof*⟩

lemma *bmssp-partition-key-inject*:
assumes *eq*: *bmssp-partition-key v d* = *bmssp-partition-key w e*
shows *v* = *w* ∧ *d* = *e*
⟨*proof*⟩

lemma *distinct-map-fst-mem-eq*:
assumes *distinct*: *distinct* (*map fst xs*)
and *p*: *p* ∈ *set xs*
and *q*: *q* ∈ *set xs*
and *fst-eq*: *fst p* = *fst q*
shows *p* = *q*
⟨*proof*⟩

lemma *bmssp-partition-entries-value-inj*:
assumes *match*: *bmssp-partition-state-match vertices settled ds P*
and *ord*: *bp-ordered-invariant P*
shows *inj-on snd* (*set* (*bp-entries P*))
⟨*proof*⟩

lemma *bmssp-can-first-bucket-pull-if-singleton-shape*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
and *ord*: *bp-ordered-invariant P*
and *values-inj*: *inj-on snd* (*set* (*bp-entries P*))
and *many*: *length* (*bp-entries P*) > *bmssp-block-size*
shows *bp-can-first-bucket-pull bmssp-block-size P*
⟨*proof*⟩

lemma *bmssp-pull-nonempty-if-keys-nonempty*:
assumes *shape*: *bmssp-singleton-bucket-shape P*
and *ord*: *bp-ordered-invariant P*
and *values-inj*: *inj-on snd* (*set* (*bp-entries P*))
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
and *keys-nonempty*: *keys-of* (*bp-view P*) ≠ {}
shows *S* ≠ {}
⟨*proof*⟩

definition *bmssp-dijkstra-loop-state* ::
nat list ⇒ *nat list* ⇒
nat-dist ⇒ *nat bucketed-partition* ⇒ *bool* **where**
bmssp-dijkstra-loop-state vertices settled ds P ⟷
bmssp-dijkstra-state G src vertices settled ds P ∧

bp-ordered-invariant P ∧
partition-upper-bound (*bp-view P*) *bmssp-bound*

lemma *bmssp-dijkstra-loop-state-initial*:
assumes *src-vertices*: *src* ∈ *set vertices*
shows *bmssp-dijkstra-loop-state vertices* [] [(*src*, *0*)]
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*)))
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *loop-state*:
bmssp-dijkstra-loop-state vertices old-settled ds P
and *edge-targets*:
⋀*a b w*. (*a*, *b*, *w*) ∈ *set G* ⟹ *b* ∈ *set vertices*
and *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*)
vertices
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
shows *bmssp-dijkstra-loop-state vertices settled ds' P2*
⟨*proof*⟩

lemma *distinct-subset-length-le*:
assumes *xs*: *distinct xs*
and *ys*: *distinct ys*
and *subset*: *set xs* ⊆ *set ys*
shows *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-settled-subset-vertices*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
shows *set settled* ⊆ *set vertices*
⟨*proof*⟩

lemma *bmssp-loop-quiescent-state*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *shape*: *bmssp-singleton-bucket-shape P*
and *edge-targets*:

$\bigwedge a\ b\ w.\ (a,\ b,\ w) \in set\ G \Longrightarrow b \in set\ vertices$
and *distinct-vertices*: *distinct vertices*
and *distinct-settled*: *distinct settled*
and *fuel*: *fuel* > *length vertices* − *length settled*
shows $\exists$ *settled′ ds′ P′ S beta P1*.
bmssp-loop fuel G vertices settled ds P = *bmssp-normalize-dist ds′* ∧
bmssp-dijkstra-loop-state vertices settled′ ds′ P′ ∧
bmssp-singleton-bucket-shape P′ ∧
distinct settled′ ∧
set settled ⊆ *set settled′* ∧
bp-pull bmssp-block-size bmssp-bound P′ = (*S*, *beta*, *P1*) ∧
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled′*) *vertices* = []
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-lookup-exact-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *lookup*: *bmssp-lookup-dist ds v* = *Some d*
shows *real d* = *nat-graph-dist G src v*
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-normalize-exact-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *mem*: (*v*, *d*) ∈ *set* (*bmssp-normalize-dist ds*)
shows *real d* = *nat-graph-dist G src v*
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-normalize-reachable*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *mem*: (*v*, *d*) ∈ *set* (*bmssp-normalize-dist ds*)
shows *nat-graph-reachable G src v*
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-reachable-lookup-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *reach*: *nat-graph-reachable G src v*
shows $\exists$ *d*. *bmssp-lookup-dist ds v* = *Some d*
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-normalize-complete-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *reach*: *nat-graph-reachable G src v*
shows $\exists$ *d*. (*v*, *d*) ∈ *set* (*bmssp-normalize-dist ds*)
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-quiescent-pull-empty*:

assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
and *stopped*:
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices* = []
shows *S* = {}
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-keys-empty-if-pull-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *shape*: *bmssp-singleton-bucket-shape P*
and *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
and *S-empty*: *S* = {}
shows *keys-of* (*bp-view P*) = {}
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-quiescent-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *shape*: *bmssp-singleton-bucket-shape P*
and *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
and *stopped*:
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices* = []
shows *keys-of* (*bp-view P*) = {}
⟨*proof*⟩

definition *bmssp-executable-distances-fuel* :: *nat* **where**
bmssp-executable-distances-fuel =
Suc (*length* (*bmssp-vertices G src*) ∗ *Suc* (*length G*))

definition *bmssp-executable-initial-partition* :: *nat bucketed-partition* **where**
bmssp-executable-initial-partition =
bp-result-of
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*))

lemma *bmssp-distances-unfold-executable*:
bmssp-distances G src =
bmssp-loop bmssp-executable-distances-fuel G (*bmssp-vertices G src*) []
[(*src*, *0*)] *bmssp-executable-initial-partition*
⟨*proof*⟩

lemma *bmssp-executable-initial-partition-singleton-shape*:
bmssp-singleton-bucket-shape bmssp-executable-initial-partition
⟨*proof*⟩

lemma *bmssp-dijkstra-loop-state-initial-executable*:
bmssp-dijkstra-loop-state (*bmssp-vertices G src*) [] [(*src*, *0*)]
bmssp-executable-initial-partition
⟨*proof*⟩

lemma *bmssp-distances-quiescent-state-executable*:
obtains *settled ds P S beta P1* **where**
bmssp-distances G src = bmssp-normalize-dist ds
bmssp-dijkstra-loop-state (*bmssp-vertices G src*) *settled ds P*
bmssp-singleton-bucket-shape P
bp-pull bmssp-block-size bmssp-bound P = (*S*, *beta*, *P1*)
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*)
(*bmssp-vertices G src*) = []
⟨*proof*⟩

theorem *bmssp-correct-instance*:
shows
∀ (*v*, *d*)∈*set* (*bmssp-distances G src*).
real d = *nat-graph-dist G src v*
∀ *v*∈*nat-graph-vertices G*. *nat-graph-reachable G src v* ⟶
(∃ *d*. (*v*, *d*) ∈ *set* (*bmssp-distances G src*))
⟨*proof*⟩

lemma *bmssp-distances-executable-integral-on-keys*:
real-label-integral-on (*set* (*map fst* (*bmssp-distances G src*)))
(*executable-label-of* (*bmssp-distances G src*))
⟨*proof*⟩

end

theorem *bmssp-correct-executable*:
assumes *nat-graph-well-formed G*
and *src* ∈ *nat-graph-vertices G*
shows
∀ (*v*, *d*) ∈ *set* (*bmssp-distances G src*).
real d = *nat-graph-dist G src v*
∀ *v* ∈ *nat-graph-vertices G*. *nat-graph-reachable G src v* ⟶
(∃ *d*. (*v*, *d*) ∈ *set* (*bmssp-distances G src*))
⟨*proof*⟩

end
theory *BMSSP-Executable-Refinement*
imports *BMSSP-Executable-Refinement-Internal*
begin

39 Executable Refinement

This theory is the public import point for the executable refinement chain. The detailed refinement proof lives in *BMSSP-Executable-Refinement-Internal*. The aliases below keep the outward-facing facts under the public theory name.

lemmas *bmssp-distances-distinct-keys* =
BMSSP-Executable-Refinement-Internal.bmssp-distances-distinct-keys

lemmas *bmssp-correct-executable* =
BMSSP-Executable-Refinement-Internal.bmssp-correct-executable

end
theory *BMSSP-Executable-Headline*
imports *BMSSP-Executable-Refinement*
begin

40 Executable Headline

This theory states the headline correctness theorem of the entire development. It is the entry point for a reader who wants one statement that summarises what is proved, and it is the theorem that should be consulted before any other. The body of the proof is a short forward reference to the verified executable contract.

The headline applies to the executable entry point *bmssp-distances*, which takes a finite directed graph of natural vertices with natural-valued edge weights and a source vertex, and returns a finite association list mapping each reachable vertex to its shortest-path distance from the source. The graph representation *nat-graph*, the well-formedness predicate *nat-graph-well-formed*, the vertex set *nat-graph-vertices*, the reachability relation *nat-graph-reachable*, and the shortest distance function *nat-graph-dist* are all defined earlier in the session and used unchanged here.

The theorem below is intentionally a single complete statement. It exposes soundness, completeness, and uniqueness of output keys together, so readers do not have to reconstruct the exact map property from several smaller auxiliary facts.

theorem *bmssp-correct-strong*:
assumes *well-formed*: *nat-graph-well-formed G*
and *src-in*: *src* ∈ *nat-graph-vertices G*
shows
distinct (*map fst* (*bmssp-distances G src*)) ∧
set (*map fst* (*bmssp-distances G src*)) =
{*v* ∈ *nat-graph-vertices G*. *nat-graph-reachable G src v*} ∧
(∀ (*v*, *d*) ∈ *set* (*bmssp-distances G src*).
real d = *nat-graph-dist G src v*)
⟨*proof*⟩

The proof combines the underlying executable theorem ⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ (*v*, *d*)∈*set* (*bmssp-distances ?G ?src*). *real d* = *nat-graph-dist ?G ?src v*

⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ *v*∈*nat-graph-vertices ?G*. *nat-graph-reachable ?G ?src v* ⟶ (∃ *d*. (*v*, *d*) ∈ *set* (*bmssp-distances ?G ?src*)) with the distinct-key and output-reachability facts from the executable refinement layer. The two clauses of ⟦*nat-graph-well-*

formed ?G; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ (*v*, *d*)∈*set* (*bmssp-distances ?G ?src*). *real d* = *nat-graph-dist ?G ?src v*

⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ *v*∈*nat-graph-vertices ?G*. *nat-graph-reachable ?G ?src v* ⟶ (∃ *d*. (*v*, *d*) ∈ *set* (*bmssp-distances ?G ?src*)) give exact distances for every returned pair and completeness for every reachable vertex; the local key-set argument above adds the converse inclusion and the distinctness statement.

The proof is deliberately structured rather than written as a one-line automation script. The intention is to make the proof obligation visible and to expose how the headline is derived from the verified executable contract. The automated steps are confined to small set and tuple manipulations after the named refinement facts have been supplied. Tracing the proof reveals exactly which named theorem proves what.

In short, the entire development reduces to:

- *bmssp-distances* computes a finite list of pairs.
- For every well-formed graph and every source vertex of that graph, every returned pair gives the exact shortest-path distance from the source.
- For every well-formed graph and every source vertex of that graph, every reachable vertex appears in the output, and no unreachable vertex appears.

The remaining theories prove these facts. This file states them.

end
theory *BMSSP-Runtime-Instance*
  imports *BMSSP-Top-Level-Bounds BMSSP-Executable-Base-Case*
begin

41 A Concrete Non-Vacuous Instance of the Runtime Locale

The asymptotic running-time headline lives in the *bounded-reduced-positive-instance* locale, whose interpretation yields the closed theorem *bmssp-runtime-headline-instance.bmssp-runtime-bigo-target*. A locale theorem only carries content once the locale is shown to be inhabited. This theory exhibits a concrete, non-trivial graph—the unit-weight directed path *0* → *1* → *2* → *3*—and discharges every assumption of that locale, obtaining the *O*(*m* ∗ (*ln n*) *powr* (*2* / *3*)) bound for it as a closed, assumption-free statement. This certifies that the runtime headline is not vacuously true.

Uniqueness of shortest walks—the only non-routine locale assumption—is obtained from the verified simple-walk enumerator *exec-simple-walks-betw*:

on this graph the enumerator returns exactly one simple walk to each vertex, so any two shortest walks to a vertex coincide.

definition *path-vs* :: *nat list* **where**
path-vs = [*0*, *1*, *2*, *3*]

definition *path-es* :: (*nat* × *nat*) *list* **where**
path-es = [(*0*, *1*), (*1*, *2*), (*2*, *3*)]

definition *path-V* :: *nat set* **where**
path-V = *set path-vs*

definition *path-E* :: (*nat* × *nat*) *set* **where**
path-E = *set path-es*

definition *path-weight* :: *nat* ⇒ *nat* ⇒ *real* **where**
path-weight u v = (*if* (*u*, *v*) ∈ *path-E then 1 else 0*)

lemma *path-vs-V*: *set path-vs* = *path-V*
⟨*proof*⟩

lemma *path-es-E*: *set path-es* = *path-E*
⟨*proof*⟩

lemma *path-positive-weight*: (*u*, *v*) ∈ *path-E* ⟹ *0* < *path-weight u v*
⟨*proof*⟩

lemma *path-finite-weighted-digraph*:
finite-weighted-digraph path-V path-E path-weight 0
⟨*proof*⟩

interpretation *pfw*: *finite-weighted-digraph path-V path-E path-weight 0*
⟨*proof*⟩

The enumerator returns exactly one simple walk from *0* to each vertex, checked by evaluation of the generated code.

lemma *path-walk-lists*:
exec-simple-walks-betw path-vs path-es 0 0 = [[*0*]]
exec-simple-walks-betw path-vs path-es 0 (*Suc 0*) = [[*0*, *1*]]
exec-simple-walks-betw path-vs path-es 0 2 = [[*0*, *1*, *2*]]
exec-simple-walks-betw path-vs path-es 0 3 = [[*0*, *1*, *2*, *3*]]
⟨*proof*⟩

lemma *path-simple-walk-sets*:
{*p*. *pfw.simple-walk-betw 0 p 0*} = {[*0*]}
{*p*. *pfw.simple-walk-betw 0 p* (*Suc 0*)} = {[*0*, *1*]}
{*p*. *pfw.simple-walk-betw 0 p 2*} = {[*0*, *1*, *2*]}
{*p*. *pfw.simple-walk-betw 0 p 3*} = {[*0*, *1*, *2*, *3*]}
⟨*proof*⟩

lemma *path-walk-vertex*:
  assumes *pfw.simple-walk-betw 0 p v*
  shows *v* ∈ *path-V*
⟨*proof*⟩

lemma *path-unique-shortest-walk*:
  assumes *pfw.shortest-walk 0 p v* **and** *pfw.shortest-walk 0 q v*
  shows *p* = *q*
⟨*proof*⟩

lemma *path-all-reachable*:
  assumes *v* ∈ *path-V*
  shows *pfw.reachable 0 v*
⟨*proof*⟩

interpretation *pusd*: *unique-shortest-digraph path-V path-E path-weight 0*
  ⟨*proof*⟩

lemma *path-edge-outdegree*: *pusd.edge-outdegree-le 1*
  ⟨*proof*⟩

interpretation *pbr*: *bounded-reduced-positive-instance path-V path-E path-weight 0 1*
  ⟨*proof*⟩

The closed running-time bound for the concrete path instance. Because it is obtained by interpreting *bounded-reduced-positive-instance* on an explicit graph, it has no remaining locale hypotheses: it is an assumption-free witness that the BMSSP runtime headline is satisfiable by a genuine non-trivial digraph.

lemmas *path-runtime-bigo-target* =
  pbr.runtime-headline.bmssp-runtime-bigo-target

end
theory *BMSSP-Bucketed-Cost-Bridge*
  imports *BMSSP-Bucketed-Partition BMSSP-Top-Level-Bounds*
begin

42 Bridging the Bucketed Operation Costs to the Schedule Parameters

This theory closes the gap between the two cost layers of the development.

The costed BMSSP recurrence (the relations *direct-insert-costed-bmssp* and *exact-concrete-bmssp*) charges every direct insert at the abstract parameter t and every batch-prepended item at the abstract parameter h, through the predicates *partition-insert-cost-bound* and *partition-batch-cost-bound*. The top-level analysis then sets t to the two-thirds logarithmic

schedule parameter and h to the one-third parameter.

Independently, the bucketed partition theory proves that its concrete operations realise the abstract predicates with the paper-tight budgets *bp-insert-paper-budget* and *bp-batch-prepend-paper-budget*, each of the shape *const* + *floor-log* (*Suc* (*N div M*)).

What was missing is the arithmetic bridge: the realised bucketed budget *const* + *floor-log* (*Suc* (*N div M*)) is no larger than the schedule parameter whenever the ratio *N div M* is below the corresponding power of two. The bucket directory is searched in *floor-log* (*Suc* (*N div M*)) steps, so the only fact needed is that *floor-log* of a number strictly below *2* ^ *E* does not exceed *E*. This theory proves that arithmetic, packages it into the two interface predicates, and exposes the resulting connected statements: with the bucketed costs in force, the per-level ratio bound is exactly what makes the abstract Insert and BatchPrepend cost parameters realisable at the logarithmic schedule scales used by the headline recurrence.

42.1 Logarithmic Arithmetic

The single arithmetic fact underlying the whole bridge: if a positive number is strictly below *2* ^ *E*, its binary logarithm is at most *E*. Everything about the bucketed budgets reduces to this through monotonicity of *floor-log*.

lemma *floor-log-less-exp2-le*:
  assumes $n < 2 \mathbin{\hat{}} E$
  shows *floor-log* $n \leq E$
⟨*proof*⟩

lemma *floor-log-Suc-div-le*:
  assumes *Suc* (*N div M*) $\leq 2 \mathbin{\hat{}} E$
  shows *floor-log* (*Suc* (*N div M*)) $\leq E$
⟨*proof*⟩

The ratio condition we use throughout: at most *2* ^ *E* blocks worth of entries are live, i.e. *N div M* $< 2 \mathbin{\hat{}} E$. Equivalently *Suc* (*N div M*) $\leq 2 \mathbin{\hat{}} E$, the form consumed by *Suc* (*?N div ?M*) $\leq 2^{?E} \Longrightarrow$ *floor-log* (*Suc* (*?N div ?M*)) $\leq$ *?E*.

definition *ratio-below-pow2* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  ratio-below-pow2 N M E ⟷ *N div M* $< 2 \mathbin{\hat{}} E$

lemma *ratio-below-pow2-floor-log*:
  assumes *ratio-below-pow2 N M E*
  shows *floor-log* (*Suc* (*N div M*)) $\leq E$
⟨*proof*⟩

42.2 Bucketed Budgets at the Ratio Bound

Under the ratio bound the two paper budgets collapse to a constant plus the exponent E. These are the budgets that the realisation corollaries *bp-regular-invariant ?P* $\Longrightarrow \exists\, t.$ *partition-insert-cost-bound* (*bp-steps-of* (*c-bp-paper-insert ?x ?b ?P*)) $t \wedge t =$ *bp-insert-paper-budget* (*length* (*bp-entries ?P*)) (*bp-block-size ?P*) and $\exists\, t.$ *partition-batch-cost-bound* (*bp-steps-of* (*c-bp-paper-batch-prepend ?xs ?P*)) *t ?xs* $\wedge$ $t =$ *bp-batch-prepend-paper-budget* (*length ?xs*) (*bp-block-size ?P*) attach to the concrete bucketed operations.

lemma *bp-insert-paper-budget-le-const-plus-exp*:
  assumes *ratio-below-pow2 N M E*
  shows *bp-insert-paper-budget N M* $\leq$ *9 + E*
  ⟨*proof*⟩

lemma *bp-batch-prepend-paper-budget-le-const-plus-exp*:
  assumes *ratio-below-pow2 N M E*
  shows *bp-batch-prepend-paper-budget N M* $\leq$ *2 + E*
  ⟨*proof*⟩

42.3 Monotonicity of the Abstract Cost Predicates

The abstract cost predicates are $c \leq t$ and $c \leq t * \mathit{length}\ \mathit{xs}$, so they are monotone in the budget parameter. Weakening a realised budget to a larger one therefore preserves the predicate; this is what lets us replace the precise bucketed budget by the constant-plus-exponent bound and then by the schedule parameter.

lemma *partition-insert-cost-bound-mono*:
  fixes *c t t′* :: *nat*
  assumes *partition-insert-cost-bound c t*
    and $t \leq t'$
  shows *partition-insert-cost-bound c t′*
⟨*proof*⟩

lemma *partition-batch-cost-bound-mono*:
  fixes *c t t′* :: *nat*
  assumes *partition-batch-cost-bound c t xs*
    and $t \leq t'$
  shows *partition-batch-cost-bound c t′ xs*
⟨*proof*⟩

42.4 Bucketed Operations Realise the Schedule Cost Parameters

These are the central connecting facts. Each says: the *actual* step count of a concrete bucketed operation satisfies the abstract cost predicate with the budget set to *const* + E, provided the operation runs on a state whose live ratio is below *2* $\widehat{}$ E. No free abstract parameter remains: the budget

is the measured cost of the bucketed implementation, bounded through the logarithmic arithmetic above.

These bridge the realisation corollaries *bp-regular-invariant ?P* $\Longrightarrow \exists\, t.$ *partition-insert-cost-bound* (*bp-steps-of* (*c-bp-paper-insert ?x ?b ?P*)) $t \wedge t$ = *bp-insert-paper-budget* (*length* (*bp-entries ?P*)) (*bp-block-size ?P*) and $\exists\, t.$ *partition-batch-cost-bound* (*bp-steps-of* (*c-bp-paper-batch-prepend ?xs ?P*)) *t ?xs* $\wedge$ *t* = *bp-batch-prepend-paper-budget* (*length ?xs*) (*bp-block-size ?P*) (which exhibit the budget as *some* value equal to the paper budget) to a concrete, schedule-facing inequality.

theorem *c-bp-paper-insert-realises-const-plus-exp*:
  assumes *reg*: *bp-regular-invariant P*
    and *ratio*: *ratio-below-pow2* (*length* (*bp-entries P*)) (*bp-block-size P*) *E*
  shows *partition-insert-cost-bound*
    (*bp-steps-of* (*c-bp-paper-insert x b P*)) (*9* + *E*)
⟨*proof*⟩

theorem *c-bp-paper-batch-prepend-realises-const-plus-exp*:
  assumes *ratio*: *ratio-below-pow2* (*length xs*) (*bp-block-size P*) *E*
  shows *partition-batch-cost-bound*
    (*bp-steps-of* (*c-bp-paper-batch-prepend xs P*)) (*2* + *E*) *xs*
⟨*proof*⟩

The Pull cost needs no ratio bound: the bucketed Pull already realises the abstract pull predicate exactly (cost = *card S*), independent of the live ratio. We re-expose it here so the three operation costs sit together as the connected interface used by the recurrence.

theorem *c-bp-paper-pull-realises-partition-pull-cost-bound*:
  assumes *pull*: *bp-result-of* (*c-bp-paper-pull M B P*) = (*S*, *beta*, *P′*)
  shows *partition-pull-cost-bound* (*bp-steps-of* (*c-bp-paper-pull M B P*)) *S*
  ⟨*proof*⟩

42.5 The Bucketed Cost Parameters Sit at the Schedule Scales

The previous theorems show, per operation, that the measured bucketed step count satisfies the abstract cost predicate with budget *const* + *E*, where *E* is the live-ratio exponent. The headline recurrence is solved by ⟦*0 < ?Cn*; *0 < ?Ca*; *0 ≤ ?Cl*; *0 < ?Cr*; *0 < ?Ct*; *0 < ?Ch*; $\forall_F$ *n in sequentially. real n ≤ ?Cn * real* (*?m n*); $\forall_F$ *n in sequentially. real* (*?A n*) ≤ *?Ca * sssp-log-factor-one-third n*; $\forall_F$ *n in sequentially. real* (*?l n*) ≤ *?Cl * sssp-log-factor-one-third n*; $\forall_F$ *n in sequentially. real* (*?R n*) ≤ *?Cr * sssp-log-factor n*; $\forall_F$ *n in sequentially. real* (*?t n*) ≤ *?Ct * sssp-log-factor n*; $\forall_F$ *n in sequentially. real* (*?H n*) ≤ *?Ch * sssp-log-factor-one-third n*; $\forall_F$ *n in sequentially. ?T n ≤ bmssp-refined-graph-time-bound ?A ?R ?H ?l ?t ?m n*⟧ $\Longrightarrow$ ($\lambda n.$ *real* (*?T n*)) $\in O$(*sssp-time-target ?m*), which needs the Insert cost parameter *t* to grow like $log^{2/3}$ and the BatchPrepend cost parameter

H to grow like $log^{1/3}$.

We now discharge exactly those two envelope facts for the bucketed budgets. For Insert, the live ratio is allowed to reach the two-thirds schedule parameter, so the budget is *9 + sssp-log-two-thirds-param n*; a constant added to $log^{2/3}$ is still $O(log^{2/3})$. For BatchPrepend, the per-pull batch has length at most a constant multiple of the block size (bounded out-degree), so its ratio exponent is a constant c; the budget is then $2 + c$, a constant, which is $O(log^{1/3})$ because the one-third factor tends to infinity. These are the genuine reasons the bucketed costs fit the deterministic schedule.

lemma *sssp-log-factor-at-top*:
filterlim sssp-log-factor at-top at-top
⟨*proof*⟩

Insert envelope: a constant plus the two-thirds schedule parameter is eventually dominated by twice the two-thirds log factor. This is the *t-bound* hypothesis with the bucketed Insert budget in the parameter slot.

lemma *const-plus-two-thirds-param-le-two-thirds-factor*:
eventually
(λ*n*. *real* (*c* + *sssp-log-two-thirds-param n*) ≤
real (*c* + *2*) ∗ *sssp-log-factor n*) *at-top*
⟨*proof*⟩

BatchPrepend envelope: a constant budget is eventually dominated by any positive multiple of the one-third log factor, because that factor tends to infinity. This is the *H-bound* hypothesis with the bucketed BatchPrepend budget (a constant under bounded out-degree) in the parameter slot.

lemma *const-le-one-third-factor*:
eventually
(λ*n*. *real* (*c*::*nat*) ≤ *sssp-log-factor-one-third n*) *at-top*
⟨*proof*⟩

The two envelopes packaged as the schedule-facing component bounds. The first states that the bucketed Insert cost parameter λ*n*. *9 + sssp-log-two-thirds-param n* is $O(log^{2/3})$; the second that the bucketed BatchPrepend cost parameter λ*n*. *2 + c* is $O(log^{1/3})$. These are exactly the *t-bound* and *H-bound* premises consumed by the recurrence solver, now justified by the realised bucketed budgets instead of left as free parameters.

theorem *bucketed-insert-param-two-thirds-envelope*:
eventually
(λ*n*. *real* (*9 + sssp-log-two-thirds-param n*) ≤
11 ∗ *sssp-log-factor n*) *at-top*
⟨*proof*⟩

theorem *bucketed-batch-param-one-third-envelope*:
eventually
(λ*n*. *real* (*2 + c*) ≤ *real* (*Suc* (*2 + c*)) ∗ *sssp-log-factor-one-third n*)

at-top
⟨*proof*⟩

42.6 Connected Headline: Bucketed-Costed BMSSP is $O(m\ log^{2/3}\ n)$

The capstone. It plugs the bucketed cost parameters directly into the recurrence solver ⟦*0 < ?Cn*; *0 < ?Ca*; *0 ≤ ?Cl*; *0 < ?Cr*; *0 < ?Ct*; *0 < ?Ch*; ∀$_F$ *n in sequentially. real n ≤ ?Cn * real (?m n)*; ∀$_F$ *n in sequentially. real (?A n) ≤ ?Ca * sssp-log-factor-one-third n*; ∀$_F$ *n in sequentially. real (?l n) ≤ ?Cl * sssp-log-factor-one-third n*; ∀$_F$ *n in sequentially. real (?R n) ≤ ?Cr * sssp-log-factor n*; ∀$_F$ *n in sequentially. real (?t n) ≤ ?Ct * sssp-log-factor n*; ∀$_F$ *n in sequentially. real (?H n) ≤ ?Ch * sssp-log-factor-one-third n*; ∀$_F$ *n in sequentially. ?T n ≤ bmssp-refined-graph-time-bound ?A ?R ?H ?l ?t ?m n*⟧ ⟹ (λ*n. real (?T n)*) ∈ *O*(*sssp-time-target ?m*). The Insert cost slot t carries the bucketed Insert budget λ*n. 9 + sssp-log-two-thirds-param n*, and the BatchPrepend cost slot H carries the bucketed BatchPrepend budget λ-. *2 + cbatch* (a constant under bounded out-degree, when the per-pull batch length stays within a constant multiple of the block size). Crucially, no cost parameter is left free: both are the measured bucketed budgets established above. The conclusion is the deterministic SSSP target $O(m * log^{2/3}\ n)$.

Recall the argument order of *bmssp-refined-graph-time-bound* is *A R H l t m n*, with body *2 A (2 l + 1) n + (R + t + l H) m*; here the fifth argument t is the Insert cost and the third argument H is the BatchPrepend per-item cost.

This theorem is what unifies the two previously separate claims of the entry: the asymptotic recurrence (claim 2) and the bucketed operation costs (claim 3). In the original development the recurrence's Insert cost was a free parameter that the headline simply set to the schedule scale; here that parameter is the actual bucketed budget, and the bound still holds.

theorem *bucketed-refined-cost-bigo-sssp-time-target*:
  fixes *m A l R T* :: *nat* ⇒ *nat*
    and *cbatch* :: *nat*
    and *Cn Ca Cl Cr* :: *real*
  assumes *Cn-pos*: *0 < Cn*
    and *Ca-pos*: *0 < Ca*
    and *Cl-nonneg*: *0 ≤ Cl*
    and *Cr-pos*: *0 < Cr*
    and *vertex-count-dominated*:
      eventually (λ*n. real n ≤ Cn * real* (*m n*)) *at-top*
    and *A-bound*:
      eventually (λ*n. real* (*A n*) ≤ *Ca * sssp-log-factor-one-third n*) *at-top*
    and *l-bound*:
      eventually (λ*n. real* (*l n*) ≤ *Cl * sssp-log-factor-one-third n*) *at-top*
    and *R-bound*:

```
    eventually (λn. real (R n) ≤ Cr * sssp-log-factor n) at-top
  and cost-bound:
    eventually
      (λn. T n ≤ bmssp-refined-graph-time-bound
        A R (λ-. 2 + cbatch) l
        (λn. 9 + sssp-log-two-thirds-param n) m n)
      at-top
 shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
⟨proof⟩
```

Specialised to the sparse-graph regime where the vertex term is dominated by the edge term, this is the familiar deterministic SSSP envelope $O(m * (ln\ n)\ powr\ (2/3))$, now carrying the bucketed Insert and BatchPrepend costs in the recurrence rather than free abstract parameters.

```
corollary bucketed-refined-cost-bigo-log-target:
 fixes m A l R T :: nat ⇒ nat
   and cbatch :: nat
   and Cn Ca Cl Cr :: real
 assumes Cn-pos: 0 < Cn
   and Ca-pos: 0 < Ca
   and Cl-nonneg: 0 ≤ Cl
   and Cr-pos: 0 < Cr
   and vertex-count-dominated:
    eventually (λn. real n ≤ Cn * real (m n)) at-top
   and A-bound:
    eventually (λn. real (A n) ≤ Ca * sssp-log-factor-one-third n) at-top
   and l-bound:
    eventually (λn. real (l n) ≤ Cl * sssp-log-factor-one-third n) at-top
   and R-bound:
    eventually (λn. real (R n) ≤ Cr * sssp-log-factor n) at-top
   and cost-bound:
    eventually
      (λn. T n ≤ bmssp-refined-graph-time-bound
        A R (λ-. 2 + cbatch) l
        (λn. 9 + sssp-log-two-thirds-param n) m n)
      at-top
 shows (λn. real (T n)) ∈
   O(λn. real (m n) * (ln (real n + 2)) powr (2 / 3))
⟨proof⟩

end
theory BMSSP-Path-Family
 imports BMSSP-Top-Level-Bounds BMSSP-Executable-Base-Case
begin
```

43 A Size-Indexed Family of Runtime Instances

The runtime headline *bmssp-runtime-headline-instance ?V ?E ?w ?s ?D* $\Longrightarrow$ (*λn. real* (*bmssp-runtime-headline-instance.T-bmssp ?V ?E ?w ?s ?D n*)) $\in$ *O*(*λn. real* (*bmssp-runtime-headline-instance.m ?E n*) $*$ *ln* (*real n* + *2*) *powr* (*2 / 3*)) is a theorem of the *bounded-reduced-positive-instance* locale. The theory `BMSSP_Runtime_Instance.thy` interprets that locale on the single fixed graph *0* $\rightarrow$ *1* $\rightarrow$ *2* $\rightarrow$ *3*, which certifies non-vacuity at one size only, and whose edge count is therefore a constant.

This theory removes that limitation. For every n it builds the unit-weight directed path *0* $\rightarrow$ *1* $\rightarrow \ldots \rightarrow$ n, proves that it satisfies all assumptions of *bounded-reduced-positive-instance uniformly in n*, and obtains the deterministic running-time bound for a family whose vertex count $n + 1$ and edge count n grow without bound. The only non-routine assumption, uniqueness of shortest walks, is discharged by the structural fact that in such a path the only walk starting at *0* is an initial segment $[0, 1, \ldots, k]$. Unlike the fixed instance, no step uses code evaluation; the argument is a single induction valid for all n.

43.1 The Path Graph of Size n

definition *pf-V* :: *nat* $\Rightarrow$ *nat set* **where**
  pf-V n = {*0*..*n*}

definition *pf-E* :: *nat* $\Rightarrow$ (*nat* $\times$ *nat*) *set* **where**
  pf-E n = {(*i*, *Suc i*) | *i*. *i* < *n*}

definition *pf-w* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *real* **where**
  pf-w u v = *1*

lemma *pf-E-iff*: (*a*, *b*) $\in$ *pf-E n* $\longleftrightarrow$ *a* < *n* $\wedge$ *b* = *Suc a*
  ⟨*proof*⟩

lemma *pf-finite-weighted-digraph*:
  finite-weighted-digraph (*pf-V n*) (*pf-E n*) *pf-w 0*
⟨*proof*⟩

43.2 The Path Family Locale

Fixing the size n in a locale lets us interpret the graph locales for *pf-V n*, *pf-E n*, while keeping n schematic across the whole development.

locale *path-family* =
  fixes *n* :: *nat*
begin

interpretation *pf*: *finite-weighted-digraph pf-V n pf-E n pf-w 0*
  ⟨*proof*⟩

43.2.1 The Only Walk From the Source Is an Initial Segment

Vertex i has the single out-edge $(i, \mathit{Suc}\ i)$, so any walk starting at 0 is forced to visit $0, 1, 2, \ldots$ in order: its i-th element is i.

lemma *pf-walk-nth*:
assumes *walk*: $\mathit{pf.walk}\ p$
and *hd0*: $\mathit{hd}\ p = 0$
and *i*: $i < \mathit{length}\ p$
shows $p\ !\ i = i$
⟨*proof*⟩

lemma *pf-walk-from-0-eq-upt*:
assumes *walk*: $\mathit{pf.walk}\ p$
and *hd0*: $\mathit{hd}\ p = 0$
shows $p = [0..<\mathit{length}\ p]$
⟨*proof*⟩

lemma *pf-simple-walk-betw-eq*:
assumes $\mathit{pf.simple\text{-}walk\text{-}betw}\ 0\ p\ v$
shows $p = [0..<\mathit{Suc}\ v]$
⟨*proof*⟩

43.2.2 Reachability and Uniqueness

lemma *pf-upt-simple-walk*:
assumes $v \leq n$
shows $\mathit{pf.simple\text{-}walk\text{-}betw}\ 0\ [0..<\mathit{Suc}\ v]\ v$
⟨*proof*⟩

lemma *pf-all-reachable*:
assumes $v \in \mathit{pf\text{-}V}\ n$
shows $\mathit{pf.reachable}\ 0\ v$
⟨*proof*⟩

lemma *pf-unique-shortest-walk*:
assumes $\mathit{pf.shortest\text{-}walk}\ 0\ p\ v$ **and** $\mathit{pf.shortest\text{-}walk}\ 0\ q\ v$
shows $p = q$
⟨*proof*⟩

With uniqueness in hand we may interpret the *unique-shortest-digraph* layer, which is where *unique-shortest-digraph.edge-outdegree-le* and *unique-shortest-digraph.outgoing-edges* live.

interpretation *pf-usd*: $\mathit{unique\text{-}shortest\text{-}digraph}\ \mathit{pf\text{-}V}\ n\ \mathit{pf\text{-}E}\ n\ \mathit{pf\text{-}w}\ 0$
⟨*proof*⟩

43.2.3 Bounded Out-Degree and Positive Weights

lemma *pf-positive-weight*: $(u, v) \in \mathit{pf\text{-}E}\ n \Longrightarrow 0 < \mathit{pf\text{-}w}\ u\ v$
⟨*proof*⟩

lemma *pf-edge-outdegree*: *pf-usd.edge-outdegree-le 1*
⟨*proof*⟩

43.2.4 The Reduced Positive Instance and Its Running-Time Bound

sublocale *pf-bri*: *bounded-reduced-positive-instance pf-V n pf-E n pf-w 0 1*
⟨*proof*⟩

The vertex count is $n + 1$ and the edge count is exactly n: both grow with the size parameter, unlike the constant edge count of the fixed instance.

lemma *pf-vertex-count*: *pf-usd.vertex-count = Suc n*
⟨*proof*⟩

lemma *pf-edge-count*: *pf-usd.edge-count = n*
⟨*proof*⟩

The closed running-time bound for the path of size n, with all locale hypotheses discharged. This is the deterministic $O(m * (ln\ N)\ powr\ (2/3))$ envelope of the headline, now holding for a member of the family of *every* size. The time and size functions are named locally so the re-exported statement below is free of deep qualified names.

definition *pf-T* :: *nat* ⇒ *nat* **where**
pf-T = pf-bri.runtime-headline.T-bmssp

definition *pf-m* :: *nat* ⇒ *nat* **where**
pf-m = pf-bri.runtime-headline.m

lemma *pf-runtime-bigo-target*:
(λ*N*. *real* (*pf-T N*)) ∈
O(λ*N*. *real* (*pf-m N*) * (*ln* (*real N* + *2*)) *powr* (*2* / *3*))
⟨*proof*⟩

Locale-level names for the graph-size measures, so the external re-exports avoid deep interpretation-qualified constants.

definition *pf-verts* :: *nat* **where**
pf-verts = pf-usd.vertex-count

definition *pf-edges* :: *nat* **where**
pf-edges = pf-usd.edge-count

lemma *pf-verts-eq*: *pf-verts = Suc n*
⟨*proof*⟩

lemma *pf-edges-eq*: *pf-edges = n*
⟨*proof*⟩

end

43.3 The Bound Holds at Every Size

Re-exported outside the locale: for every size parameter n, the path graph on $n + 1$ vertices and n edges satisfies the deterministic BMSSP running-time bound, and its edge count really is n. Because n is universally quantified, this is a single statement witnessing non-vacuity of the runtime locale across all sizes, not merely at one fixed graph.

theorem *path-family-runtime-bigo-target*:
  fixes n :: *nat*
  shows $(\lambda N.\ real\ (path\text{-}family.pf\text{-}T\ n\ N)) \in$
    $O(\lambda N.\ real\ (path\text{-}family.pf\text{-}m\ n\ N) * (ln\ (real\ N + 2))\ powr\ (2\ /\ 3))$
⟨*proof*⟩

theorem *path-family-edge-count*:
  fixes n :: *nat*
  shows *path-family.pf-edges* $n = n$
⟨*proof*⟩

theorem *path-family-vertex-count*:
  fixes n :: *nat*
  shows *path-family.pf-verts* $n = Suc\ n$
⟨*proof*⟩

end
theory *BMSSP-Bucketed-Runtime*
  imports *BMSSP-Top-Level-Bounds BMSSP-Bucketed-Cost-Bridge*
begin

44 Non-Conditional Bucketed Running-Time Bound

The capstone of `BMSSP_Bucketed_Cost_Bridge.thy` is conditional: it assumes a cost function already bounded by the refined graph-time expression with the bucketed costs in the Insert and BatchPrepend slots. This theory discharges that assumption against an actual costed BMSSP run, producing a hypothesis-free $O(m * log^{2/3}\ n)$ bound that carries the bucketed operation costs.

The key structural fact making this possible is that the local-budget amortised theorem *direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-invariants* is parametric in the level geometry *M-of/cap* and in the per-operation costs t/h *independently*. We therefore keep the level geometry at the paper's schedule exponents $p = log^{1/3}$, $q = log^{2/3}$ (so the recursion tree, pivot growth, and pull blocks are unchanged), while charging Insert at the bucketed budget $9 + q$ and BatchPrepend at $2 + p$. Because the FindPivots pivot set *find-pivots-pivots-capped* depends only on the geometry, not on the costs, the same emptiness argument that makes the existing

headline total applies here verbatim, so the bound is total in N.

Throughout, D is a fixed outdegree bound and the running graph is the locale graph of *strict-tie-breaking-digraph*.

44.1 A One-Third-Scale Envelope for the BatchPrepend Cost

The BatchPrepend cost $2 + p$ here grows like $p = log^{1/3}$ (the schedule parameter), not a constant. This envelope, the one-third analogue of the two-thirds envelope in the cost bridge, shows it stays within a constant multiple of $log^{1/3}$ — exactly the *H-bound* the recurrence solver needs.

lemma *const-plus-one-third-param-le-one-third-factor*:
  eventually
    (λ*n*. *real* (*c* + *sssp-log-one-third-param n*) ≤
      real (*c* + *2*) ∗ *sssp-log-factor-one-third n*) *at-top*
  ⟨*proof*⟩

An affine version used for the vertex factor $A = a * p + b$: it is dominated by $(2\ a + b)$ times the one-third log factor, hence $O(log^{1/3})$.

lemma *affine-one-third-param-le-one-third-factor*:
  eventually
    (λ*n*. *real* (*a* ∗ *sssp-log-one-third-param n* + *b*) ≤
      real (*2* ∗ *a* + *b*) ∗ *sssp-log-factor-one-third n*) *at-top*
  ⟨*proof*⟩

context *strict-tie-breaking-digraph*
begin

44.2 A Decoupled Amortised Bound: Geometry Exponent vs. Operation Cost

This is the existing level-cap amortised theorem with one generalisation: the level geometry uses an exponent *tg* that is decoupled from the per-Insert cost *t*. In the original theorem the two coincide (*M-of* = *bmssp-level-cap k t*); here *M-of* = *bmssp-level-cap k tg*, while *t* remains the cost charged at each direct insert. The proof is the original one: the local-budget amortised lemma is parametric in *M-of*, *cap*, *t*, *h*, *A*, *R*, *k*, so the only coupling needed is *M-of i* ≤ *cap* for $i \le l$, which still holds by monotonicity of *bmssp-level-cap* in the level.

theorem *finite-initial-label-decoupled-amortized*:
  assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
    and *degree*: *edge-outdegree-le* Δ
    and *degree-factor*: Δ ≤ *A*
    and *R-pos*: *0* < *R*
    and *insert-factor*: *t* ≤ *A* ∗ *k*
    and *insert-scaled-factor*: *t* ≤ *A-insert* ∗ *k*
    and *seen-scaled-factor*: *k* ∗ Δ + *A-insert* ≤ *2* ∗ *A*

and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k tg*) *t h k*
(*bmssp-level-cap k tg l*) *l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *h*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

44.3 A Costed Run With Decoupled Geometry and Bucketed Costs

bmssp-bucketed-run D N is a top-level costed run whose level geometry uses the schedule exponents *p*, *q* but whose Insert cost is the bucketed budget *9* + *q* and whose BatchPrepend cost is *2* + *p*.

definition *bmssp-bucketed-run* ::
nat ⇒ *nat* ⇒ (′*a* ⇒ *real*) ⇒ ′*a set* ⇒ *nat* ⇒ *bool* **where**
bmssp-bucketed-run D N d′ U c ⟷
(*let p* = *sssp-log-one-third-param N*;
q = *sssp-log-two-thirds-param N*
in exact-concrete-bmssp D (*bmssp-level-cap p q*) (*9* + *q*) (*2* + *p*) *p*
(*bmssp-level-cap p q p*) *p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*)

definition *bmssp-bucketed-cost* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
bmssp-bucketed-cost D N c ⟷ (∃ *d′ U*. *bmssp-bucketed-run D N d′ U c*)

definition *bmssp-bucketed-time* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bmssp-bucketed-time D N = (*LEAST c*. *bmssp-bucketed-cost D N c*)

44.4 Totality of the Bucketed Cost

The same FindPivots-emptiness argument as the existing headline, with the bucketed costs in the cost slots. Since *exact-concrete-bmssp-Suc-exists-if-pivots-empty-same-bound* is parametric in the costs, the substitution is immediate.

lemma *bmssp-bucketed-cost-exists-if-top-pivots-empty*:
assumes *pivots-empty*:
find-pivots-pivots-capped (*sssp-log-one-third-param N*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
finite-initial-label {*s*} *Infinity* = {}
shows ∃ *c*. *bmssp-bucketed-cost D N c*
⟨*proof*⟩

lemma *eventually-bmssp-bucketed-cost*:

eventually ($\lambda N.\ \exists\, c.$ *bmssp-bucketed-cost D N c*) *at-top*
⟨*proof*⟩

lemma *bmssp-bucketed-time-witness*:
assumes *bmssp-bucketed-cost D N c*
shows *bmssp-bucketed-cost D N* (*bmssp-bucketed-time D N*)
⟨*proof*⟩

44.5 Cost Bound for a Bucketed Run

A bucketed run refines a direct-insert run with the same (decoupled) geometry and costs, to which the decoupled amortised theorem applies. Choosing $A =$ *Suc D* $*$ *p* $+$ *9* makes the local-budget side conditions hold for *all N* (the binding constraint $9 + q \leq A * p$ follows from $q \leq p^2$ and $9 \leq 9 * p$), and A still grows only like $log^{1/3}$.

lemma *bmssp-bucketed-run-refined-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*: *bmssp-bucketed-run D N d′ U c*
shows *c* $\leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* $*$ *sssp-log-one-third-param N* $+$ *9*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *2* $+$ *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *9* $+$ *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
⟨*proof*⟩

44.6 The Non-Conditional Bucketed Running-Time Bound

Assembling the pieces. Totality of the bucketed cost (via the FindPivots-emptiness argument) discharges the existence premise; the cost bound discharges the recurrence premise; and the schedule envelopes for the five cost parameters discharge the component bounds. Nothing is left conditional: the bucketed BMSSP running time is $O(m * log^{2/3}\ n)$.

theorem *bmssp-bucketed-time-bigo-sssp-time-target*:
fixes *m* :: *nat* $\Rightarrow$ *nat*
and *Cn Cm* :: *real*
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *Cn-pos*: *0* $<$ *Cn*
and *Cm-pos*: *0* $<$ *Cm*
and *vc-dom*: *eventually* ($\lambda n.$ *real vertex-count* $\leq$ *Cn* $*$ *real* (*m n*)) *at-top*
and *ec-dom*: *eventually* ($\lambda n.$ *real edge-count* $\leq$ *Cm* $*$ *real* (*m n*)) *at-top*
shows ($\lambda n.$ *real* (*bmssp-bucketed-time D n*)) $\in$ *O*($\lambda n.$ *sssp-time-target m n*)
⟨*proof*⟩

The sparse-graph specialisation, fully closed: for the bounded-outdegree

locale graph with at least one edge, the bucketed BMSSP running time is $O(m * (ln\ n)\ powr\ (2/3))$, where the Insert and BatchPrepend costs charged inside the recurrence are the actual bucketed budgets, not free parameters.

theorem *bmssp-bucketed-runtime-bigo-target*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow reachable\ s\ v$
    and *degree*: *edge-outdegree-le D*
    and *edge-count-pos*: *0 < edge-count*
  shows $(\lambda n.\ real\ (bmssp\text{-}bucketed\text{-}time\ D\ n)) \in$
    $O(\lambda n.\ real\ edge\text{-}count * (ln\ (real\ n + 2))\ powr\ (2\ /\ 3))$
⟨*proof*⟩

end

end

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. DOI: `https://doi.org/10.1007/BF01386390`.

[2] R. Duan, J. Mao, X. Mao, X. Shu, and L. Yin. Breaking the sorting barrier for directed single-source shortest paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, 2025. DOI: `https://doi.org/10.1145/3717823.3718179`.

[3] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. DOI: `https://doi.org/10.1145/28869.28874`.