Runtime Extensions for BMSSP

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

November 5, 2026

Abstract

This entry extends the AFP formalization *Correctness and Complexity of BMSSP in Isabelle/HOL* [2], which formalizes the bounded multi-source shortest path algorithm of Duan, Mao, Mao, Shu, and Yin [1]. The extension adds a charged direct-insert relation that records both the range charged by the recurrence and the semantic output returned by each recursive child. It proves that least costs are realized by verified-correct runs, exhibits a closed branching instance, and records a machine-checked totality obstruction for the older diagonal exact-concrete path-family relation. Its main theorem, `path_nonvac_runtime_bigo_card_V_unconditional`, proves without run-existence, cost-bound, or closed-bound assumptions that the least charged cost on the growing unit-weight path family is $O(|V_n| \log^{2/3} |V_n|)$ when the internal schedule is evaluated at n^2. This is a cost-relation theorem, not a machine-step bound for the SML executable from the base entry.

AI assistance was used for proof engineering.

Contents

1 Relationship to the Published Entry **2**

2 The Unconditional Size-Parametric Headline **3**

3 Verified Runtime and the Obstruction Record **3**

4 Scope **4**

5 Checked Source **4**

6 Runtime Bound for a Verified-Correct BMSSP Run **55**

7 A Second, Structurally Different Non-Vacuous Instance **59**

8 A Size-Parametric BMSSP Running-Time Bound over a Growing Graph Family 62
8.1 The path family 62
8.2 Structural and cardinality lemmas 62
8.3 Easy locale obligations, uniform in k 63
8.4 The unconditional cost-recurrence bound (FLOOR) 64
8.5 The forced-degenerate (empty-pivot) regime is provably avoided 65
8.6 Unique shortest walks on the path family 66
8.7 The exact-concrete algorithm-level running-time bound 73
8.8 A charged path-family theorem waiting on charged cost bounds 75

9 On the Totality of the Exact-Concrete Cost Relation 77

10 An unconditional, non-vacuous size-parametric BMSSP running-time bound 116
10.1 Elementary analytic facts 116
10.2 A reusable "the source is the sole pivot" brick 118
10.3 The genuine shortest-path distance on the path family 120
10.4 A schedule that dominates the path size (non-vacuous regime) 120
10.5 The bucket cap dominates the path size at the inflated schedule 170
10.6 The inflated schedule index stays below the graph size 173
10.7 The inflated schedule is genuine and keeps the polylog in $\Theta(ln\ n)$ 175
10.8 The decoupled charged running time of the path family . . . 176
10.9 The decoupled charged headline (conditional skeleton for **B3**) 178
10.10 Dropping the run-existence premise (B1, unconditional in runs) 183
10.11 Charged-cost upper bound via the uncharged refinement (B2, uncharged route) 188
10.12 The unconditional non-vacuous size-parametric running-time bound 193
10.13 Charged closed-bound engine (B2): the charged level recurrence solver 195

1 Relationship to the Published Entry

The base entry [2] supplies finite weighted digraph semantics, BMSSP pre- and postconditions, the recursive and exact-concrete cost relations, logarithmic schedule bounds, a bucketed partition model, and executable correctness. This entry is a child session of `BMSSP_Correctness`. Its first three theories add only the new charged relation and lemmas required by the follow-up; all other theories import those extensions and the published AFP theories.

Theory	Added surface
`BMSSP_Direct_Insert_Extensions`	Charged direct-insert run and loop relations, correctness, existence, and child-cost accounting.
`BMSSP_Strict_Tie_Breaking_Extensions`	Charged progress and level-bound closers under strict tie breaking.
`BMSSP_Top_Level_Bounds_Extensions`	Charged top-level run, cost, least-time, correctness, and refined-bound interfaces.
`BMSSP_Verified_Runtime`	Joins the existing asymptotic least-cost theorem with correctness of a run realizing that least cost.
`BMSSP_Runtime_Instance_Branching`	A closed out-star instance of the verified-runtime theorem.
`BMSSP_Path_Family_Extended`	The growing unit-weight path family, the unconditional cost-expression floor, and conditional diagonal runtime statements.
`BMSSP_Cost_Totality`	The exact-concrete diagonal totality obstruction.
`BMSSP_NonVacuous_Family`	The decoupled charged family and unconditional non-vacuous runtime headline.

2 The Unconditional Size-Parametric Headline

For $n \in \mathbb{N}$, let P_n be the directed unit-weight path with vertices $0, \ldots, n$. The development proves

`path_nonvac_runtime_bigo_card_V_unconditional`

with the asymptotic meaning

$$T(n) \in O\Big(|V_n|\big(\ln(|V_n|+2)\big)^{2/3}\Big),$$

where $T(n) =$ `path_nonvac_time` n is the least charged direct-insert top-level cost for P_n at internal schedule n^2.

The statement has no assumptions. Its non-vacuity is explicit: `eventually_path_nonvac_charged_cost_exists` proves that charged runs exist for all but finitely many n, and `path_k_charged_cheap_imp_nonvac_time_le_bound` constructs a charged cost below the refined graph-time budget. The square schedule leaves the target logarithmic class unchanged because $\ln(n^2) = 2\ln n$, while making the level cap eventually large enough to cover the path.

3 Verified Runtime and the Obstruction Record

The theorem `verified_bmssp_runtime_bigo_target` strengthens the base entry's least-cost asymptotic statement by proving that the least cost is eventually realized by an exact-concrete top-level run whose completed set is the whole graph and whose output satisfies `sssp_correct`. The theory

`BMSSP_Runtime_Instance_Branching` closes this statement on an out-star whose source has two incomparable outgoing shortest paths.

Separately, `BMSSP_Cost_Totality` proves the structural obstruction in the older diagonal exact-concrete loop relation: a genuine step is forced to the initial prefix under a sound label, and no genuine step can begin at a positive lower bound. The conditional diagonal family theorem is therefore not used as evidence for non-vacuity. The unconditional headline instead uses the charged relation and decouples graph size from the internal schedule.

4 Scope

The imported base entry remains the source of executable `bmssp_distances` and its functional-correctness theorem. This entry does not claim that generated SML executes within the asymptotic cost bound. The runtime statements concern the formal cost relations and their graph-size-parametric instances.

5 Checked Source

theory *BMSSP-Direct-Insert-Extensions*
imports *BMSSP-Correctness.BMSSP-Direct-Insert-Costed*
begin

context *unique-shortest-digraph*
begin

inductive *charged-direct-insert-costed-partition-loop-state*
and *charged-direct-insert-costed-bmssp* **where**
Charged-Direct-Insert-State-Done:
keys-of $D = \{\} \Longrightarrow$
bound-le (*Fin a*) $B \Longrightarrow$
complete-on d' (*bound-tree* P B) $\Longrightarrow$
charged-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*) $\cup \bigcup$ (*set* [*range-tree* P *a* B])) *0* []
| *Charged-Direct-Insert-State-Stop*:
bound-le (*Fin a*) $B \Longrightarrow$
complete-on d' (*bound-tree* P (*Fin a*)) $\Longrightarrow$
$k * cap \leq$ *card* (*bound-tree* P (*Fin a*)) $\Longrightarrow$
charged-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*) $\cup \bigcup$ (*set* [*range-tree* P *a* (*Fin a*)])) *0* []
| *Charged-Direct-Insert-State-Step*:
pull-separates D (*M-of l*) *Bmax S-pull beta D-pull* $\Longrightarrow$
bound-le (*Fin beta*) $B \Longrightarrow$
bmssp-pre-full d S-pull (*Fin beta*) $\Longrightarrow$

$S\text{-}pull = split\text{-}below\ d\ P\ beta \Longrightarrow$
$(\forall\, x{\in}S\text{-}pull.\ reachable\ s\ x) \Longrightarrow$
$a \leq b \Longrightarrow$
$bound\text{-}le\ (Fin\ a)\ B' \Longrightarrow$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a)) \Longrightarrow$
$card\ (bound\text{-}tree\ P\ (Fin\ a)) < k * cap \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ S\text{-}pull\ (Fin\ beta)$
$d\text{-}child\ (Fin\ b)\ U\text{-}child\ c\text{-}child \Longrightarrow$
$charged\text{-}child = range\text{-}tree\ P\ a\ (Fin\ b) \Longrightarrow$
$complete\text{-}preserved\ d\text{-}child\ d'\ U\text{-}child \Longrightarrow$
$direct\text{-}edge\text{-}batch =$
$edge\text{-}relaxation\text{-}pairs\text{-}in\text{-}bound\ d\text{-}child\ charged\text{-}child\ beta\ B \Longrightarrow$
$lower\text{-}edge\text{-}batch =$
$edge\text{-}relaxation\text{-}pairs\text{-}between\ d\text{-}child\ charged\text{-}child\ b\ beta \Longrightarrow$
$source\text{-}batch = label\text{-}pairs\text{-}between\ d\ S\text{-}pull\ b\ beta \Longrightarrow$
$batch = direct\text{-}edge\text{-}batch\ @\ lower\text{-}edge\text{-}batch\ @\ source\text{-}batch \Longrightarrow$
$D\text{-}next = batch\text{-}min\text{-}update\ D\text{-}pull\ batch \Longrightarrow$
$partition\text{-}pull\text{-}cost\text{-}bound\ c\text{-}pull\ S\text{-}pull \Longrightarrow$
$partition\text{-}batch\text{-}cost\text{-}bound\ c\text{-}direct\ t\ direct\text{-}edge\text{-}batch \Longrightarrow$
$partition\text{-}batch\text{-}cost\text{-}bound\ c\text{-}lower\ h\ lower\text{-}edge\text{-}batch \Longrightarrow$
$partition\text{-}batch\text{-}cost\text{-}bound\ c\text{-}sources\ h\ source\text{-}batch \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P\ B\ d'$
$D\text{-}next\ b\ betas\ bs\ B'\ charged\text{-}tail\ child\text{-}outputs\text{-}tail\ U\text{-}tail\ c\text{-}tail$
$child\text{-}costs\text{-}tail \Longrightarrow$
$c = c\text{-}pull + c\text{-}direct + c\text{-}lower + c\text{-}sources + c\text{-}child + c\text{-}tail \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
$(beta\ \#\ betas)\ (b\ \#\ bs)\ B'$
$(charged\text{-}child\ \#\ charged\text{-}tail)\ (U\text{-}child\ \#\ child\text{-}outputs\text{-}tail)$
$(bound\text{-}tree\ P\ (Fin\ a) \cup \bigcup (set\ (charged\text{-}child\ \#\ charged\text{-}tail)))\ c$
$(c\text{-}child\ \#\ child\text{-}costs\text{-}tail)$

| *Charged-Direct-Insert-Base*:
$S = \{x\} \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ 0\ d\ S\ B$
$(\lambda v.\ if\ v \in base\text{-}case\text{-}vertices\ k\ x\ B\ then\ dist\ s\ v\ else\ d\ v)$
$(base\text{-}case\text{-}bound\ k\ x\ B)$
$(base\text{-}case\text{-}vertices\ k\ x\ B)$
$(base\text{-}case\text{-}scan\text{-}cost\ \Delta\ k\ x\ B)$

| *Charged-Direct-Insert-Step*:
$D = label\text{-}partition\text{-}view$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'$
$charged\text{-}Us\ child\text{-}outputs\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs\text{-}loop \Longrightarrow$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$

$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$c = fp\text{-}iter\text{-}capped\text{-}scan\text{-}cost\ k\ cap\ d\ S\ S\ B + c\text{-}insert + c\text{-}loop \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ (Suc\ l)\ d\ S\ B\ d'\ B'\ U\ c$

lemma *charged-direct-insert-partition-loop-state-step-done-exists*:
fixes *a b beta Bmax* :: *real*
and *B* :: *bound*
and *D D-pull D-next* :: *'a partition-view*
and *M-of* :: *nat* ⇒ *nat*
and Δ *t h k cap l c-pull c-direct c-lower c-sources c-child* :: *nat*
and *d d' d-child* :: *'a* ⇒ *real*
and *P S-pull U-child* :: *'a set*
assumes *pull*: *pull-separates D* (*M-of l*) *Bmax S-pull beta D-pull*
and *beta-bound*: *bound-le* (*Fin beta*) *B*
and *pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
and *split*: *S-pull* = *split-below d P beta*
and *reaches*: ∀ *x*∈*S-pull*. *reachable s x*
and *a-le-b*: *a* ≤ *b*
and *a-bound*: *bound-le* (*Fin a*) *B*
and *complete-a*: *complete-on d'* (*bound-tree P* (*Fin a*))
and *small*: *card* (*bound-tree P* (*Fin a*)) < *k* ∗ *cap*
and *child*:
charged-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*
and *preserved*: *complete-preserved d-child d' U-child*
and *D-next*:
D-next = *batch-min-update D-pull*
(*edge-relaxation-pairs-in-bound d-child* (*range-tree P a* (*Fin b*)) *beta B* @
edge-relaxation-pairs-between d-child (*range-tree P a* (*Fin b*)) *b beta* @
label-pairs-between d S-pull b beta)
and *c-pull*: *partition-pull-cost-bound c-pull S-pull*
and *c-direct*:
partition-batch-cost-bound c-direct t
(*edge-relaxation-pairs-in-bound d-child* (*range-tree P a* (*Fin b*)) *beta B*)
and *c-lower*:
partition-batch-cost-bound c-lower h
(*edge-relaxation-pairs-between d-child* (*range-tree P a* (*Fin b*)) *b beta*)
and *c-sources*:
partition-batch-cost-bound c-sources h (*label-pairs-between d S-pull b beta*)
and *tail-empty*: *keys-of D-next* = {}
and *b-bound*: *bound-le* (*Fin b*) *B*
and *complete-B*: *complete-on d'* (*bound-tree P B*)
shows
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B*
d' D a [*beta*] [*b*] *B* [*range-tree P a* (*Fin b*), *range-tree P b B*]
[*U-child*]
(*bound-tree P* (*Fin a*) ∪
⋃(*set* [*range-tree P a* (*Fin b*), *range-tree P b B*]))

```
    (c-pull + c-direct + c-lower + c-sources + c-child + 0) [c-child]
proof −
  have tail:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B
      d′ D-next b [] [] B [range-tree P b B] []
      (bound-tree P (Fin b) ∪ ⋃(set [range-tree P b B])) 0 []
    by (rule Charged-Direct-Insert-State-Done)
      (use tail-empty b-bound complete-B in auto)
  show ?thesis
    by (rule Charged-Direct-Insert-State-Step
        [where D-pull = D-pull
          and direct-edge-batch =
            edge-relaxation-pairs-in-bound d-child (range-tree P a (Fin b)) beta B
          and lower-edge-batch =
            edge-relaxation-pairs-between d-child (range-tree P a (Fin b)) b beta
          and source-batch = label-pairs-between d S-pull b beta
          and batch =
            edge-relaxation-pairs-in-bound d-child (range-tree P a (Fin b)) beta B @
             edge-relaxation-pairs-between d-child (range-tree P a (Fin b)) b beta @
             label-pairs-between d S-pull b beta
          and D-next = D-next])
      (use pull beta-bound pre split reaches a-le-b a-bound complete-a small child
        preserved D-next c-pull c-direct c-lower c-sources tail in auto)
qed

theorem direct-insert-costed-refines-charged-direct-insert:
  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 ⟹
   charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
    betas bs B′ Us (range-tree-child-list P a bs) U c child-costs
and direct-insert-costed-bmssp-refines-charged-direct-insert:
  direct-insert-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c ⟹
   charged-direct-insert-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c
proof (induction rule:
    direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
  case (Direct-Insert-Costed-State-Done D a B d′ P Δ M-of t h k cap l d)
  have run:
    charged-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 []
    by (rule Charged-Direct-Insert-State-Done)
      (use Direct-Insert-Costed-State-Done in auto)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Stop a B d′ P k cap Δ M-of t h l d D)
  have run:
    charged-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 []
   by (rule Charged-Direct-Insert-State-Stop)
    (use Direct-Insert-Costed-State-Stop in auto)
 then show ?case
   by simp
next
 case (Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d
    P a b B' d' k cap Δ t h d-child U-child c-child direct-edge-batch
    lower-edge-batch source-batch batch D-next c-pull c-direct c-lower
    c-sources betas bs Us-tail U-tail c-tail child-costs-tail c)
 have U-child-eq: U-child = range-tree P a (Fin b)
   using Direct-Insert-Costed-State-Step by blast
 have run:
   charged-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'
    (U-child # Us-tail) (U-child # range-tree-child-list P b bs)
    (bound-tree P (Fin a) ∪ ⋃(set (U-child # Us-tail)))
    (c-pull + c-direct + c-lower + c-sources + c-child + c-tail)
    (c-child # child-costs-tail)
   by (rule Charged-Direct-Insert-State-Step
    [where batch = batch and D-next = D-next and charged-child = U-child])
    (use Direct-Insert-Costed-State-Step in auto)
 then show ?case
   using U-child-eq Direct-Insert-Costed-State-Step by simp
next
 case (Direct-Insert-Costed-Base S x Δ M-of t h k cap d B)
 then show ?case
   by (rule Charged-Direct-Insert-Base)
next
 case (Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l d' a
    betas bs B' Us U-loop c-loop child-costs-loop U c)
 show ?case
   by (rule Charged-Direct-Insert-Step
    [where D = D and c-insert = c-insert and a = a and betas = betas
     and bs = bs and charged-Us = Us
     and child-outputs =
       range-tree-child-list (find-pivots-pivots-capped k cap d S B) a bs
     and U-loop = U-loop and c-loop = c-loop
     and child-costs-loop = child-costs-loop])
    (use Direct-Insert-Costed-Step in auto)
qed

theorem charged-direct-insert-costed-partition-loop-state-child-call-charged-ranges:
 charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' charged-Us child-outputs U c child-costs ⟹
  length child-costs = length bs ∧
  length child-outputs = length bs ∧
  list-all2
    (λc-child U-child. ∃ S-child B-child d-child B-child'.
```

charged-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 child-outputs $\wedge$
list-all2
$(\lambda$*c-child charged-child.* $\exists$ *U-child S-child B-child d-child B-child′.*
charged-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*) $\wedge$
charged-Us = *range-tree-chain-list P a bs B′*
and *charged-direct-insert-costed-bmssp-child-call-charged-ranges-trivial*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
proof (*induction rule*:
charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d*
P a b B′ d′ k cap Δ *t h d-child U-child c-child charged-child*
direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
c-direct c-lower c-sources betas bs charged-tail child-outputs-tail
U-tail c-tail child-costs-tail c)
have *card-pull*: *card S-pull* $\leq$ *M-of l*
using *Charged-Direct-Insert-State-Step*
unfolding *pull-separates-def* **by** *blast*
have *below-pull*: $\forall\, x{\in}$*S-pull. below-bound* $(d\ x)$ (*Fin beta*)
proof
fix *x*
assume *xS*: *x* $\in$ *S-pull*
have *S-pull* = *split-below d P beta*
using *Charged-Direct-Insert-State-Step* **by** *blast*
then show *below-bound* $(d\ x)$ (*Fin beta*)
using *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *head*:
$\exists$ *S-child B-child d-child′ B-child′.*

charged-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*$)$
using *Charged-Direct-Insert-State-Step card-pull below-pull* **by** *blast*
have *head-charged*:
$\exists$ *U-child′ S-child B-child d-child′ B-child′.*
charged-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*$)$
using *head* **by** *blast*
show *?case*
using *Charged-Direct-Insert-State-Step head head-charged* **by** *simp*
next
case (*Charged-Direct-Insert-Base S x* Δ *M-of t h k cap d B*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-Step D k cap d S B c-insert t* Δ *M-of h l d′ a*
betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c)
then show *?case*
by *simp*
qed

theorem *charged-direct-insert-costed-partition-loop-state-trace*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ charged-Us child-outputs 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′ charged-Us U
and *charged-direct-insert-costed-bmssp-correct*:
charged-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 (*induction rule*:
charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
have *a-le*: *bound-le* (*Fin a*) *B*
using *Charged-Direct-Insert-State-Done* **by** *blast*
have *complete-B*: *complete-on d′* (*bound-tree P B*)

```
    using Charged-Direct-Insert-State-Done by blast
  have lower: complete-on d' (bound-tree P (Fin a))
    by (rule complete-on-subset[OF complete-B])
      (rule bound-tree-bound-mono[OF a-le])
  have range-complete: complete-on d' (range-tree P a B)
    by (rule complete-on-subset[OF complete-B])
      (rule range-tree-subset-bound-tree)
  have B-le: bound-le B B
    by (cases B) simp-all
  show ?case
    unfolding concrete-partition-loop-trace-def
    using B-le a-le lower range-complete by simp
next
  case (Charged-Direct-Insert-State-Stop a B d' P k cap Δ M-of t h l d D)
  then show ?case
    unfolding concrete-partition-loop-trace-def complete-on-def by simp
next
  case (Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child charged-child
      direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
      c-direct c-lower c-sources betas bs charged-tail child-outputs-tail
      U-tail c-tail child-costs-tail c)
  have pre-beta: bmssp-pre-full d P (Fin beta)
    by (rule bmssp-pre-full-bound-mono
      [OF Charged-Direct-Insert-State-Step.prems(2)])
      (use Charged-Direct-Insert-State-Step.hyps in simp)
  have cover-beta: complete-tree-cover d P (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pre-beta] .
  have child-post:
    bmssp-post-full d S-pull (Fin beta) d-child (Fin b) U-child
    using Charged-Direct-Insert-State-Step by metis
  have child-post-split:
    bmssp-post-full d (split-below d P beta) (Fin beta) d-child (Fin b) U-child
    using child-post Charged-Direct-Insert-State-Step.hyps by simp
  have U-child-tree: U-child = bound-tree P (Fin b)
    using pull-recursive-post-lifts-bound-tree[OF cover-beta child-post-split] .
  have complete-child: complete-on d-child U-child
    using child-post unfolding bmssp-post-full-def by blast
  have complete-child':
    complete-on d' U-child
    using complete-child Charged-Direct-Insert-State-Step
    unfolding complete-preserved-def by metis
  have head-complete: complete-on d' charged-child
    by (rule complete-on-subset[OF complete-child'])
      (use U-child-tree Charged-Direct-Insert-State-Step
        range-tree-subset-bound-tree in metis)
  have tail-trace:
    concrete-partition-loop-trace P B b bs d' B' charged-tail U-tail
    using Charged-Direct-Insert-State-Step by metis
```

then have *tail-le*: *bound-le B′ B*
and *tail-mono*: *nondecreasing-from b bs*
and *tail-bounds*: *bounds-le B′ (b # bs)*
and *tail-children*:
list-all2 (λU X. U = X ∧ complete-on d′ U) charged-tail
(range-tree-chain-list P b bs B′)
using *tail-trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast+*
have *lower-complete*: *complete-on d′ (bound-tree P (Fin a))*
using *Charged-Direct-Insert-State-Step* **by** *metis*
have *charged-eq*: *charged-child = range-tree P a (Fin b)*
using *Charged-Direct-Insert-State-Step* **by** *metis*
show *?case*
unfolding *concrete-partition-loop-trace-def*
using *tail-le tail-mono tail-bounds tail-children head-complete lower-complete charged-eq*
Charged-Direct-Insert-State-Step.hyps
by *simp*
next
case (*Charged-Direct-Insert-Base S x Δ M-of t h k cap d B*)
have *post*:
bmssp-post 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)
using *base-case-result-bmssp-post*[*OF Charged-Direct-Insert-Base.hyps*,
where *k = k* **and** *B = B* **and** *d = d*]
unfolding *base-case-result-def* **by** *simp*
then show *?case*
by (*rule bmssp-post-imp-post-full*)
next
case (*Charged-Direct-Insert-Step D k cap d S B c-insert t Δ M-of h l d′ a betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c*)
let *?d-fp = find-pivots-label-capped k cap d S B*
let *?P = find-pivots-pivots-capped k cap d S B*
let *?W = {v ∈ bound-tree S B′. ?d-fp v = dist s v}*
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF Charged-Direct-Insert-Step.prems*(*1*) *Charged-Direct-Insert-Step.prems*(*3*)])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF Charged-Direct-Insert-Step.prems*(*1*) *Charged-Direct-Insert-Step.prems*(*2*) *Charged-Direct-Insert-Step.prems*(*3*)] **.**
have *P-subset-S*: *?P ⊆ S*
unfolding *find-pivots-pivots-capped-def* **by** *auto*
have *P-reaches*: *⋀x. x ∈ ?P ⟹ reachable s x*
using *P-subset-S Charged-Direct-Insert-Step.prems*(*3*) **by** *blast*
have *trace*:
concrete-partition-loop-trace ?P B a bs d′ B′ charged-Us U-loop

```
    using Charged-Direct-Insert-Step.IH sound-fp pivot-pre P-reaches by blast
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have le: bound-le B' B
    using loop-post unfolding bmssp-post-full-def by blast
  have U-tree: U = bound-tree S B'
  proof −
    have U-loop ∪ ?W = bound-tree S B'
      by (rule concrete-capped-find-pivots-final-tree-assembly
        [OF Charged-Direct-Insert-Step.prems(1)
          Charged-Direct-Insert-Step.prems(2)
          Charged-Direct-Insert-Step.prems(3) loop-post])
    then show ?thesis
      using Charged-Direct-Insert-Step(6) by simp
  qed
  have loop-complete: complete-on d' U-loop
    using loop-post unfolding bmssp-post-full-def by blast
  have U-complete: complete-on d' U
    using complete-on-Un[OF loop-complete Charged-Direct-Insert-Step(5)]
      Charged-Direct-Insert-Step(6) by simp
  show ?case
    using le U-tree U-complete unfolding bmssp-post-full-def by blast
qed

lemma charged-direct-insert-costed-bmssp-step-with-insert-cost:
  assumes init:
      D = label-partition-view
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B)
    and insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
    and loop:
      charged-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'
        charged-Us child-outputs 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}
    and c:
      c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop
  shows charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U
c
  using assms by (rule Charged-Direct-Insert-Step)
```

lemma *charged-direct-insert-costed-bmssp-Suc-exists-from-initial-loop-at-bound*:
assumes *loop*:
∃ *d′ a betas bs charged-Us child-outputs U-loop c-loop child-costs-loop*.
charged-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′*
(*label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*))
a betas bs B′ charged-Us child-outputs 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*} ∧
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
shows ∃ *d′ U c*.
charged-direct-insert-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*
proof −
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?D* = *label-partition-view ?d-fp ?P*
obtain *d′ a betas bs charged-Us child-outputs U-loop c-loop*
child-costs-loop **where**
loop-run:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
?d-fp ?P B d′ ?D a betas bs B′ charged-Us child-outputs U-loop c-loop
child-costs-loop
and *complete*:
complete-on d′ {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
and *insert*:
partition-initial-insert-cost-bound c-insert t ?P
using *loop* **by** *blast*
let *?U* = *U-loop* ∪ {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
have *run*:
charged-direct-insert-costed-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* + *c-insert* + *c-loop*)
by (*rule charged-direct-insert-costed-bmssp-step-with-insert-cost*
[*OF refl insert loop-run complete refl refl*])
then show *?thesis*
by *blast*
qed

lemma *charged-direct-insert-costed-bmssp-Suc-exists-from-initial-loop-at-bound-with-insert*:
assumes *loop*:
∃ *d′ a betas bs charged-Us child-outputs U-loop c-loop*
child-costs-loop c-insert.
charged-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'
      (label-partition-view
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B))
      a betas bs B' charged-Us child-outputs 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} ∧
    partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
 shows ∃ d' U c.
   charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
proof −
 obtain c-insert where
   ∃ d' a betas bs charged-Us child-outputs U-loop c-loop
       child-costs-loop.
     charged-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'
       (label-partition-view
         (find-pivots-label-capped k cap d S B)
         (find-pivots-pivots-capped k cap d S B))
       a betas bs B' charged-Us child-outputs 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} ∧
     partition-initial-insert-cost-bound c-insert t
       (find-pivots-pivots-capped k cap d S B)
   using loop by blast
 then show ?thesis
   by (rule charged-direct-insert-costed-bmssp-Suc-exists-from-initial-loop-at-bound)
qed

lemma charged-direct-insert-costed-bmssp-Suc-exists-from-initial-loop:
 assumes loop:
    ∃ d' a betas bs B' charged-Us child-outputs U-loop c-loop child-costs-loop.
      charged-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'
        (label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B))
        a betas bs B' charged-Us child-outputs 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} ∧
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
 shows ∃ d' B' U c.
```

charged-direct-insert-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*
proof −
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?D* = *label-partition-view ?d-fp ?P*
obtain *d′ a betas bs B′ charged-Us child-outputs U-loop c-loop*
child-costs-loop **where**
loop-run:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
?d-fp ?P B d′ ?D a betas bs B′ charged-Us child-outputs U-loop c-loop
child-costs-loop
and *complete*:
complete-on d′ {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
and *insert*:
partition-initial-insert-cost-bound c-insert t ?P
using *loop* **by** *blast*
let *?U* = *U-loop* ∪ {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
have *run*:
charged-direct-insert-costed-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* + *c-insert* + *c-loop*)
by (*rule charged-direct-insert-costed-bmssp-step-with-insert-cost*
[*OF refl insert loop-run complete refl refl*])
then show *?thesis*
by *blast*
qed

lemma *charged-direct-insert-partition-loop-state-done-exists*:
assumes *empty*: *keys-of D* = {}
and *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d′* (*bound-tree P B*)
shows ∃ *betas bs B′ charged-Us child-outputs U c child-costs*.
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
d P B d′ D a betas bs B′ charged-Us child-outputs U c child-costs
proof −
have *run*:
charged-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* []
by (*rule Charged-Direct-Insert-State-Done*[*OF empty a-bound complete*])
then show *?thesis*
by *blast*
qed

lemma *charged-direct-insert-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* ≤ *card* (*bound-tree P* (*Fin a*))
shows ∃ *betas bs B′ charged-Us child-outputs U c child-costs*.
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*

```
    d P B d' D a betas bs B' charged-Us child-outputs U c child-costs
proof −
  have run:
    charged-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 []
    by (rule Charged-Direct-Insert-State-Stop[OF a-bound complete threshold])
  then show ?thesis
    by blast
qed

lemma charged-direct-insert-partition-loop-state-step-exists:
  assumes pull: pull-separates D (M-of l) Bmax 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:
      charged-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
    and preserved: complete-preserved d-child d' U-child
    and tail:
      charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
        d P B d'
        (batch-min-update D-pull
          (edge-relaxation-pairs-in-bound d-child (range-tree P a (Fin b)) beta B @
           edge-relaxation-pairs-between d-child (range-tree P a (Fin b)) b beta @
           label-pairs-between d S-pull b beta))
        b betas bs B' charged-tail child-outputs-tail U-tail c-tail
        child-costs-tail
  shows ∃ c child-costs.
    charged-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) # charged-tail)
      (U-child # child-outputs-tail)
      (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # charged-tail))) c child-costs
proof −
  let ?charged-child = range-tree P a (Fin b)
  let ?direct = edge-relaxation-pairs-in-bound d-child ?charged-child beta B
  let ?lower = edge-relaxation-pairs-between d-child ?charged-child b beta
  let ?sources = label-pairs-between d S-pull b beta
  let ?batch = ?direct @ ?lower @ ?sources
  have pull-cost: partition-pull-cost-bound 0 S-pull
    unfolding partition-pull-cost-bound-def by simp
```

```
  have direct-cost: partition-batch-cost-bound 0 t ?direct
    unfolding partition-batch-cost-bound-def by simp
  have lower-cost: partition-batch-cost-bound 0 h ?lower
    unfolding partition-batch-cost-bound-def by simp
  have source-cost: partition-batch-cost-bound 0 h ?sources
    unfolding partition-batch-cost-bound-def by simp
  have run:
    charged-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'
      (?charged-child # charged-tail) (U-child # child-outputs-tail)
      (bound-tree P (Fin a) ∪ ⋃(set (?charged-child # charged-tail)))
      (0 + 0 + 0 + 0 + c-child + c-tail) (c-child # child-costs-tail)
    by (rule Charged-Direct-Insert-State-Step
      [where Bmax = Bmax and S-pull = S-pull and D-pull = D-pull
        and charged-child = ?charged-child and direct-edge-batch = ?direct
        and lower-edge-batch = ?lower and source-batch = ?sources
        and batch = ?batch and D-next = batch-min-update D-pull ?batch
        and c-pull = 0 and c-direct = 0 and c-lower = 0 and c-sources = 0
        and c-child = c-child and c-tail = c-tail])
      (use pull beta-bound child-pre S-pull-def S-pull-reaches a-le-b
        a-bound complete-prefix below-threshold child preserved tail
        pull-cost direct-cost lower-cost source-cost in simp-all)
  then show ?thesis
    by blast
qed

lemma charged-direct-insert-partition-loop-state-cases-exists-with-complete:
  fixes W :: bound ⇒ 'a set
  assumes cases:
      (keys-of 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'.
        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) ∧
        complete-on d' (bound-tree P (Fin a)) ∧
        card (bound-tree P (Fin a)) < k * cap ∧
        complete-on d' (W B') ∧
        (∃ d-child b U-child c-child betas bs charged-tail
            child-outputs-tail U-tail c-tail child-costs-tail.
          a ≤ b ∧
```

```
      bound-le (Fin a) B' ∧
      charged-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 ∧
      complete-preserved d-child d' U-child ∧
      charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
        d P B d'
        (batch-min-update D-pull
          (edge-relaxation-pairs-in-bound d-child
            (range-tree P a (Fin b)) beta B @
           edge-relaxation-pairs-between d-child
            (range-tree P a (Fin b)) b beta @
           label-pairs-between d S-pull b beta))
        b betas bs B' charged-tail child-outputs-tail U-tail c-tail
        child-costs-tail))
  shows ∃ betas bs B' charged-Us child-outputs U c child-costs.
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
      d P B d' D a betas bs B' charged-Us child-outputs U c child-costs ∧
    complete-on d' (W B')
proof −
  consider
    (Done-Case) keys-of D = {}
      bound-le (Fin a) B
      complete-on d' (bound-tree P B)
      complete-on d' (W B)
  | (Stop-Case) 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))
  | (Step-Case) S-pull beta D-pull B' where
      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
      complete-on d' (bound-tree P (Fin a))
      card (bound-tree P (Fin a)) < k * cap
      complete-on d' (W B')
      ∃ d-child b U-child c-child betas bs charged-tail child-outputs-tail
          U-tail c-tail child-costs-tail.
        a ≤ b ∧
        bound-le (Fin a) B' ∧
        charged-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 ∧
        complete-preserved d-child d' U-child ∧
        charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
          d P B d'
          (batch-min-update D-pull
            (edge-relaxation-pairs-in-bound d-child
              (range-tree P a (Fin b)) beta B @
```

```
        edge-relaxation-pairs-between d-child
          (range-tree P a (Fin b)) b beta @
        label-pairs-between d S-pull b beta))
      b betas bs B' charged-tail child-outputs-tail U-tail c-tail
      child-costs-tail
  using cases by blast
then show ?thesis
proof cases
  case Done-Case
  have run:
    charged-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 []
    by (rule Charged-Direct-Insert-State-Done[OF Done-Case(1−3)])
  show ?thesis
    using run Done-Case(4) by blast
next
  case Stop-Case
  have run:
    charged-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 []
    by (rule Charged-Direct-Insert-State-Stop[OF Stop-Case(1−3)])
  show ?thesis
    using run Stop-Case(4) by blast
next
  case (Step-Case S-pull beta D-pull B')
  then obtain d-child b U-child c-child betas bs charged-tail
      child-outputs-tail U-tail c-tail child-costs-tail where
    a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
    and child:
      charged-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
    and preserved: complete-preserved d-child d' U-child
    and tail:
      charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
        d P B d'
        (batch-min-update D-pull
          (edge-relaxation-pairs-in-bound d-child
            (range-tree P a (Fin b)) beta B @
          edge-relaxation-pairs-between d-child
            (range-tree P a (Fin b)) b beta @
          label-pairs-between d S-pull b beta))
        b betas bs B' charged-tail child-outputs-tail U-tail c-tail
        child-costs-tail
    by blast
  have run:
    ∃ c child-costs.
```

```
      charged-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) # charged-tail)
        (U-child # child-outputs-tail)
        (bound-tree P (Fin a) ∪
          ⋃(set (range-tree P a (Fin b) # charged-tail))) c child-costs
    by (rule charged-direct-insert-partition-loop-state-step-exists
      [OF Step-Case(1) Step-Case(2) Step-Case(3) Step-Case(4)
        Step-Case(5) a-le-b a-bound Step-Case(6) Step-Case(7)
        child preserved tail])
  then show ?thesis
    using Step-Case(8) by blast
 qed
qed

lemma charged-direct-insert-costed-bmssp-Suc-exists-if-pivots-empty-same-bound:
 assumes a-bound: bound-le (Fin a) B
   and pivots-empty:
     find-pivots-pivots-capped k cap d S B = {}
 shows ∃ d' U c.
   charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B
     d' B U c
proof −
 let ?d-fp = find-pivots-label-capped k cap d S B
 let ?P = find-pivots-pivots-capped k cap d S B
 let ?D = label-partition-view ?d-fp ?P
 let ?U-loop =
   bound-tree ?P (Fin a) ∪ ⋃(set [range-tree ?P a B])
 have empty: keys-of ?D = {}
   using pivots-empty unfolding label-partition-view-def by simp
 have loop:
   charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
     ?d-fp ?P B (dist s) ?D a [] [] B [range-tree ?P a B] []
     ?U-loop 0 []
   by (rule Charged-Direct-Insert-State-Done)
     (use empty a-bound in ‹simp-all add: complete-on-def›)
 have complete:
   complete-on (dist s)
     {v ∈ bound-tree S B. ?d-fp v = dist s v}
   unfolding complete-on-def by simp
 have insert: partition-initial-insert-cost-bound 0 t ?P
   unfolding partition-initial-insert-cost-bound-def by simp
 let ?U = ?U-loop ∪ {v ∈ bound-tree S B. ?d-fp v = dist s v}
 have run:
   charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B
     (dist s) B ?U
     (fp-iter-capped-scan-cost k cap d S S B + 0 + 0)
   by (rule charged-direct-insert-costed-bmssp-step-with-insert-cost
     [OF refl insert loop complete refl refl])
```

```
  then show ?thesis
    by blast
qed

lemma charged-direct-insert-costed-bmssp-Suc-exists-if-initial-threshold-bound:
  assumes 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.
    charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B
      d' (Fin a) U c
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?D = label-partition-view ?d-fp ?P
  let ?U-loop =
    bound-tree ?P (Fin a) ∪ ⋃(set [range-tree ?P a (Fin a)])
  have loop:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
      ?d-fp ?P B (dist s) ?D a [] [] (Fin a)
      [range-tree ?P a (Fin a)] [] ?U-loop 0 []
    by (rule Charged-Direct-Insert-State-Stop)
      (use a-bound threshold in ‹simp-all add: complete-on-def›)
  have complete:
    complete-on (dist s)
      {v ∈ bound-tree S (Fin a). ?d-fp v = dist s v}
    unfolding complete-on-def by simp
  have insert: partition-initial-insert-cost-bound 0 t ?P
    unfolding partition-initial-insert-cost-bound-def by simp
  let ?U =
    ?U-loop ∪ {v ∈ bound-tree S (Fin a). ?d-fp v = dist s v}
  have run:
    charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B
      (dist s) (Fin a) ?U
      (fp-iter-capped-scan-cost k cap d S S B + 0 + 0)
    by (rule charged-direct-insert-costed-bmssp-step-with-insert-cost
      [OF refl insert loop complete refl refl])
  then show ?thesis
    by blast
qed

theorem finite-initial-label-charged-direct-insert-costed-top-level-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run:
      charged-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 −
```

```
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule charged-direct-insert-costed-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  then have U-V: U = V
    using bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  have complete: complete-on d' U
    using post unfolding bmssp-post-def by auto
  then show ?thesis
    using U-V unfolding complete-on-def sssp-correct-def by auto
qed

theorem finite-initial-label-charged-direct-insert-costed-top-level-reachable-correct:
  assumes run:
      charged-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 −
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using finite-initial-label-source-complete
    by (rule top-bmssp-pre-full-reachable)
  have sound: sound-label finite-initial-label
    by (rule finite-initial-label-sound-reachable)
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using reachable-refl[OF source-in-V] by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule charged-direct-insert-costed-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  then show ?thesis
    by (rule successful-root-bmssp-sssp-correct)
qed

inductive-cases charged-direct-insert-costed-bmssp-zeroE:
  charged-direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c

inductive-cases charged-direct-insert-costed-bmssp-SucE:
  charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
```

lemma *charged-direct-insert-costed-partition-loop-state-mono*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs U c child-costs*
shows *nondecreasing-from a bs*
using *run* **by** (*induction*) *auto*

lemma *charged-direct-insert-costed-partition-loop-state-beta-bounds*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs U c child-costs*
shows *bounds-le B betas*
using *run* **by** (*induction*) *auto*

theorem *charged-direct-insert-costed-partition-loop-state-success-or-threshold*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs 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 *charged-direct-insert-costed-bmssp-success-or-threshold-trivial*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
proof (*induction rule*:
charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d P a b B-out d-out k cap* Δ *t h d-child U-child c-child charged-child direct-edge-batch lower-edge-batch source-batch batch D-next c-pull c-direct c-lower c-sources betas bs charged-tail child-outputs-tail U-tail c-tail child-costs-tail c*)
have *tail*:
B-out $= B \vee k * cap \leq$ *card U-tail*
using *Charged-Direct-Insert-State-Step.IH Charged-Direct-Insert-State-Step.prems* **by** *blast*
then show *?case*
proof
assume *B-out* $= B$
then show *?thesis*
by *blast*
next

assume *threshold-tail*: $k * cap \leq card\ U\text{-}tail$
have *tail-trace*:
concrete-partition-loop-trace P B b bs d-out B-out charged-tail U-tail
by (*rule charged-direct-insert-costed-partition-loop-state-trace*)
(*use Charged-Direct-Insert-State-Step* **in** *blast*)+
have *U-tail-def*:
$U\text{-}tail = bound\text{-}tree\ P\ (Fin\ b) \cup \bigcup (set\ charged\text{-}tail)$
using *tail-trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *charged-child-eq*: $charged\text{-}child = range\text{-}tree\ P\ a\ (Fin\ b)$
using *Charged-Direct-Insert-State-Step* **by** *blast*
have *split-b*:
$bound\text{-}tree\ P\ (Fin\ b) =$
$bound\text{-}tree\ P\ (Fin\ a) \cup range\text{-}tree\ P\ a\ (Fin\ b)$
using *bound-tree-split-at*[*of a Fin b P*] *Charged-Direct-Insert-State-Step*
by *simp*
have *U-eq-tail*:
$bound\text{-}tree\ P\ (Fin\ a) \cup$
$\bigcup (set\ (charged\text{-}child\ \#\ charged\text{-}tail)) =$
$U\text{-}tail$
using *U-tail-def split-b charged-child-eq* **by** *auto*
show *?thesis*
using *threshold-tail U-eq-tail* **by** *simp*
qed
next
case (*Charged-Direct-Insert-Base S x Δ M-of t h k cap d B*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-Step D k cap d S B c-insert t Δ M-of h l d′ a*
betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c)
then show *?case*
by *simp*
qed

theorem *charged-direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-and-edge-ranges*:
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ P\ B\ d'\ D\ a$
betas bs B′ charged-Us child-outputs U c child-costs $\Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$\exists\, child\text{-}sources.$
$length\ child\text{-}sources = length\ bs \wedge$
$list\text{-}all2$
$(\lambda S\text{-}child\ charged\text{-}child.\ \exists\, c\text{-}child\ U\text{-}child\ B\text{-}child\ d\text{-}child\ B\text{-}child'.$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\ S\text{-}child\ B\text{-}child$
$d\text{-}child\ B\text{-}child'\ U\text{-}child\ c\text{-}child \wedge$
$bmssp\text{-}pre\text{-}full\ d\ S\text{-}child\ B\text{-}child \wedge$
$(\forall\, x \in S\text{-}child.\ reachable\ s\ x) \wedge$

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-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 ∗ *sum-list* (*map card child-sources*)
and *charged-direct-insert-costed-bmssp-cost-child-sources-and-edge-ranges-trivial*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d*′ *B*′ *U c* ⟹ *True*
proof (*induction rule*:
charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d*′ *P* Δ *M-of t h k cap l d*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Charged-Direct-Insert-State-Stop a B d*′ *P k cap* Δ *M-of t h l d D*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d*
P a b B′ *d*′ *k cap* Δ *t h d-child U-child c-child charged-child*
direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
c-direct c-lower c-sources betas bs charged-tail child-outputs-tail
U-tail c-tail child-costs-tail c)
obtain *child-sources-tail* **where** *len-tail*:
length child-sources-tail = *length bs*
and *sources-tail*:
list-all2
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child*′.
charged-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 ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources-tail (*range-tree-child-list P b bs*)
and *tail-cost*:
c-tail ≤
sum-list child-costs-tail + *sum-list* (*map card child-sources-tail*) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B b betas bs*)) +
h ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P b betas bs*)) +
h ∗ *sum-list* (*map card child-sources-tail*)
using *Charged-Direct-Insert-State-Step.IH Charged-Direct-Insert-State-Step.prems* **by** *blast*

have *card-pull*: *card S-pull* $\leq$ *M-of l*
using *Charged-Direct-Insert-State-Step*
unfolding *pull-separates-def* **by** *blast*
have *below-pull*: $\forall$ *x*$\in$*S-pull*. *below-bound* (*d x*) (*Fin beta*)
proof
fix *x*
assume *xS*: *x* $\in$ *S-pull*
have *S-pull* = *split-below d P beta*
using *Charged-Direct-Insert-State-Step* **by** *blast*
then show *below-bound* (*d x*) (*Fin beta*)
using *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *pull-cost*: *c-pull* $\leq$ *card S-pull*
using *Charged-Direct-Insert-State-Step*
unfolding *partition-pull-cost-bound-def* **by** *blast*
have *child-run*:
charged-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*
using *Charged-Direct-Insert-State-Step* **by** *blast*
have *child-pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
using *Charged-Direct-Insert-State-Step* **by** *blast*
have *child-reaches*: $\bigwedge$*x*. *x* $\in$ *S-pull* $\Longrightarrow$ *reachable s x*
using *Charged-Direct-Insert-State-Step* **by** *blast*
have *child-post*:
bmssp-post-full d S-pull (*Fin beta*) *d-child* (*Fin b*) *U-child*
by (*rule charged-direct-insert-costed-bmssp-correct*
[*OF child-run Charged-Direct-Insert-State-Step.prems*(*1*) *child-pre child-reaches*])
have *child-complete*: *complete-on d-child U-child*
using *child-post* **unfolding** *bmssp-post-full-def* **by** *blast*
have *pre-beta*: *bmssp-pre-full d P* (*Fin beta*)
by (*rule bmssp-pre-full-bound-mono*
[*OF Charged-Direct-Insert-State-Step.prems*(*2*)])
(*use Charged-Direct-Insert-State-Step.hyps* **in** *simp*)
have *cover-beta*: *complete-tree-cover d P* (*Fin beta*)
using *bmssp-pre-full-complete-tree-cover*[*OF pre-beta*] **.**
have *child-post-split*:
bmssp-post-full d (*split-below d P beta*) (*Fin beta*) *d-child*
(*Fin b*) *U-child*
using *child-post Charged-Direct-Insert-State-Step.hyps* **by** *simp*
have *U-child-tree*: *U-child* = *bound-tree P* (*Fin b*)
using *pull-recursive-post-lifts-bound-tree*[*OF cover-beta child-post-split*] **.**
have *charged-child-eq*: *charged-child* = *range-tree P a* (*Fin b*)
using *Charged-Direct-Insert-State-Step* **by** *blast*
have *charged-complete*: *complete-on d-child charged-child*
by (*rule complete-on-subset*[*OF child-complete*])
(*use U-child-tree charged-child-eq range-tree-subset-bound-tree* **in** *metis*)
have *charged-reaches*: $\bigwedge$*u*. *u* $\in$ *charged-child* $\Longrightarrow$ *reachable s u*
unfolding *charged-child-eq range-tree-def tree-set-def tree-path-def* **by** *blast*

```
  have direct-cost:
    c-direct ≤ t * card (outgoing-edges-range charged-child beta B)
    by (rule partition-batch-cost-bound-edge-relaxation-pairs-in-bound
      [where d = d-child])
       (use Charged-Direct-Insert-State-Step charged-complete charged-reaches in
auto)
  have lower-source-cost:
    c-lower + c-sources ≤
      h * card (outgoing-edges-range charged-child b (Fin beta)) +
      h * card S-pull
    by (rule range-costed-split-batch-cost-le-range-card[
        where d = d and S = S-pull and d-child = d-child and b = b
          and beta = beta])
       (use Charged-Direct-Insert-State-Step charged-complete charged-reaches in
auto)
  let ?child-sources = S-pull # child-sources-tail
  have len: length ?child-sources = length (b # bs)
    using len-tail by simp
  have head-source:
    ∃ c-child' U-child' B-child d-child' B-child'.
      charged-direct-insert-costed-bmssp Δ M-of t h k cap l d S-pull B-child
        d-child' B-child' U-child' c-child' ∧
      bmssp-pre-full d S-pull B-child ∧
      (∀ x∈S-pull. reachable s x) ∧
      card S-pull ≤ M-of l ∧
      (∀ x∈S-pull. below-bound (d x) B-child)
    using Charged-Direct-Insert-State-Step card-pull below-pull by blast
  have sources:
    list-all2
      (λS-child charged-child. ∃ c-child U-child B-child d-child B-child'.
        charged-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 ∧
        (∀ 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 (b # bs))
    using head-source sources-tail Charged-Direct-Insert-State-Step by simp
  have cost:
    c ≤ sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * sum-list
        (map card
          (range-tree-child-direct-edge-range-list P B a (beta # betas) (b # bs))) +
      h * sum-list
        (map card
          (range-tree-child-edge-range-list P a (beta # betas) (b # bs))) +
      h * sum-list (map card ?child-sources)
  proof −
```

```
    have c-eq:
      c = c-pull + c-direct + c-lower + c-sources + c-child + c-tail
      using Charged-Direct-Insert-State-Step by blast
    have c-pull + c-direct + c-lower + c-sources + c-child + c-tail ≤
        card S-pull +
        t * card (outgoing-edges-range charged-child beta B) +
        (h * card (outgoing-edges-range charged-child b (Fin beta)) +
          h * card S-pull) +
        c-child +
        (sum-list child-costs-tail + sum-list (map card child-sources-tail) +
          t * sum-list
            (map card (range-tree-child-direct-edge-range-list P B b betas bs)) +
          h * sum-list
            (map card (range-tree-child-edge-range-list P b betas bs)) +
          h * sum-list (map card child-sources-tail))
      using pull-cost direct-cost lower-source-cost tail-cost by linarith
    also have ... =
        sum-list (c-child # child-costs-tail) +
        sum-list (map card ?child-sources) +
        t * (card (outgoing-edges-range charged-child beta B) +
          sum-list
            (map card (range-tree-child-direct-edge-range-list P B b betas bs))) +
        h * (card (outgoing-edges-range charged-child b (Fin beta)) +
          sum-list
            (map card (range-tree-child-edge-range-list P b betas bs))) +
        h * sum-list (map card ?child-sources)
      by (simp add: algebra-simps)
    also have ... =
        sum-list (c-child # child-costs-tail) +
        sum-list (map card ?child-sources) +
        t * sum-list
          (map card
            (range-tree-child-direct-edge-range-list P B a (beta # betas) (b # bs))) +
        h * sum-list
          (map card
            (range-tree-child-edge-range-list P a (beta # betas) (b # bs))) +
        h * sum-list (map card ?child-sources)
      using charged-child-eq by simp
    finally show ?thesis
      using c-eq by simp
  qed
  show ?case
    by (intro exI[of - ?child-sources] conjI)
      (use len sources cost in simp-all)
next
  case (Charged-Direct-Insert-Base S x Δ M-of t h k cap d B)
  show ?case
    by simp
next
```

case (*Charged-Direct-Insert-Step D k cap d S B c-insert t* Δ *M-of h l*
d′ a betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c)
show *?case*
by *simp*
qed

theorem *charged-direct-insert-costed-partition-loop-state-child-cost-bounds-with-invariants*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs U c child-costs* $\Longrightarrow$
$P \subseteq V \Longrightarrow$
$k * card\ P \leq cap \Longrightarrow$
tree-antichain P $\Longrightarrow$
($\bigwedge$*c-child charged-child U-child S-child B-child d-child B-child′*.
⟦*charged-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 charged-child*) $\Longrightarrow$
list-all2 (λ*c-child charged-child*.
c-child $\leq$ *level-range-cost-bound A R L charged-child*)
child-costs (*range-tree-child-list P a bs*)
and *charged-direct-insert-costed-bmssp-child-cost-bounds-with-invariants-trivial*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
proof (*induction rule*:
charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d P a b B′ d′ k cap* Δ *t h d-child U-child c-child charged-child direct-edge-batch lower-edge-batch source-batch batch D-next c-pull c-direct c-lower c-sources betas bs charged-tail child-outputs-tail U-tail c-tail child-costs-tail c*)
have *tail*:
list-all2 (λ*c-child charged-child*.
c-child $\leq$ *level-range-cost-bound A R L charged-child*)
child-costs-tail (*range-tree-child-list P b bs*)
using *Charged-Direct-Insert-State-Step.IH Charged-Direct-Insert-State-*

```
Step.prems
    by blast
  have card-pull: card S-pull ≤ M-of l
    using Charged-Direct-Insert-State-Step
    unfolding pull-separates-def by blast
  have below-pull: ⋀x. x ∈ S-pull ⟹ below-bound (d x) (Fin beta)
  proof −
    fix x
    assume xS: x ∈ S-pull
    have S-pull = split-below d P beta
      using Charged-Direct-Insert-State-Step by blast
    then show below-bound (d x) (Fin beta)
      using xS unfolding split-below-def by auto
  qed
  have pull-k-cap: k ∗ card S-pull ≤ cap
  proof −
    have S-pull = split-below d P beta
      using Charged-Direct-Insert-State-Step by blast
    then show ?thesis
      using split-below-scaled-card-le
        [OF Charged-Direct-Insert-State-Step.prems(1)
          Charged-Direct-Insert-State-Step.prems(2), of d beta]
      by simp
  qed
  have pull-anti: tree-antichain S-pull
  proof −
    have S-pull = split-below d P beta
      using Charged-Direct-Insert-State-Step by blast
    then show ?thesis
      using split-below-tree-antichain
        [OF Charged-Direct-Insert-State-Step.prems(3), of d beta]
      by simp
  qed
  have child-run:
    charged-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
    using Charged-Direct-Insert-State-Step by blast
  have pre-pull: bmssp-pre-full d S-pull (Fin beta)
    using Charged-Direct-Insert-State-Step by blast
  have reaches-pull: ⋀x. x ∈ S-pull ⟹ reachable s x
    using Charged-Direct-Insert-State-Step by blast
  have head:
    c-child ≤ level-range-cost-bound A R L charged-child
    by (rule Charged-Direct-Insert-State-Step.prems(4)
      [OF child-run pre-pull reaches-pull card-pull below-pull
        pull-k-cap pull-anti])
  show ?case
    using head tail Charged-Direct-Insert-State-Step by simp
next
```

case (*Charged-Direct-Insert-Base S x Δ M-of t h k cap d B*)
then show *?case*
by *simp*
next
case (*Charged-Direct-Insert-Step D k cap d S B c-insert t Δ M-of h l d′ a betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c*)
then show *?case*
by *simp*
qed

theorem *charged-direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-ranges*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs 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 charged-child U-child S-child B-child d-child B-child′*.
⟦*charged-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 charged-child*
and *source-factor*: $Suc\ h \leq A$
and *source-progress*:
$\bigwedge$*child-sources*.
list-all2
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child $\leq$ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*) $\Longrightarrow$
sum-list (*map card child-sources*) $\leq$
card (*range-tree-chain P a bs B′*)
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 −
  have P-subset: P ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have child-cost-bounds:
    list-all2 (λc-child charged-child.
      c-child ≤ level-range-cost-bound A R L charged-child)
      child-costs (range-tree-child-list P a bs)
    by (rule charged-direct-insert-costed-partition-loop-state-child-cost-bounds-with-
invariants
      [OF run P-subset P-k-cap P-anti child-bound])
  obtain child-sources where sources:
      list-all2
        (λS-child charged-child. ∃ c-child U-child B-child d-child B-child′.
          charged-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 ∧
          (∀ 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)
    and cost:
      c ≤ 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 * sum-list
          (map card (range-tree-child-edge-range-list P a betas bs)) +
        h * sum-list (map card child-sources)
   using charged-direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-
and-edge-ranges
      [OF run sound pre P-reaches] by blast
  have source-sum:
    sum-list (map card child-sources) ≤
      card (range-tree-chain P a bs B′)
    by (rule source-progress[OF sources])
  have mono: nondecreasing-from a bs
    by (rule charged-direct-insert-costed-partition-loop-state-mono[OF run])
  have child-sum:
    sum-list child-costs ≤
      A * L * card (range-tree-chain P a bs B′) +
      R * card (outgoing-edges (range-tree-chain P a bs B′))
    by (rule child-costs-le-level-range-child-list-bound
      [OF mono child-cost-bounds])
  have trace: concrete-partition-loop-trace P B a bs d′ B′ charged-Us U
    by (rule charged-direct-insert-costed-partition-loop-state-trace
      [OF run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d′ U) charged-Us
```

```
    (range-tree-chain-list P a bs B')
  using trace unfolding concrete-partition-loop-trace-def by blast
have U-def: U = bound-tree P (Fin a) ∪ ⋃(set charged-Us)
  using trace unfolding concrete-partition-loop-trace-def by blast
have Union-eq:
  ⋃(set charged-Us) = ⋃(set (range-tree-chain-list P a bs B'))
  using children by (induction rule: list-all2-induct) auto
have U-eq-chain:
  U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
  using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
have finite-U: finite U
  using U-eq-chain by simp
have chain-subset: range-tree-chain P a bs B' ⊆ U
  using U-eq-chain by blast
have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
  by (rule card-mono[OF finite-U chain-subset])
have outgoing-subset:
  outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
  by (rule outgoing-edges-mono[OF chain-subset])
have finite-out-U: finite (outgoing-edges U)
  by simp
have card-out-le:
  card (outgoing-edges (range-tree-chain P a bs B')) ≤
    card (outgoing-edges U)
  by (rule card-mono[OF finite-out-U outgoing-subset])
have c ≤
    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-direct-edge-range-list P B a betas bs)) +
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs)) +
    sum-list (map card child-sources) +
    h * sum-list (map card child-sources)
  using cost child-sum by linarith
also have ... ≤
    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-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 −
  have h-part:
    h * sum-list (map card child-sources) ≤
      h * card (range-tree-chain P a bs B')
    using source-sum by simp
  have sum-list (map card child-sources) +
```

```
      h * sum-list (map card child-sources) ≤
    card (range-tree-chain P a bs B') +
      h * card (range-tree-chain P a bs B')
    using source-sum h-part by linarith
  then show ?thesis
    by (simp add: algebra-simps)
qed
also have ... ≤
    A * 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)) +
    A * card U
proof −
  have vterm:
    A * L * card (range-tree-chain P a bs B') ≤ A * L * card U
    using card-chain-le by simp
  have eterm:
    R * card (outgoing-edges (range-tree-chain P a bs B')) ≤
     R * card (outgoing-edges U)
    using card-out-le by simp
  have source-term:
    Suc h * card (range-tree-chain P a bs B') ≤ A * card U
  proof −
    have Suc h * card (range-tree-chain P a bs B') ≤
        A * card (range-tree-chain P a bs B')
      by (rule mult-right-mono[OF source-factor]) simp
    also have ... ≤ A * card U
      using card-chain-le by simp
    finally show ?thesis .
  qed
  show ?thesis
    using vterm eterm source-term by linarith
qed
also have ... =
    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))
  by (simp add: algebra-simps)
finally show ?thesis .
qed

end

end
theory BMSSP-Strict-Tie-Breaking-Extensions
```

imports *BMSSP-Correctness.BMSSP-Strict-Tie-Breaking*
BMSSP-Direct-Insert-Extensions
begin

context *strict-tie-breaking-digraph*
begin

theorem *charged-direct-insert-costed-step-seen-success*:
assumes *loop*:
charged-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′*
charged-Us child-outputs 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 *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *success*: $B' = B$
shows *card* (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
proof −
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF sound S-reaches*])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF sound pre S-reaches*] **.**
have *P-subset-S*: *?P* $\subseteq$ *S*
unfolding *find-pivots-pivots-capped-def* **by** (*auto split*: *if-splits*)
have *P-reaches*: $\bigwedge x.\ x \in$ *?P* $\Longrightarrow$ *reachable s x*
using *P-subset-S S-reaches* **by** *blast*
have *trace*:
concrete-partition-loop-trace ?P B a bs d′ B′ charged-Us U-loop
by (*rule charged-direct-insert-costed-partition-loop-state-trace*
[*OF loop sound-fp pivot-pre P-reaches*])
have *loop-post*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF trace*])
have *assembly*:
U-loop $\cup$
{$v \in$ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* =
dist s v} =
bound-tree S B′
by (*rule concrete-capped-find-pivots-final-tree-assembly*
[*OF sound pre S-reaches loop-post*])

```
  have U-tree: U = bound-tree S B
    using U-def assembly success by simp
  have seen-subset:
    find-pivots-seen-capped k cap d S B ⊆ U
    using find-pivots-seen-capped-subset-bound-tree
      [OF sound pre S-reaches below] U-tree by simp
  have finite-U: finite U
    unfolding U-tree bound-tree-def using finite-V by auto
  show ?thesis
    by (rule card-mono[OF finite-U seen-subset])
qed

theorem charged-direct-insert-costed-capped-step-threshold-if-not-success:
  assumes loop:
      charged-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'
        charged-Us child-outputs 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 not-success: B' ≠ B
  shows k * cap ≤ card U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have loop-class:
    B' = B ∨ k * cap ≤ card U-loop
    by (rule charged-direct-insert-costed-partition-loop-state-success-or-threshold
      [OF loop sound-fp pivot-pre P-reaches])
  have threshold-loop: k * cap ≤ card U-loop
    using loop-class not-success by blast
  have trace:
    concrete-partition-loop-trace ?P B a bs d' B' charged-Us U-loop
    by (rule charged-direct-insert-costed-partition-loop-state-trace
```

```
    [OF loop sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have finite-U-loop: finite U-loop
    using loop-post unfolding bmssp-post-full-def by simp
  have finite-U: finite U
    unfolding U-def using finite-U-loop by simp
  have U-loop-subset: U-loop ⊆ U
    unfolding U-def by blast
  have card U-loop ≤ card U
    by (rule card-mono[OF finite-U U-loop-subset])
  then show ?thesis
    using threshold-loop by linarith
qed

theorem charged-direct-insert-costed-capped-step-scan-insert-budget-from-scaled-
seen-or-threshold:
  assumes degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    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 insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
    and loop:
      charged-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'
        charged-Us child-outputs 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 S-k-cap: k * card S ≤ cap
    and anti: tree-antichain S
    and k-pos: 0 < k
    and seen-success:
      B' = B ⟹
       card (find-pivots-seen-capped k cap d S B) ≤ card U
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    2 * A * card U
proof (cases B' = B)
  case True
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
```

```
  show ?thesis
    by (rule find-pivots-scan-and-initial-insert-budget-from-seen-scaled
      [OF S-subset degree anti S-k-cap insert insert-scaled-factor
        seen-scaled-factor seen-success[OF True]])
next
  case False
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have S-cap: card S ≤ cap
  proof −
    have card S ≤ k ∗ card S
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-k-cap by linarith
  qed
  have progress: k ∗ cap ≤ card U
    by (rule charged-direct-insert-costed-capped-step-threshold-if-not-success
      [OF loop U-def sound pre S-reaches False])
  show ?thesis
    by (rule find-pivots-scan-and-initial-insert-budget-from-progress
      [OF S-subset degree S-cap insert degree-factor insert-factor progress])
qed

theorem charged-direct-insert-costed-partition-loop-state-closes-level-bound-from-
child-bound-with-invariants-and-edge-budget:
  assumes run:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      betas bs B′ charged-Us child-outputs 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 P-k-cap: k ∗ card P ≤ cap
    and P-anti: tree-antichain P
    and child-bound:
      ⋀c-child charged-child U-child S-child B-child d-child B-child′.
        ⟦charged-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;
          ⋀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 charged-child
    and source-factor: Suc h ≤ A
    and source-progress:
      ⋀child-sources.
        list-all2
          (λS-child charged-child. ∃ c-child U-child B-child d-child B-child′.
```

charged-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 ∧
(∀ *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*) ⟹
sum-list (*map card child-sources*) ≤
card (*range-tree-chain P a bs B′*)
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*))
≤ *H* ∗ *card* (*outgoing-edges U*)
shows *c* ≤ *A* ∗ *Suc L* ∗ *card U* + (*R* + *H*) ∗ *card* (*outgoing-edges U*)
proof −
have *loop*:
c ≤
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*))
by (*rule charged-direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-ranges*
[*OF run sound pre P-reaches P-k-cap P-anti child-bound source-factor source-progress*])
have *c* ≤ *A* ∗ *Suc L* ∗ *card U* + *R* ∗ *card* (*outgoing-edges U*) +
H ∗ *card* (*outgoing-edges U*)
using *loop edge-budget* **by** *linarith*
also have . . . = *A* ∗ *Suc L* ∗ *card U* + (*R* + *H*) ∗ *card* (*outgoing-edges U*)
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

theorem *charged-direct-insert-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *loop*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs U-loop c-loop 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 *P-k-cap*: *k* ∗ *card P* ≤ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:
⋀*c-child charged-child U-child S-child B-child d-child B-child′*.
⟦*charged-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;
     ⋀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 charged-child
 and source-factor: Suc h ≤ A
 and source-progress:
  ⋀child-sources.
   list-all2
    (λS-child charged-child. ∃ c-child U-child B-child d-child B-child'.
      charged-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 ∧
      (∀ 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) ⟹
    sum-list (map card child-sources) ≤
     card (range-tree-chain P a bs B')
 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))
  ≤ H * card (outgoing-edges U-loop)
 and U-def: U = U-loop ∪ W
 and finite-U: finite U
 and scan-insert: c-scan-insert ≤ A * card U
 and c-def: c = c-scan-insert + c-loop
shows c ≤ A * Suc (Suc L) * card U +
  (R + H) * card (outgoing-edges U)
proof −
 have loop-bound:
  c-loop ≤ A * Suc L * card U-loop +
   (R + H) * card (outgoing-edges U-loop)
  by (rule charged-direct-insert-costed-partition-loop-state-closes-level-bound-from-
child-bound-with-invariants-and-edge-budget
    [OF loop sound pre P-reaches P-k-cap P-anti child-bound source-factor
     source-progress edge-budget])
 have U-loop-subset: U-loop ⊆ U
  using U-def by blast
 have card-loop-le: card U-loop ≤ card U
  by (rule card-mono[OF finite-U U-loop-subset])
 have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
  by (rule outgoing-edges-mono[OF U-loop-subset])
 have finite-out-U: finite (outgoing-edges U)
```

```
    by simp
  have card-out-le:
    card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have c ≤
      A * card U +
      (A * Suc L * card U + (R + H) * card (outgoing-edges U))
  proof −
    have loop-to-U:
      c-loop ≤ A * Suc L * card U +
        (R + H) * card (outgoing-edges U)
    proof −
      have A * Suc L * card U-loop ≤ A * Suc L * card U
        using card-loop-le by simp
      moreover have (R + H) * card (outgoing-edges U-loop) ≤
          (R + H) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc L) * card U + (R + H) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

end

end
theory BMSSP-Top-Level-Bounds-Extensions
  imports BMSSP-Correctness.BMSSP-Top-Level-Bounds
    BMSSP-Strict-Tie-Breaking-Extensions
begin

context strict-tie-breaking-digraph
begin

theorem charged-direct-insert-algorithm-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run:
      charged-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′
proof
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
```

using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity* d' *Infinity U*
by (*rule charged-direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity* d' *Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
show $U = V$
using *post bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
show *sssp-correct* d'
by (*rule finite-initial-label-charged-direct-insert-costed-top-level-correct*
[*OF all-reachable run*])
qed

theorem *charged-direct-insert-algorithm-reachable-correct*:
assumes *run*:
charged-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'
by (*rule finite-initial-label-charged-direct-insert-costed-top-level-reachable-correct*[*OF run*])

definition *charged-direct-insert-top-level-run* ::
nat $\Rightarrow$ *nat* $\Rightarrow$ ($'a \Rightarrow$ *real*) $\Rightarrow$ $'a$ *set* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
charged-direct-insert-top-level-run D N d' *U c* $\longleftrightarrow$
(*let p* = *sssp-log-one-third-param N*;
q = *sssp-log-two-thirds-param N*
in charged-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*)

definition *charged-direct-insert-top-level-cost* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
charged-direct-insert-top-level-cost D N c $\longleftrightarrow$
($\exists\, d'\ U.$ *charged-direct-insert-top-level-run D N* d' *U c*)

definition *charged-direct-insert-top-level-time* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
charged-direct-insert-top-level-time D N =
(*LEAST c. charged-direct-insert-top-level-cost D N c*)

definition *charged-direct-insert-closed-refined-bound-log-params-fixed-degree* ::
nat $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
charged-direct-insert-closed-refined-bound-log-params-fixed-degree D N $\longleftrightarrow$
(*let p* = *sssp-log-one-third-param N*;

```
      q = sssp-log-two-thirds-param N
    in ∀ d' U c.
      charged-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 ⟶
      c ≤ bmssp-refined-graph-time-bound
        (λ-. Suc D * p) (λ-. q) (λ-. p)
        (λ-. p) (λ-. q) (λ-. edge-count) vertex-count)

theorem charged-direct-insert-top-level-run-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run: charged-direct-insert-top-level-run D N d' U c
  shows U = V ∧ sssp-correct d'
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  have run':
    charged-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
    using run unfolding charged-direct-insert-top-level-run-def
    by (simp add: Let-def)
  show ?thesis
    by (rule charged-direct-insert-algorithm-correct[OF all-reachable run'])
qed

lemma charged-direct-insert-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' a betas bs charged-Us child-outputs U-loop c-loop
        child-costs-loop c-insert.
      charged-direct-insert-costed-partition-loop-state D
        (bmssp-level-cap p q) q p p cap l d-fp P Infinity d'
        (label-partition-view d-fp P) a betas bs Infinity charged-Us
        child-outputs U-loop c-loop child-costs-loop ∧
      complete-on d'
        {v ∈ bound-tree {s} Infinity. d-fp v = dist s v} ∧
```

```
      partition-initial-insert-cost-bound c-insert q P
  shows ∃ c. charged-direct-insert-top-level-cost D N c
proof −
  have p-eq: p = Suc l
    using sssp-log-one-third-param-pos[of N]
    unfolding p-def l-def by simp
  obtain d' a betas bs charged-Us child-outputs U-loop c-loop
      child-costs-loop c-insert where
    loop-run:
      charged-direct-insert-costed-partition-loop-state D
        (bmssp-level-cap p q) q p p cap l d-fp P Infinity d'
        (label-partition-view d-fp P) a betas bs Infinity charged-Us
        child-outputs U-loop c-loop child-costs-loop
    and complete:
      complete-on d' {v ∈ bound-tree {s} Infinity. d-fp v = dist s v}
    and insert: partition-initial-insert-cost-bound c-insert q P
    using loop by blast
  let ?U =
    U-loop ∪ {v ∈ bound-tree {s} Infinity. d-fp v = dist s v}
  let ?c =
    fp-iter-capped-scan-cost p cap finite-initial-label {s} {s} Infinity +
      c-insert + c-loop
  have root-run:
    charged-direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p cap
      p finite-initial-label {s} Infinity d' Infinity ?U ?c
  proof −
    have step-run:
      charged-direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p cap
        (Suc l) finite-initial-label {s} Infinity d' Infinity ?U ?c
      by (rule charged-direct-insert-costed-bmssp-step-with-insert-cost
        [where D = label-partition-view d-fp P])
        (use loop-run complete insert in ‹simp-all add: d-fp-def P-def›)
    then show ?thesis
      using p-eq by simp
  qed
  have charged-direct-insert-top-level-run D N d' ?U ?c
    unfolding charged-direct-insert-top-level-run-def
    using root-run unfolding p-def q-def cap-def by (simp add: Let-def)
  then have charged-direct-insert-top-level-cost D N ?c
    unfolding charged-direct-insert-top-level-cost-def by blast
  then show ?thesis
    by blast
qed

lemma charged-direct-insert-top-level-time-witness:
  assumes charged-direct-insert-top-level-cost D N c
  shows charged-direct-insert-top-level-cost D N
    (charged-direct-insert-top-level-time D N)
  unfolding charged-direct-insert-top-level-time-def
```

by (*rule LeastI-ex*) (*use assms* **in** *blast*)

lemma *charged-direct-insert-top-level-time-le*:
assumes *charged-direct-insert-top-level-cost D N c*
shows *charged-direct-insert-top-level-time D N* $\leq$ *c*
unfolding *charged-direct-insert-top-level-time-def*
by (*rule Least-le*) (*rule assms*)

theorem *charged-direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree-from-closed-bound*:
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 *closed-bound*:
charged-direct-insert-closed-refined-bound-log-params-fixed-degree D N
and *run*:
charged-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 −
have *UV-correct*: *U* = *V* $\wedge$ *sssp-correct d*′
by (*rule charged-direct-insert-algorithm-correct*[*OF all-reachable run*])
have *cost*:
c $\leq$ *bmssp-refined-graph-time-bound*
(*λ-. Suc D* $*$ *p*) (*λ-. q*) (*λ-. p*)
(*λ-. p*) (*λ-. q*) (*λ-. edge-count*) *vertex-count*
using *closed-bound run*
unfolding *charged-direct-insert-closed-refined-bound-log-params-fixed-degree-def p-def q-def*
by (*simp add*: *Let-def*)
show *?thesis*
using *UV-correct cost* **by** *blast*
qed

theorem *charged-direct-insert-top-level-run-correct-and-refined-bound-log-params-fixed-degree-from-closed-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *closed-bound*:
charged-direct-insert-closed-refined-bound-log-params-fixed-degree D N
and *run*: *charged-direct-insert-top-level-run D N d*′ *U c*
shows *U* = *V* $\wedge$ *sssp-correct d*′ $\wedge$
c $\leq$ *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-. edge-count) vertex-count
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  have run':
    charged-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
    using run unfolding charged-direct-insert-top-level-run-def
    by (simp add: Let-def)
  show ?thesis
    by (rule charged-direct-insert-algorithm-correct-and-refined-graph-bound-log-
params-fixed-degree-from-closed-bound
      [OF all-reachable closed-bound run'])
qed

theorem charged-direct-insert-top-level-cost-refined-bound-log-params-fixed-degree-
from-closed-bound:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and closed-bound:
      charged-direct-insert-closed-refined-bound-log-params-fixed-degree D N
    and cost: charged-direct-insert-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 −
  obtain d' U where run: charged-direct-insert-top-level-run D N d' U c
    using cost unfolding charged-direct-insert-top-level-cost-def by blast
  show ?thesis
    using charged-direct-insert-top-level-run-correct-and-refined-bound-log-params-
fixed-degree-from-closed-bound
      [OF all-reachable closed-bound run]
    by blast
qed

theorem charged-direct-insert-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 closed-bound:
      charged-direct-insert-closed-refined-bound-log-params-fixed-degree D N
    and cost: charged-direct-insert-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*
by (*rule charged-direct-insert-top-level-cost-refined-bound-log-params-fixed-degree-from-closed-bound*
[*OF all-reachable closed-bound cost*])

theorem *charged-direct-insert-top-level-time-refined-bound-log-params-fixed-degree-from-closed-bound*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *closed-bound*:
charged-direct-insert-closed-refined-bound-log-params-fixed-degree D N
and *run-exists*: *charged-direct-insert-top-level-cost D N c*
shows *charged-direct-insert-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 −
have *cost*:
charged-direct-insert-top-level-cost D N
(*charged-direct-insert-top-level-time D N*)
by (*rule charged-direct-insert-top-level-time-witness*[*OF run-exists*])
show *?thesis*
by (*rule charged-direct-insert-top-level-cost-refined-bound-log-params-fixed-degree-from-closed-bound*
[*OF all-reachable closed-bound cost*])
qed

lemma *charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes*:
fixes *D N p q cap l* :: *nat*
and *d-fp* :: *'a* ⇒ *real*
and *P* :: *'a set*
and *Bmax* :: *real*
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 P-source: P = {s}
    and source-below: d-fp s < Bmax
    and Bmax-nonneg: 0 ≤ Bmax
    and child-pre: bmssp-pre-full d-fp {s} (Fin Bmax)
    and below-threshold: card (bound-tree P (Fin 0)) < p * cap
    and child:
      charged-direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
        cap l d-fp {s} (Fin Bmax) (dist s) (Fin Bmax) U-child c-child
    and direct-empty:
      edge-relaxation-pairs-in-bound (dist s) (range-tree P 0 (Fin Bmax))
        Bmax Infinity = []
  shows ∃ c. charged-direct-insert-top-level-cost D N c
proof −
  let ?D = label-partition-view d-fp P
  let ?D-pull = label-partition-view d-fp {}
  let ?tail-U =
    bound-tree P (Fin Bmax) ∪ ⋃(set [range-tree P Bmax Infinity])
  let ?W = λB. {v ∈ bound-tree {s} B. d-fp v = dist s v}
  have p-pos: 0 < p
    using sssp-log-one-third-param-pos[of N] unfolding p-def .
  have p-eq: p = Suc l
    using sssp-log-one-third-param-pos[of N]
    unfolding p-def l-def by simp
  have M-pos: 0 < bmssp-level-cap p q l
    using p-pos unfolding bmssp-level-cap-def by simp
  have pull:
    pull-separates ?D (bmssp-level-cap p q l) Bmax {s} Bmax ?D-pull
    unfolding pull-separates-def
    using P-source M-pos by simp
  have split: {s} = split-below d-fp P Bmax
    using P-source source-below unfolding split-below-def by auto
  have source-reaches: ∀ x∈{s}. reachable s x
    using reachable-refl[OF source-in-V] by simp
  have preserved: complete-preserved (dist s) (dist s) U-child
    unfolding complete-preserved-def by simp
  have complete-prefix: complete-on (dist s) (bound-tree P (Fin 0))
    unfolding complete-on-def by simp
  have tail:
    charged-direct-insert-costed-partition-loop-state D
      (bmssp-level-cap p q) q p p cap l d-fp P Infinity (dist s)
      (batch-min-update ?D-pull
        (edge-relaxation-pairs-in-bound (dist s)
          (range-tree P 0 (Fin Bmax)) Bmax Infinity @
        edge-relaxation-pairs-between (dist s)
          (range-tree P 0 (Fin Bmax)) Bmax Bmax @
        label-pairs-between d-fp {s} Bmax Bmax))
      Bmax [] [] Infinity [range-tree P Bmax Infinity] []
      ?tail-U 0 []
```

```
proof −
  have lower-empty:
    edge-relaxation-pairs-between (dist s)
      (range-tree P 0 (Fin Bmax)) Bmax Bmax = []
    unfolding edge-relaxation-pairs-between-def
    by (auto simp: filter-empty-conv split: prod.splits)
  have source-empty: label-pairs-between d-fp {s} Bmax Bmax = []
    unfolding label-pairs-between-def
    by (simp add: filter-empty-conv)
  have D-next:
    batch-min-update ?D-pull
      (edge-relaxation-pairs-in-bound (dist s)
        (range-tree P 0 (Fin Bmax)) Bmax Infinity @
       edge-relaxation-pairs-between (dist s)
        (range-tree P 0 (Fin Bmax)) Bmax Bmax @
       label-pairs-between d-fp {s} Bmax Bmax) = ?D-pull
    using direct-empty lower-empty source-empty
    unfolding batch-min-update-def by simp
  have empty: keys-of ?D-pull = {}
    by simp
  have complete: complete-on (dist s) (bound-tree P Infinity)
    unfolding complete-on-def by simp
  have run:
    charged-direct-insert-costed-partition-loop-state D
      (bmssp-level-cap p q) q p p cap l d-fp P Infinity (dist s)
      ?D-pull Bmax [] [] Infinity [range-tree P Bmax Infinity] []
      ?tail-U 0 []
    by (rule Charged-Direct-Insert-State-Done)
      (use empty complete in simp-all)
  then show ?thesis
    using D-next by simp
qed
let ?charged-child = range-tree P 0 (Fin Bmax)
let ?direct =
  edge-relaxation-pairs-in-bound (dist s) ?charged-child Bmax Infinity
let ?lower =
  edge-relaxation-pairs-between (dist s) ?charged-child Bmax Bmax
let ?sources = label-pairs-between d-fp {s} Bmax Bmax
let ?batch = ?direct @ ?lower @ ?sources
have step-exists:
  ∃ c-loop child-costs-loop.
    charged-direct-insert-costed-partition-loop-state D
      (bmssp-level-cap p q) q p p cap l d-fp P Infinity (dist s) ?D
      0 [Bmax] [Bmax] Infinity [?charged-child, range-tree P Bmax Infinity]
      [U-child]
      (bound-tree P (Fin 0) ∪
        ⋃(set [?charged-child, range-tree P Bmax Infinity]))
      c-loop child-costs-loop
  by (rule charged-direct-insert-partition-loop-state-step-exists
```

```
      [where Δ = D and M-of = bmssp-level-cap p q and t = q and h = p
        and k = p and cap = cap and l = l and d = d-fp and P = P
        and B = Infinity and d' = dist s and D = ?D and a = 0
        and Bmax = Bmax and S-pull = {s} and beta = Bmax
        and D-pull = ?D-pull and d-child = dist s and b = Bmax
        and U-child = U-child and c-child = c-child and betas = []
        and bs = [] and B' = Infinity
        and charged-tail = [range-tree P Bmax Infinity]
        and child-outputs-tail = [] and U-tail = ?tail-U and c-tail = 0
        and child-costs-tail = []])
       (use pull child-pre split source-reaches complete-prefix Bmax-nonneg below-
threshold
        child preserved tail in simp-all)
  then obtain c-loop child-costs-loop where loop:
    charged-direct-insert-costed-partition-loop-state D
      (bmssp-level-cap p q) q p p cap l d-fp P Infinity (dist s) ?D
      0 [Bmax] [Bmax] Infinity [?charged-child, range-tree P Bmax Infinity]
      [U-child]
      (bound-tree P (Fin 0) ∪
        ⋃(set [?charged-child, range-tree P Bmax Infinity]))
      c-loop child-costs-loop
    by blast
  have completeW: complete-on (dist s) (?W Infinity)
    unfolding complete-on-def by simp
  have insert: partition-initial-insert-cost-bound 0 q P
    unfolding partition-initial-insert-cost-bound-def by simp
  let ?U = (bound-tree P (Fin 0) ∪
      ⋃(set [?charged-child, range-tree P Bmax Infinity])) ∪ ?W Infinity
  let ?c = fp-iter-capped-scan-cost p cap finite-initial-label {s} {s}
      Infinity + 0 + c-loop
  have root-suc:
    charged-direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p cap
      (Suc l) finite-initial-label {s} Infinity (dist s) Infinity ?U ?c
  proof (rule charged-direct-insert-costed-bmssp-step-with-insert-cost
      [where D = ?D and c-insert = 0])
    show ?D =
        label-partition-view
          (find-pivots-label-capped p cap finite-initial-label {s} Infinity)
          (find-pivots-pivots-capped p cap finite-initial-label {s} Infinity)
      unfolding d-fp-def P-def by simp
    show partition-initial-insert-cost-bound 0 q
        (find-pivots-pivots-capped p cap finite-initial-label {s} Infinity)
      using insert unfolding P-def .
    show charged-direct-insert-costed-partition-loop-state D
        (bmssp-level-cap p q) q p p cap l
        (find-pivots-label-capped p cap finite-initial-label {s} Infinity)
        (find-pivots-pivots-capped p cap finite-initial-label {s} Infinity)
        Infinity (dist s) ?D 0 [Bmax] [Bmax] Infinity
        [?charged-child, range-tree P Bmax Infinity] [U-child]
```

```
      (bound-tree P (Fin 0) ∪
        ⋃(set [?charged-child, range-tree P Bmax Infinity]))
      c-loop child-costs-loop
    using loop unfolding d-fp-def P-def .
  show complete-on (dist s)
      {v ∈ bound-tree {s} Infinity.
        find-pivots-label-capped p cap finite-initial-label {s} Infinity v =
        dist s v}
    using completeW unfolding d-fp-def .
  show ?U =
      (bound-tree P (Fin 0) ∪
        ⋃(set [?charged-child, range-tree P Bmax Infinity])) ∪
      {v ∈ bound-tree {s} Infinity.
        find-pivots-label-capped p cap finite-initial-label {s} Infinity v =
        dist s v}
    unfolding d-fp-def by simp
  show ?c = fp-iter-capped-scan-cost p cap finite-initial-label {s} {s}
      Infinity + 0 + c-loop
    by simp
qed
then have root:
  charged-direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p cap
    p finite-initial-label {s} Infinity (dist s) Infinity ?U ?c
  using p-eq by simp
have charged-direct-insert-top-level-run D N (dist s) ?U ?c
  unfolding charged-direct-insert-top-level-run-def
  using root unfolding p-def q-def cap-def by (simp add: Let-def)
then show ?thesis
  unfolding charged-direct-insert-top-level-cost-def by blast
qed

definition charged-direct-insert-source-pivot-finishes :: nat ⇒ nat ⇒ bool where
  charged-direct-insert-source-pivot-finishes D N ⟷
    (let p = sssp-log-one-third-param N;
         q = sssp-log-two-thirds-param N;
         cap = bmssp-level-cap p q p;
         l = p − 1;
         d-fp = find-pivots-label-capped p cap finite-initial-label {s}
           Infinity;
         P = find-pivots-pivots-capped p cap finite-initial-label {s}
           Infinity
     in ∃ Bmax U-child c-child.
       P = {s} ∧
       d-fp s < Bmax ∧
       0 ≤ Bmax ∧
       bmssp-pre-full d-fp {s} (Fin Bmax) ∧
       card (bound-tree P (Fin 0)) < p ∗ cap ∧
       charged-direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
         cap l d-fp {s} (Fin Bmax) (dist s) (Fin Bmax)
```

```
    U-child c-child ∧
   edge-relaxation-pairs-in-bound (dist s) (range-tree P 0 (Fin Bmax))
    Bmax Infinity = [])

lemma charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes-pred:
  assumes charged-direct-insert-source-pivot-finishes D N
  shows ∃ c. charged-direct-insert-top-level-cost D N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  let ?l = ?p − 1
  let ?d-fp =
    find-pivots-label-capped ?p ?cap finite-initial-label {s} Infinity
  let ?P =
    find-pivots-pivots-capped ?p ?cap finite-initial-label {s} Infinity
  obtain Bmax U-child c-child where
    P-source: ?P = {s}
    and source-below: ?d-fp s < Bmax
    and Bmax-nonneg: 0 ≤ Bmax
    and child-pre: bmssp-pre-full ?d-fp {s} (Fin Bmax)
    and below-threshold: card (bound-tree ?P (Fin 0)) < ?p * ?cap
    and child:
      charged-direct-insert-costed-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p
        ?cap ?l ?d-fp {s} (Fin Bmax) (dist s) (Fin Bmax)
        U-child c-child
    and direct-empty:
      edge-relaxation-pairs-in-bound (dist s) (range-tree ?P 0 (Fin Bmax))
        Bmax Infinity = []
    using assms unfolding charged-direct-insert-source-pivot-finishes-def
    by (auto simp: Let-def)
  show ?thesis
  proof (rule charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes)
    show ?P = {s}
      by (rule P-source)
    show ?d-fp s < Bmax
      by (rule source-below)
    show 0 ≤ Bmax
      by (rule Bmax-nonneg)
    show bmssp-pre-full ?d-fp {s} (Fin Bmax)
      by (rule child-pre)
    show card (bound-tree ?P (Fin 0)) < ?p * ?cap
      by (rule below-threshold)
    show charged-direct-insert-costed-bmssp D (bmssp-level-cap ?p ?q) ?q
        ?p ?p ?cap ?l ?d-fp {s} (Fin Bmax) (dist s) (Fin Bmax)
        U-child c-child
      by (rule child)
    show edge-relaxation-pairs-in-bound (dist s) (range-tree ?P 0 (Fin Bmax))
        Bmax Infinity = []
```

```
    by (rule direct-empty)
  qed
qed

lemma eventually-charged-direct-insert-top-level-cost-if-source-pivot-finishes:
  assumes
    eventually (λN. charged-direct-insert-source-pivot-finishes D N) at-top
  shows eventually (λN. ∃ c. charged-direct-insert-top-level-cost D N c)
    at-top
  using assms
proof eventually-elim
  case (elim N)
  show ?case
    by (rule charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes-pred
      [OF elim])
qed

lemma charged-direct-insert-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. charged-direct-insert-top-level-cost D N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  let ?l = ?p − 1
  let ?d-fp = find-pivots-label-capped ?p ?cap finite-initial-label {s} Infinity
  let ?P = find-pivots-pivots-capped ?p ?cap finite-initial-label {s} Infinity
  let ?D = label-partition-view ?d-fp ?P
  let ?U-loop = bound-tree ?P (Fin 0) ∪ ⋃(set [range-tree ?P 0 Infinity])
  let ?W = {v ∈ bound-tree {s} Infinity. ?d-fp v = dist s v}
  let ?U = ?U-loop ∪ ?W
  let ?c = fp-iter-capped-scan-cost ?p ?cap finite-initial-label {s} {s} Infinity
  have p-eq: ?p = Suc ?l
    using sssp-log-one-third-param-pos[of N] by simp
  have empty: keys-of ?D = {}
    using pivots-empty unfolding label-partition-view-def by simp
  have loop:
    charged-direct-insert-costed-partition-loop-state D
      (bmssp-level-cap ?p ?q) ?q ?p ?p ?cap ?l ?d-fp ?P Infinity
      (dist s) ?D 0 [] [] Infinity [range-tree ?P 0 Infinity] []
      ?U-loop 0 []
    by (rule Charged-Direct-Insert-State-Done)
      (use empty complete-on-dist in simp-all)
  have complete: complete-on (dist s) ?W
    by (rule complete-on-dist)
```

```
  have insert: partition-initial-insert-cost-bound 0 ?q ?P
    unfolding partition-initial-insert-cost-bound-def by simp
  have root-suc:
    charged-direct-insert-costed-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p
      ?cap (Suc ?l) finite-initial-label {s} Infinity (dist s) Infinity
      ?U ?c
    by (rule charged-direct-insert-costed-bmssp-step-with-insert-cost
      [where D = ?D and c-insert = 0])
      (use loop complete insert in simp-all)
  then have root:
    charged-direct-insert-costed-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p
      ?cap ?p finite-initial-label {s} Infinity (dist s) Infinity ?U ?c
    using p-eq by simp
  have charged-direct-insert-top-level-run D N (dist s) ?U ?c
    unfolding charged-direct-insert-top-level-run-def
    using root by (simp add: Let-def)
  then have charged-direct-insert-top-level-cost D N ?c
    unfolding charged-direct-insert-top-level-cost-def by blast
  then show ?thesis
    by blast
qed

lemma eventually-charged-direct-insert-top-level-cost-from-empty-top-pivots:
  eventually (λN. ∃ c. charged-direct-insert-top-level-cost D N c) at-top
  using eventually-top-level-pivots-empty
    [of finite-initial-label Infinity]
proof eventually-elim
  case (elim N)
  show ?case
    by (rule charged-direct-insert-top-level-cost-exists-if-top-pivots-empty
      [OF elim])
qed

end

end
theory BMSSP-Verified-Runtime
  imports BMSSP-Top-Level-Bounds-Extensions
begin
```

6 Runtime Bound for a Verified-Correct BMSSP Run

The top-level theory *BMSSP-Top-Level-Bounds* proves two facts that, taken separately, leave a gap a reader is right to worry about.

- *exact-concrete-top-level-run-correct-and-refined-bound-log-params-fixed-degree* shows that *one* costed run is simultaneously correct and

cost-bounded: for a single *exact-concrete-top-level-run* producing output label d', completed set U and cost c, we have $U = V$, *sssp-correct* d', and a closed-form bound on c. This is a per-run statement in closed form.

- *bmssp-runtime-bigo-target* lifts the closed-form bound to the asymptotic headline $O(m * (ln\ n)\ powr\ (2/3))$, but it is stated about *T-bmssp* D alone — the *least cost* over valid runs. As a number, *T-bmssp D N* carries no evidence that the run achieving it computes shortest paths.

  The asymptotic headline therefore speaks about a least-cost integer, while the correctness theorem speaks about an individual run; nothing yet says the run whose cost realises the asymptotic bound is the same run that is proved correct. This theory closes that gap. Its central observation is that *T-bmssp D N*, being a least cost, is realised by an *actual* run (by *exact-concrete-top-level-time-witness*), and that this realising run is governed by the per-run correctness theorem. Consequently the asymptotic running-time bound is a statement about runs that are themselves verified to return exact single-source shortest-path distances.

context *strict-tie-breaking-digraph*
begin

First, the least-cost value *T-bmssp D N* is realised by a concrete top-level run, and that run satisfies full single-source shortest-path correctness. The hypothesis is exactly the one already used throughout the top-level development: at least one valid run exists for the given size.

theorem *T-bmssp-realised-by-verified-run*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *run-exists*: *exact-concrete-top-level-cost D N c*
  shows $\exists\, d'\ U.$
      exact-concrete-top-level-run D N d′ U (*T-bmssp D N*) $\wedge$
      $U = V \wedge$ *sssp-correct d′*
proof −
  have *cost-T*:
    exact-concrete-top-level-cost D N (*exact-concrete-top-level-time D N*)
    by (*rule exact-concrete-top-level-time-witness*[*OF run-exists*])
  then obtain *d′ U* **where** *run*:
    exact-concrete-top-level-run D N d′ U (*exact-concrete-top-level-time D N*)
    unfolding *exact-concrete-top-level-cost-def* **by** *blast*
  have *correct*:
    $U = V \wedge$ *sssp-correct d′* $\wedge$
      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-. edge-count) vertex-count
   by (rule exact-concrete-top-level-run-correct-and-refined-bound-log-params-fixed-degree
      [OF all-reachable degree run])
 show ?thesis
   using run correct unfolding T-bmssp-def by blast
qed
```

The realising run exists for all but finitely many sizes: the same eventual-existence fact that powers the asymptotic proof (*eventually-exact-concrete-top-level-cost-from-empty-top-pivots*) guarantees a valid run, hence a verified realising run, eventually in N.

```
theorem eventually-T-bmssp-realised-by-verified-run:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
   and degree: edge-outdegree-le D
 shows eventually
   (λN. ∃ d' U.
      exact-concrete-top-level-run D N d' U (T-bmssp D N) ∧
      U = V ∧ sssp-correct d') at-top
proof −
 have runs:
   eventually (λN. ∃ c. exact-concrete-top-level-cost D N c) at-top
   by (rule eventually-exact-concrete-top-level-cost-from-empty-top-pivots)
 show ?thesis
 proof (rule eventually-mono[OF runs])
   fix N
   assume ∃ c. exact-concrete-top-level-cost D N c
   then obtain c where c: exact-concrete-top-level-cost D N c by blast
   show ∃ d' U.
      exact-concrete-top-level-run D N d' U (T-bmssp D N) ∧
      U = V ∧ sssp-correct d'
    by (rule T-bmssp-realised-by-verified-run[OF all-reachable degree c])
 qed
qed

end
```

The public headline of this theory lives in the same asymptotic locale as *bmssp-runtime-headline-instance.bmssp-runtime-bigo-target*. It states the deterministic BMSSP running-time bound together with the witness that the running time belongs to runs returning exact shortest-path distances: for all but finitely many sizes the checked running time *T-bmssp N* is the cost of a concrete run whose output is *sssp-correct* and whose completed set is the whole vertex set.

```
context bmssp-runtime-headline-instance
```

begin

theorem *verified-bmssp-runtime-bigo-target*:
shows $(\lambda n.\ real\ (T\text{-}bmssp\ n)) \in$
$O(\lambda n.\ real\ (m\ n) * (ln\ (real\ n + 2))\ powr\ (2\ /\ 3)) \wedge$
eventually
$(\lambda N.\ \exists d'\ U.$
bounded.exact-concrete-top-level-run D N d′ U $(T\text{-}bmssp\ N) \wedge$
$U = V \wedge$ *bounded.sssp-correct d′*) *at-top*
proof
show $(\lambda n.\ real\ (T\text{-}bmssp\ n)) \in$
$O(\lambda n.\ real\ (m\ n) * (ln\ (real\ n + 2))\ powr\ (2\ /\ 3))$
by (*rule bmssp-runtime-bigo-target*)
next
have *eventually*
$(\lambda N.\ \exists d'\ U.$
bounded.exact-concrete-top-level-run D N d′ U
(*bounded.T-bmssp D N*) $\wedge$
$U = V \wedge$ *bounded.sssp-correct d′*) *at-top*
by (*rule bounded.eventually-T-bmssp-realised-by-verified-run*
[*OF bounded.all-vertices-reachable bounded.bounded-edge-outdegree*])
then show *eventually*
$(\lambda N.\ \exists d'\ U.$
bounded.exact-concrete-top-level-run D N d′ U $(T\text{-}bmssp\ N) \wedge$
$U = V \wedge$ *bounded.sssp-correct d′*) *at-top*
unfolding *T-bmssp-def* **.**
qed

Read together, the two conjuncts of *verified-bmssp-runtime-bigo-target* say that the $O(\lambda n.\ real\ (m\ n) * ln\ (real\ n + 2)\ powr\ (2\ /\ 3))$ running-time bound is not a statement about an abstract least cost in isolation: the cost it bounds is, for all but finitely many sizes, achieved by an explicit BMSSP run that provably computes the exact single-source shortest-path distance function on every reachable vertex. This is the sense in which the entry delivers a single algorithm that is at once correct and within the deterministic sorting-barrier running time.

end

end
theory *BMSSP-Runtime-Instance-Branching*
imports *BMSSP-Verified-Runtime*
BMSSP-Correctness.BMSSP-Executable-Base-Case
begin

7 A Second, Structurally Different Non-Vacuous Instance

The path instance in *BMSSP-Runtime-Instance* certifies the runtime locale on a single chain. To show that inhabitation does not rely on the degenerate linear shape, this theory discharges the same locale on a structurally different graph: the out-star $1 \leftarrow 0 \rightarrow 2$ with two leaves, in which the source has outdegree two and there are two incomparable shortest paths. The proof structure mirrors the path instance: the verified simple-walk enumerator pins down the unique shortest walk to each vertex.

definition *star-vs* :: *nat list* **where**
star-vs = [*0*, *1*, *2*]

definition *star-es* :: (*nat* × *nat*) *list* **where**
star-es = [(*0*, *1*), (*0*, *2*)]

definition *star-V* :: *nat set* **where**
star-V = *set star-vs*

definition *star-E* :: (*nat* × *nat*) *set* **where**
star-E = *set star-es*

definition *star-weight* :: *nat* ⇒ *nat* ⇒ *real* **where**
star-weight u v = (*if* (*u*, *v*) ∈ *star-E then 1 else 0*)

lemma *star-vs-V*: *set star-vs* = *star-V*
unfolding *star-V-def* **by** (*rule refl*)

lemma *star-es-E*: *set star-es* = *star-E*
unfolding *star-E-def* **by** (*rule refl*)

lemma *star-positive-weight*: (*u*, *v*) ∈ *star-E* ⟹ *0* < *star-weight u v*
unfolding *star-weight-def* **by** *simp*

lemma *star-finite-weighted-digraph*:
finite-weighted-digraph star-V star-E star-weight 0
unfolding *finite-weighted-digraph-def*
star-V-def star-vs-def star-E-def star-es-def star-weight-def
by *auto*

interpretation *sfw*: *finite-weighted-digraph star-V star-E star-weight 0*
by (*rule star-finite-weighted-digraph*)

lemma *star-walk-lists*:
exec-simple-walks-betw star-vs star-es 0 0 = [[*0*]]
exec-simple-walks-betw star-vs star-es 0 (*Suc 0*) = [[*0*, *1*]]
exec-simple-walks-betw star-vs star-es 0 2 = [[*0*, *2*]]
by *eval+*

lemma *star-simple-walk-sets*:
  $\{p.\ sfw.simple\text{-}walk\text{-}betw\ 0\ p\ 0\} = \{[0]\}$
  $\{p.\ sfw.simple\text{-}walk\text{-}betw\ 0\ p\ (Suc\ 0)\} = \{[0,\ 1]\}$
  $\{p.\ sfw.simple\text{-}walk\text{-}betw\ 0\ p\ 2\} = \{[0,\ 2]\}$
  using *sfw.set-exec-simple-walks-betw*[*OF star-vs-V star-es-E*, *symmetric*]
  by (*simp-all add*: *star-walk-lists*)

lemma *star-walk-vertex*:
  assumes *sfw.simple-walk-betw 0 p v*
  shows *v* ∈ *star-V*
proof −
  have *walk-p*: *sfw.walk p* **and** *p-ne*: *p* ≠ [] **and** *last-p*: *last p* = *v*
    using *assms* **unfolding** *sfw.simple-walk-betw-def sfw.walk-betw-def* **by** *auto*
  have *v* ∈ *set p*
    using *p-ne last-p* **by** (*metis last-in-set*)
  moreover have *set p* ⊆ *star-V*
    using *walk-p* **by** (*rule sfw.walk-set-subset*)
  ultimately show *?thesis* **by** *blast*
qed

lemma *star-unique-shortest-walk*:
  assumes *sfw.shortest-walk 0 p v* **and** *sfw.shortest-walk 0 q v*
  shows *p* = *q*
proof −
  have *p*: *sfw.simple-walk-betw 0 p v* **and** *q*: *sfw.simple-walk-betw 0 q v*
    using *assms* **unfolding** *sfw.shortest-walk-def* **by** *blast+*
  have *vV*: *v* ∈ *star-V*
    using *p* **by** (*rule star-walk-vertex*)
  have *pm*: *p* ∈ {*r*. *sfw.simple-walk-betw 0 r v*}
    using *p* **by** *simp*
  have *qm*: *q* ∈ {*r*. *sfw.simple-walk-betw 0 r v*}
    using *q* **by** *simp*
  from *vV* **consider** *v* = *0* | *v* = *Suc 0* | *v* = *2*
    unfolding *star-V-def star-vs-def* **by** *auto*
  then show *p* = *q*
  proof *cases*
    case *1* **with** *pm qm* **show** *?thesis* **by** (*simp add*: *star-simple-walk-sets*)
  next
    case *2* **with** *pm qm* **show** *?thesis* **by** (*simp add*: *star-simple-walk-sets*)
  next
    case *3* **with** *pm qm* **show** *?thesis* **by** (*simp add*: *star-simple-walk-sets*)
  qed
qed

lemma *star-all-reachable*:
  assumes *v* ∈ *star-V*
  shows *sfw.reachable 0 v*
proof −

```
  from assms consider v = 0 | v = Suc 0 | v = 2
    unfolding star-V-def star-vs-def by auto
  then show ?thesis
  proof cases
    case 1
    have [0] ∈ {r. sfw.simple-walk-betw 0 r 0}
      by (simp add: star-simple-walk-sets)
    with 1 show ?thesis unfolding sfw.reachable-def by auto
  next
    case 2
    have [0, 1] ∈ {r. sfw.simple-walk-betw 0 r (Suc 0)}
      by (simp add: star-simple-walk-sets)
    with 2 show ?thesis unfolding sfw.reachable-def by auto
  next
    case 3
    have [0, 2] ∈ {r. sfw.simple-walk-betw 0 r 2}
      by (simp add: star-simple-walk-sets)
    with 3 show ?thesis unfolding sfw.reachable-def by auto
  qed
qed

interpretation susd: unique-shortest-digraph star-V star-E star-weight 0
  by unfold-locales (rule star-unique-shortest-walk)

lemma star-edge-outdegree: susd.edge-outdegree-le 2
  unfolding susd.edge-outdegree-le-def
proof
  fix u :: nat
  assume u ∈ star-V
  then consider u = 0 | u = Suc 0 | u = 2
    unfolding star-V-def star-vs-def by auto
  then show card (susd.outgoing-edges {u}) ≤ 2
  proof cases
    case 1
    then have susd.outgoing-edges {u} = {(0, 1), (0, 2)}
      unfolding susd.outgoing-edges-def by (auto simp: star-E-def star-es-def)
    then show ?thesis by (simp add: card-insert-if)
  next
    case 2
    then have susd.outgoing-edges {u} = {}
      unfolding susd.outgoing-edges-def by (auto simp: star-E-def star-es-def)
    then show ?thesis by simp
  next
    case 3
    then have susd.outgoing-edges {u} = {}
      unfolding susd.outgoing-edges-def by (auto simp: star-E-def star-es-def)
    then show ?thesis by simp
  qed
qed
```

interpretation *sbr*: *bounded-reduced-positive-instance star-V star-E star-weight 0 2*
by *unfold-locales*
(*use star-positive-weight star-all-reachable star-edge-outdegree* **in** *auto*)

As with the path, the out-star yields the closed, assumption-free running-time bound and its verified-correct strengthening. The two instances share no structural feature beyond the locale assumptions, so together they witness that the BMSSP runtime headline holds across genuinely different graph shapes.

lemmas *star-runtime-bigo-target* =
sbr.runtime-headline.bmssp-runtime-bigo-target

lemmas *star-verified-runtime-bigo-target* =
sbr.runtime-headline.verified-bmssp-runtime-bigo-target

end
theory *BMSSP-Path-Family-Extended*
imports *BMSSP-Correctness.BMSSP-Path-Family BMSSP-Verified-Runtime*
begin

8 A Size-Parametric BMSSP Running-Time Bound over a Growing Graph Family

The headline *bmssp-runtime-headline-instance.bmssp-runtime-bigo-target* is stated for a graph fixed by the locale, with the asymptotic variable feeding only the internal logarithmic schedule; the edge-count factor in its target is a constant. This theory removes that limitation by exhibiting a genuine *family* of graphs whose size grows without bound: the unit-weight directed path P_k on vertices $0, \ldots, k$ with edges $i \to i + 1$. For this family $card\ (E_k) = k$ and $card\ (V_k) = k + 1$ both grow with k, so k is a true graph-size variable, and we prove the BMSSP running time is $O(k \cdot (ln\ k)^{2/3})$.

8.1 The path family

definition *path-k-V* :: *nat* ⇒ *nat set* **where**
path-k-V k = {*0*..*k*}

definition *path-k-E* :: *nat* ⇒ (*nat* × *nat*) *set* **where**
path-k-E k = (λ*i*. (*i*, *Suc i*)) ‘ {*0*..<*k*}

definition *path-k-weight* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *real* **where**
path-k-weight k u v = (*if* (*u*, *v*) ∈ *path-k-E k then 1 else 0*)

8.2 Structural and cardinality lemmas

lemma *path-k-V-eq*: *path-k-V k* = {*0*..*k*}

unfolding *path-k-V-def* **..**

lemma *path-k-E-iff*: $(u, v) \in$ *path-k-E k* $\longleftrightarrow$ $v = Suc\ u \wedge u < k$
unfolding *path-k-E-def* **by** *auto*

lemma *path-k-no-incoming-0*: $(u, 0) \notin$ *path-k-E k*
by (*simp add*: *path-k-E-iff*)

lemma *path-k-incoming-unique*:
$(u, j) \in$ *path-k-E k* $\longleftrightarrow$ $(0 < j \wedge j \leq k \wedge u = j - 1)$
by (*auto simp*: *path-k-E-iff*)

lemma *path-k-card-V*: *card* (*path-k-V k*) = *Suc k*
unfolding *path-k-V-eq* **by** *simp*

lemma *path-k-card-E*: *card* (*path-k-E k*) = *k*
proof −
have *inj-on* ($\lambda i.\ (i, Suc\ i)$) $\{0..<k\}$
by (*rule inj-onI*) *auto*
then show *?thesis*
unfolding *path-k-E-def* **by** (*simp add*: *card-image*)
qed

lemma *path-k-finite-V*: *finite* (*path-k-V k*)
unfolding *path-k-V-eq* **by** *simp*

lemma *path-k-finite-E*: *finite* (*path-k-E k*)
unfolding *path-k-E-def* **by** *simp*

8.3 Easy locale obligations, uniform in k

lemma *path-k-finite-weighted-digraph*:
finite-weighted-digraph (*path-k-V k*) (*path-k-E k*) (*path-k-weight k*) *0*
proof
show *finite* (*path-k-V k*)
by (*rule path-k-finite-V*)
show *0* $\in$ *path-k-V k*
unfolding *path-k-V-eq* **by** *simp*
show $\bigwedge u\ v.\ (u, v) \in$ *path-k-E k* $\Longrightarrow$ $u \in$ *path-k-V k* $\wedge$ $v \in$ *path-k-V k*
by (*auto simp*: *path-k-E-iff path-k-V-eq*)
show $\bigwedge u\ v.\ (u, v) \in$ *path-k-E k* $\Longrightarrow$ $0 \leq$ *path-k-weight k u v*
by (*simp add*: *path-k-weight-def*)
qed

lemma *path-k-positive-weight*:
$(u, v) \in$ *path-k-E k* $\Longrightarrow$ $0 <$ *path-k-weight k u v*
by (*simp add*: *path-k-weight-def*)

8.4 The unconditional cost-recurrence bound (FLOOR)

The function *path-size-cost* is exactly the closed cost expression that bounds a BMSSP run on P_k at schedule parameter k, with the graph cardinalities substituted: edge slot k and vertex slot *Suc k*. Its asymptotic bound holds unconditionally — no run existence and no shortest-walk uniqueness are required — and already exhibits a genuine size-parametric $O(k \cdot (ln\ k)^{2/3})$ growth in which both the edge count k and the vertex count *Suc k* grow with the asymptotic variable.

```
definition path-size-cost :: nat ⇒ nat where
  path-size-cost k =
    bmssp-refined-graph-time-bound
      (λ-. Suc 1 ∗ sssp-log-one-third-param k)
      (λ-. sssp-log-two-thirds-param k)
      (λ-. sssp-log-one-third-param k)
      (λ-. sssp-log-one-third-param k)
      (λ-. sssp-log-two-thirds-param k)
      (λ-. k) (Suc k)

theorem path-size-cost-bigo-size:
  (λk. real (path-size-cost k))
    ∈ O(λk. real k ∗ (ln (real k + 2)) powr (2 / 3))
proof −
  have (λk. real (path-size-cost k))
     ∈ O(λk. sssp-time-target (λk. k) k)
  proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-
degree-slack
      [where D = 1 and Cn = 2 and Cm = 1
        and v = λk. Suc k and m′ = λk. k and m = λk. k])
    show 0 < (2 :: real) by simp
    show 0 < (1 :: real) by simp
    show eventually (λk. real (Suc k) ≤ 2 ∗ real k) at-top
      by (rule eventually-at-top-linorderI[of 1]) simp
    show eventually (λk. real k ≤ 1 ∗ real k) at-top
      by simp
    show eventually
       (λk. path-size-cost k ≤
         bmssp-refined-graph-time-bound
           (λ-. Suc 1 ∗ sssp-log-one-third-param k)
           (λ-. sssp-log-two-thirds-param k)
           (λ-. sssp-log-one-third-param k)
           (λ-. sssp-log-one-third-param k)
           (λ-. sssp-log-two-thirds-param k)
           (λ-. k) (Suc k)) at-top
      by (simp add: path-size-cost-def)
  qed
  then show ?thesis
    unfolding sssp-time-target-def sssp-log-factor-def by simp
```

qed

8.5 The forced-degenerate (empty-pivot) regime is provably avoided

The only unconditional source of run existence in the development, *strict-tie-breaking-digraph.eventually-exact-concrete-top-level-cost-from-empty-top-pivots*, fires through *unique-shortest-digraph.find-pivots-pivots-capped-singleton-empty-if-params-exceed-vertex-count*, whose hypothesis requires the schedule parameter to *exceed* the vertex count. On P_k at schedule k this hypothesis is *Suc k* < *sssp-log-one-third-param k*, which fails for all large k because the one-third logarithmic factor grows far slower than k. Hence the empty-pivot no-op cannot be the run whose cost the family bound governs: any such run does genuine top-level work.

lemma *sssp-log-one-third-param-le-linear*:
  eventually (λ*k*. *sssp-log-one-third-param k* ≤ *Suc k*) *at-top*
proof (*rule eventually-at-top-linorderI*[*of 1*])
  fix *k* :: *nat* **assume** *k1*: *1* ≤ *k*
  have *pos*: (*0*::*real*) < *real k* + *2* **by** *simp*
  have *ge1*: *1* ≤ *ln* (*real k* + *2*)
  proof −
    have *exp 1* ≤ (*3*::*real*) **using** *exp-le* **by** *simp*
    also have (*3*::*real*) ≤ *real k* + *2* **using** *k1* **by** *simp*
    finally have *exp 1* ≤ *real k* + *2* **.**
    then show *?thesis* **using** *ln-ge-iff*[*of real k* + *2*] **by** *simp*
  qed
  have *ln-le*: *ln* (*real k* + *2*) ≤ *real k* + *1*
    using *ln-le-minus-one*[*OF pos*] **by** *simp*
  have *sssp-log-factor-one-third k* = (*ln* (*real k* + *2*)) *powr* (*1*/*3*)
    unfolding *sssp-log-factor-one-third-def* **..**
  also have ... ≤ (*ln* (*real k* + *2*)) *powr 1*
    by (*rule powr-mono*[*OF - ge1*]) *simp*
  also have ... = *ln* (*real k* + *2*) **by** *simp*
  also have ... ≤ *real k* + *1* **using** *ln-le* **.**
  finally have *factor-le*: *sssp-log-factor-one-third k* ≤ *real k* + *1* **.**
  have *ceil-nonneg*: *0* ≤ ⌈*sssp-log-factor-one-third k*⌉
  proof −
    have (*0*::*real*) ≤ *sssp-log-factor-one-third k*
      using *sssp-log-factor-one-third-pos*[*of k*] **by** (*rule order-less-imp-le*)
    then have ⌈(*0*::*real*)⌉ ≤ ⌈*sssp-log-factor-one-third k*⌉
      by (*rule ceiling-mono*)
    then show *?thesis* **by** *simp*
  qed
  have *real* (*sssp-log-one-third-param k*) =
      real (*nat* ⌈*sssp-log-factor-one-third k*⌉)
    unfolding *sssp-log-one-third-param-def* **..**
  also have ... = *real-of-int* ⌈*sssp-log-factor-one-third k*⌉
    using *ceil-nonneg* **by** *simp*

also have $\ldots \leq$ *real* $k + 1$
proof $-$
have $\lceil$*sssp-log-factor-one-third* $k\rceil \leq \lceil$*real* $k + 1\rceil$
by (*rule ceiling-mono*[*OF factor-le*])
also have $\ldots =$ *int* $k + 1$ **by** *simp*
finally show *?thesis* **by** *simp*
qed
finally have *real* (*sssp-log-one-third-param* k) $\leq$ *real* (*Suc* k)
by *simp*
then show *sssp-log-one-third-param* $k \leq$ *Suc* k
by *linarith*
qed

theorem *path-family-not-forced-degenerate*:
eventually
($\lambda k.\ \neg$ (*card* (*path-k-V* k) $<$ *sssp-log-one-third-param* k)) *at-top*
proof $-$
have *eventually* ($\lambda k.$ *sssp-log-one-third-param* $k \leq$ *Suc* k) *at-top*
by (*rule sssp-log-one-third-param-le-linear*)
then show *?thesis*
by (*auto elim*: *eventually-mono simp*: *path-k-card-V*)
qed

8.6 Unique shortest walks on the path family

context fixes k :: *nat* **begin**

interpretation *pk*: *finite-weighted-digraph*
path-k-V k path-k-E k path-k-weight k 0
by (*rule path-k-finite-weighted-digraph*)

lemma *path-k-canonical-walk*:
$j \leq k \Longrightarrow$ *pk.simple-walk-betw 0* [*0..<Suc j*] *j*
proof (*induction j*)
case *0*
have (*0*::*nat*) $\in$ *path-k-V k*
by (*simp add*: *path-k-V-eq*)
then have *pk.walk* [*0*] **by** *simp*
then show *?case*
by (*simp add*: *pk.simple-walk-betw-def pk.walk-betw-def*)
next
case (*Suc j*)
from *Suc.prems* **have** *jk*: $j \leq k$ **and** *jlt*: $j < k$ **by** *simp-all*
have *ih*: *pk.simple-walk-betw 0* [*0..<Suc j*] *j*
using *Suc.IH jk* **by** *blast*
have *edge*: (*j*, *Suc j*) $\in$ *path-k-E k*
using *jlt* **by** (*simp add*: *path-k-E-iff*)
have *fresh*: *Suc j* $\notin$ *set* [*0..<Suc j*] **by** *simp*
have *pk.simple-walk-betw 0* ([*0..<Suc j*] @ [*Suc j*]) (*Suc j*)

```
    using pk.simple-walk-snoc[OF ih edge fresh] .
  moreover have [0..<Suc j] @ [Suc j] = [0..<Suc (Suc j)]
    by (simp add: upt-Suc-append)
  ultimately show ?case by simp
qed

lemma path-k-all-reachable:
  v ∈ path-k-V k ⟹ pk.reachable 0 v
proof −
  assume v ∈ path-k-V k
  then have v ≤ k by (simp add: path-k-V-eq)
  then have pk.simple-walk-betw 0 [0..<Suc v] v
    by (rule path-k-canonical-walk)
  then show pk.reachable 0 v
    unfolding pk.reachable-def by blast
qed

lemma path-k-simple-walk-unique:
  pk.simple-walk-betw 0 p j ⟹ j ≤ k ⟹ p = [0..<Suc j]
proof (induction j arbitrary: p rule: less-induct)
  case (less j)
  from less.prems have sw: pk.simple-walk-betw 0 p j and jk: j ≤ k
    by blast+
  from sw have walk-p: pk.walk p and ne: p ≠ [] and hd-p: hd p = 0
      and last-p: last p = j and dist-p: distinct p
    unfolding pk.simple-walk-betw-def pk.walk-betw-def by auto
  show ?case
  proof (cases j)
    case 0
    have len1: length p = 1
    proof (rule ccontr)
      assume length p ≠ 1
      from ne have length p ≥ 1 by (cases p) auto
      with ‹length p ≠ 1› have len2: 2 ≤ length p by linarith
      have idx: Suc (length p − 2) < length p using len2 by linarith
      have e: (p ! (length p − 2), p ! Suc (length p − 2)) ∈ path-k-E k
        using pk.walk-nth-edge[OF walk-p idx] .
      have suceq: Suc (length p − 2) = length p − 1 using len2 by linarith
      have p ! (length p − 1) = last p
        using ne by (simp add: last-conv-nth)
      with last-p 0 have p ! (length p − 1) = 0 by simp
      with e suceq have (p ! (length p − 2), 0) ∈ path-k-E k by simp
      with path-k-no-incoming-0 show False by simp
    qed
    from len1 ne obtain x where p = [x] by (cases p) auto
    with hd-p have p = [0] by simp
    then show ?thesis using 0 by simp
  next
    case (Suc j')
```

```
    have j'k: j' ≤ k using jk Suc by simp
    have j'lt: j' < j using Suc by simp
    have ne1: length p ≠ 1
    proof
      assume length p = 1
      with ne obtain x where p = [x] by (cases p) auto
      with hd-p last-p Suc show False by simp
    qed
    from ne have length p ≥ 1 by (cases p) auto
    with ne1 have len2: 2 ≤ length p by linarith
    have idx: Suc (length p − 2) < length p using len2 by linarith
    have e: (p ! (length p − 2), p ! Suc (length p − 2)) ∈ path-k-E k
      using pk.walk-nth-edge[OF walk-p idx] .
    have suceq: Suc (length p − 2) = length p − 1 using len2 by linarith
    have last-nth: p ! (length p − 1) = Suc j'
    proof −
      have p ! (length p − 1) = last p
        using ne by (simp add: last-conv-nth)
      with last-p Suc show ?thesis by simp
    qed
    from e suceq last-nth
    have edge-last: (p ! (length p − 2), Suc j') ∈ path-k-E k by simp
    have nth-n2: p ! (length p − 2) = j'
      using edge-last unfolding path-k-incoming-unique by simp
    have idx2: length p − 2 < length p using len2 by linarith
    have pref: pk.simple-walk-betw 0 (take (Suc (length p − 2)) p)
       (p ! (length p − 2))
      using pk.simple-walk-prefix[OF sw idx2] .
    have pref2: pk.simple-walk-betw 0 (take (length p − 1) p) j'
      using pref suceq nth-n2 by simp
    have take-eq: take (length p − 1) p = [0..<Suc j']
      using less.IH[OF j'lt pref2 j'k] .
    have pbl: p = butlast p @ [last p]
      using append-butlast-last-id[OF ne] by simp
    have bl: butlast p = take (length p − 1) p
      by (rule butlast-conv-take)
    have p = [0..<Suc j'] @ [Suc j']
      using pbl bl last-p take-eq Suc by simp
    then show ?thesis using Suc by (simp add: upt-Suc-append)
  qed
qed

lemma path-k-unique-shortest-walk:
  pk.shortest-walk 0 p v ⟹ pk.shortest-walk 0 q v ⟹ p = q
proof −
  assume sp: pk.shortest-walk 0 p v and sq: pk.shortest-walk 0 q v
  have swp: pk.simple-walk-betw 0 p v
    using sp unfolding pk.shortest-walk-def by blast
  have swq: pk.simple-walk-betw 0 q v
```

```
    using sq unfolding pk.shortest-walk-def by blast
  have walk-p: pk.walk p and ne-p: p ≠ [] and last-pv: last p = v
    using swp unfolding pk.simple-walk-betw-def pk.walk-betw-def by auto
  have v-in: v ∈ set p
    using last-in-set[OF ne-p] last-pv by simp
  have set p ⊆ path-k-V k
    using pk.walk-set-subset[OF walk-p] .
  with v-in have v ∈ path-k-V k by blast
  then have vk: v ≤ k by (simp add: path-k-V-eq)
  have p = [0..<Suc v]
    using path-k-simple-walk-unique[OF swp vk] .
  moreover have q = [0..<Suc v]
    using path-k-simple-walk-unique[OF swq vk] .
  ultimately show p = q by simp
qed

lemma path-k-bounded-instance:
  bounded-reduced-positive-instance (path-k-V k) (path-k-E k)
    (path-k-weight k) 0 1
proof −
  interpret usd: unique-shortest-digraph
    path-k-V k path-k-E k path-k-weight k 0
    by unfold-locales (rule path-k-unique-shortest-walk)
  have outdeg: usd.edge-outdegree-le 1
    unfolding usd.edge-outdegree-le-def
  proof
    fix u :: nat
    assume u ∈ path-k-V k
    show card (usd.outgoing-edges {u}) ≤ 1
    proof (cases u < k)
      case True
      then have usd.outgoing-edges {u} = {(u, Suc u)}
        unfolding usd.outgoing-edges-def by (auto simp: path-k-E-iff)
      then show ?thesis by simp
    next
      case False
      then have usd.outgoing-edges {u} = {}
        unfolding usd.outgoing-edges-def by (auto simp: path-k-E-iff)
      then show ?thesis by simp
    qed
  qed
  show ?thesis
  proof
    show ⋀u v. (u, v) ∈ path-k-E k ⟹ 0 < path-k-weight k u v
      by (rule path-k-positive-weight)
    show ⋀v. v ∈ path-k-V k ⟹ pk.reachable 0 v
      by (rule path-k-all-reachable)
    show usd.edge-outdegree-le 1
      by (rule outdeg)
```

```
  qed
qed

interpretation bpi: bounded-reduced-positive-instance
  path-k-V k path-k-E k path-k-weight k 0 1
  by (rule path-k-bounded-instance)

lemma path-k-charged-top-level-cost-exists-if-root-run:
  assumes root:
      ∃ d' U c.
        bpi.charged-direct-insert-costed-bmssp (Suc 0)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k))
          (sssp-log-two-thirds-param k) (sssp-log-one-third-param k)
          (sssp-log-one-third-param k)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
          (sssp-log-one-third-param k)
          pk.finite-initial-label {0} Infinity d' Infinity U c
  shows ∃ c. strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
    (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c
proof −
  let ?p = sssp-log-one-third-param k
  let ?q = sssp-log-two-thirds-param k
  let ?cap = bmssp-level-cap ?p ?q ?p
  show ?thesis
    using root
    unfolding bpi.charged-direct-insert-top-level-cost-def
      bpi.charged-direct-insert-top-level-run-def
    by (auto simp: Let-def)
qed

lemma path-k-charged-root-run-exists-if-initial-loop:
  assumes loop:
      ∃ d' a betas bs charged-Us child-outputs U-loop c-loop
          child-costs-loop c-insert.
        bpi.charged-direct-insert-costed-partition-loop-state
          (Suc 0)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k))
          (sssp-log-two-thirds-param k) (sssp-log-one-third-param k)
          (sssp-log-one-third-param k)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
          (sssp-log-one-third-param k − 1)
          (bpi.find-pivots-label-capped (sssp-log-one-third-param k)
            (bmssp-level-cap (sssp-log-one-third-param k)
              (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
            pk.finite-initial-label {0} Infinity)
```

```
      (bpi.find-pivots-pivots-capped (sssp-log-one-third-param k)
        (bmssp-level-cap (sssp-log-one-third-param k)
          (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
        pk.finite-initial-label {0} Infinity)
      Infinity d'
      (bpi.label-partition-view
        (bpi.find-pivots-label-capped (sssp-log-one-third-param k)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
          pk.finite-initial-label {0} Infinity)
        (bpi.find-pivots-pivots-capped (sssp-log-one-third-param k)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
          pk.finite-initial-label {0} Infinity))
      a betas bs Infinity charged-Us child-outputs U-loop c-loop
      child-costs-loop ∧
    pk.complete-on d'
      {v ∈ pk.bound-tree {0} Infinity.
        bpi.find-pivots-label-capped (sssp-log-one-third-param k)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
          pk.finite-initial-label {0} Infinity v = pk.dist 0 v} ∧
    bpi.partition-initial-insert-cost-bound c-insert
      (sssp-log-two-thirds-param k)
      (bpi.find-pivots-pivots-capped (sssp-log-one-third-param k)
        (bmssp-level-cap (sssp-log-one-third-param k)
          (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
        pk.finite-initial-label {0} Infinity)
  shows ∃ d' U c.
    bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap (sssp-log-one-third-param k)
        (sssp-log-two-thirds-param k))
      (sssp-log-two-thirds-param k) (sssp-log-one-third-param k)
      (sssp-log-one-third-param k)
      (bmssp-level-cap (sssp-log-one-third-param k)
        (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
      (sssp-log-one-third-param k)
      pk.finite-initial-label {0} Infinity d' Infinity U c
proof −
  let ?p = sssp-log-one-third-param k
  let ?q = sssp-log-two-thirds-param k
  let ?cap = bmssp-level-cap ?p ?q ?p
  have root-suc:
    ∃ d' U c.
      bpi.charged-direct-insert-costed-bmssp (Suc 0)
        (bmssp-level-cap ?p ?q) ?q ?p ?p ?cap (Suc (?p − 1))
        pk.finite-initial-label {0} Infinity d' Infinity U c
    by (rule bpi.charged-direct-insert-costed-bmssp-Suc-exists-from-initial-loop-at-
bound-with-insert)
```

```
    (rule loop)
  then show ?thesis
    using sssp-log-one-third-param-pos[of k] by simp
qed

lemma path-k-charged-top-level-cost-exists-if-initial-loop:
  assumes loop:
      ∃ d' a betas bs charged-Us child-outputs U-loop c-loop
          child-costs-loop c-insert.
        bpi.charged-direct-insert-costed-partition-loop-state
          (Suc 0)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k))
          (sssp-log-two-thirds-param k) (sssp-log-one-third-param k)
          (sssp-log-one-third-param k)
          (bmssp-level-cap (sssp-log-one-third-param k)
            (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
          (sssp-log-one-third-param k − 1)
          (bpi.find-pivots-label-capped (sssp-log-one-third-param k)
            (bmssp-level-cap (sssp-log-one-third-param k)
              (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
            pk.finite-initial-label {0} Infinity)
          (bpi.find-pivots-pivots-capped (sssp-log-one-third-param k)
            (bmssp-level-cap (sssp-log-one-third-param k)
              (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
            pk.finite-initial-label {0} Infinity)
          Infinity d'
          (bpi.label-partition-view
            (bpi.find-pivots-label-capped (sssp-log-one-third-param k)
              (bmssp-level-cap (sssp-log-one-third-param k)
                (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
              pk.finite-initial-label {0} Infinity)
            (bpi.find-pivots-pivots-capped (sssp-log-one-third-param k)
              (bmssp-level-cap (sssp-log-one-third-param k)
                (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
              pk.finite-initial-label {0} Infinity))
          a betas bs Infinity charged-Us child-outputs U-loop c-loop
          child-costs-loop ∧
        pk.complete-on d'
          {v ∈ pk.bound-tree {0} Infinity.
            bpi.find-pivots-label-capped (sssp-log-one-third-param k)
              (bmssp-level-cap (sssp-log-one-third-param k)
                (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
              pk.finite-initial-label {0} Infinity v = pk.dist 0 v} ∧
        bpi.partition-initial-insert-cost-bound c-insert
          (sssp-log-two-thirds-param k)
          (bpi.find-pivots-pivots-capped (sssp-log-one-third-param k)
            (bmssp-level-cap (sssp-log-one-third-param k)
              (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
```

```
      pk.finite-initial-label {0} Infinity)
  shows ∃ c. strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
    (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c
proof −
  have root:
    ∃ d′ U c.
      bpi.charged-direct-insert-costed-bmssp (Suc 0)
        (bmssp-level-cap (sssp-log-one-third-param k)
          (sssp-log-two-thirds-param k))
        (sssp-log-two-thirds-param k) (sssp-log-one-third-param k)
        (sssp-log-one-third-param k)
        (bmssp-level-cap (sssp-log-one-third-param k)
          (sssp-log-two-thirds-param k) (sssp-log-one-third-param k))
        (sssp-log-one-third-param k)
        pk.finite-initial-label {0} Infinity d′ Infinity U c
    by (rule path-k-charged-root-run-exists-if-initial-loop[OF loop])
  show ?thesis
    by (rule path-k-charged-top-level-cost-exists-if-root-run[OF root])
qed

end
```

8.7 The exact-concrete algorithm-level running-time bound

Unlike the unconditional floor *path-size-cost-bigo-size*, which bounds the closed cost expression with the graph cardinalities substituted, this theorem bounds the least exact-concrete top-level cost *path-T-bmssp*. Its hypothesis is intentionally explicit: for all but finitely many k, at least one exact-concrete top-level run must exist on P_k. The separate totality-obstruction theory shows that this hypothesis is not dischargeable for the exact-concrete relation on this family; the theorem is therefore retained as a conditional comparison result, not as a non-vacuous algorithm-level claim.

```
definition path-T-bmssp :: nat ⇒ nat where
  path-T-bmssp k = strict-tie-breaking-digraph.T-bmssp
    (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k

theorem eventually-path-family-charged-direct-insert-top-level-cost-if-source-pivot-
finishes:
  assumes source-pivot-finishes:
    eventually
      (λk. strict-tie-breaking-digraph.charged-direct-insert-source-pivot-finishes
        (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k) at-top
  shows eventually
    (λk. ∃ c. strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
      (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c) at-top
  using source-pivot-finishes
proof eventually-elim
  case (elim k)
```

```
  interpret bpi: bounded-reduced-positive-instance
    path-k-V k path-k-E k path-k-weight k 0 1
    by (rule path-k-bounded-instance)
  have bpi.charged-direct-insert-source-pivot-finishes 1 k
    using elim .
  then show ?case
     by (rule bpi.charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes-
pred)
qed

theorem bmssp-path-family-runtime-bigo-size:
  assumes runs-exist:
    eventually (λk. ∃ c. strict-tie-breaking-digraph.exact-concrete-top-level-cost
        (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c) at-top
  shows (λk. real (path-T-bmssp k))
     ∈ O(λk. real k * (ln (real k + 2)) powr (2 / 3))
proof −
  have (λk. real (path-T-bmssp k))
     ∈ O(λk. sssp-time-target (λk. k) k)
  proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-
degree-slack
      [where D = 1 and Cn = 2 and Cm = 1
        and v = λk. Suc k and m' = λk. k and m = λk. k])
    show 0 < (2 :: real) by simp
    show 0 < (1 :: real) by simp
    show eventually (λk. real (Suc k) ≤ 2 * real k) at-top
      by (rule eventually-at-top-linorderI[of 1]) simp
    show eventually (λk. real k ≤ 1 * real k) at-top
      by simp
    show eventually
        (λk. path-T-bmssp k ≤
          bmssp-refined-graph-time-bound
            (λ-. Suc 1 * sssp-log-one-third-param k)
            (λ-. sssp-log-two-thirds-param k)
            (λ-. sssp-log-one-third-param k)
            (λ-. sssp-log-one-third-param k)
            (λ-. sssp-log-two-thirds-param k)
            (λ-. k) (Suc k)) at-top
      using runs-exist
    proof eventually-elim
      case (elim k)
      then obtain c where cost:
        strict-tie-breaking-digraph.exact-concrete-top-level-cost
          (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c
        by blast
      interpret bpi: bounded-reduced-positive-instance
        path-k-V k path-k-E k path-k-weight k 0 1
        by (rule path-k-bounded-instance)
      have cost': bpi.exact-concrete-top-level-cost 1 k c
```

```
    using cost .
  have key: bpi.T-bmssp 1 k ≤
      bmssp-refined-graph-time-bound
        (λ-. Suc 1 * sssp-log-one-third-param k)
        (λ-. sssp-log-two-thirds-param k)
        (λ-. sssp-log-one-third-param k)
        (λ-. sssp-log-one-third-param k)
        (λ-. sssp-log-two-thirds-param k)
        (λ-. bpi.edge-count) bpi.vertex-count
    by (rule bpi.T-bmssp-refined-bound-log-params-fixed-degree
      [OF bpi.all-vertices-reachable bpi.bounded-edge-outdegree cost'])
  have ec: bpi.edge-count = k
    by (simp add: bpi.edge-count-def path-k-card-E)
  have vc: bpi.vertex-count = Suc k
    by (simp add: bpi.vertex-count-def path-k-card-V)
  have tb: bpi.T-bmssp 1 k = path-T-bmssp k
    by (simp add: path-T-bmssp-def)
  show ?case
    using key[unfolded ec vc] tb by simp
 qed
qed
then show ?thesis
 unfolding sssp-time-target-def sssp-log-factor-def by simp
qed
```

8.8 A charged path-family theorem waiting on charged cost bounds

The charged direct-insert top-level relation is the intended successor for the path-family integration. On this branch we can already state and check the exact conditional bridge needed by downstream JAR-facing documentation: if charged top-level runs eventually exist on P_k and every such charged cost is bounded by the same refined graph-time expression, then the least charged cost has the size-parametric target. The second assumption is the charged analogue of the refined-bound theorem still expected from the runtime proof branch.

```
definition path-charged-direct-insert-time :: nat ⇒ nat where
 path-charged-direct-insert-time k =
   strict-tie-breaking-digraph.charged-direct-insert-top-level-time
     (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k

theorem bmssp-path-family-charged-direct-insert-runtime-bigo-size-if-cost-bounded:
 assumes charged-runs-exist:
   eventually
    (λk. ∃ c.
      strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
        (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c) at-top
  and charged-cost-bound:
```

```
    eventually
      (λk. ∀ c.
        strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
          (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c ⟶
        c ≤ bmssp-refined-graph-time-bound
          (λ-. Suc 1 * sssp-log-one-third-param k)
          (λ-. sssp-log-two-thirds-param k)
          (λ-. sssp-log-one-third-param k)
          (λ-. sssp-log-one-third-param k)
          (λ-. sssp-log-two-thirds-param k)
          (λ-. k) (Suc k)) at-top
  shows (λk. real (path-charged-direct-insert-time k))
    ∈ O(λk. real k * (ln (real k + 2)) powr (2 / 3))
proof −
  have (λk. real (path-charged-direct-insert-time k))
    ∈ O(λk. sssp-time-target (λk. k) k)
  proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-
degree-slack
      [where D = 1 and Cn = 2 and Cm = 1
        and v = λk. Suc k and m' = λk. k and m = λk. k])
    show 0 < (2 :: real) by simp
    show 0 < (1 :: real) by simp
    show eventually (λk. real (Suc k) ≤ 2 * real k) at-top
      by (rule eventually-at-top-linorderI[of 1]) simp
    show eventually (λk. real k ≤ 1 * real k) at-top
      by simp
    show eventually
        (λk. path-charged-direct-insert-time k ≤
          bmssp-refined-graph-time-bound
            (λ-. Suc 1 * sssp-log-one-third-param k)
            (λ-. sssp-log-two-thirds-param k)
            (λ-. sssp-log-one-third-param k)
            (λ-. sssp-log-one-third-param k)
            (λ-. sssp-log-two-thirds-param k)
            (λ-. k) (Suc k)) at-top
      using charged-runs-exist charged-cost-bound
    proof eventually-elim
      case (elim k)
      then obtain c where cost:
        strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
          (path-k-V k) (path-k-E k) (path-k-weight k) 0 1 k c
        by blast
      interpret bpi: bounded-reduced-positive-instance
        path-k-V k path-k-E k path-k-weight k 0 1
        by (rule path-k-bounded-instance)
      have cost': bpi.charged-direct-insert-top-level-cost 1 k c
        using cost .
      have time-le-c: bpi.charged-direct-insert-top-level-time 1 k ≤ c
        unfolding bpi.charged-direct-insert-top-level-time-def
```

```
      by (rule Least-le) (rule cost')
    have c-le:
      c ≤ bmssp-refined-graph-time-bound
        (λ-. Suc 1 * sssp-log-one-third-param k)
        (λ-. sssp-log-two-thirds-param k)
        (λ-. sssp-log-one-third-param k)
        (λ-. sssp-log-one-third-param k)
        (λ-. sssp-log-two-thirds-param k)
        (λ-. k) (Suc k)
      using elim(2) cost by blast
    have tb:
      bpi.charged-direct-insert-top-level-time 1 k =
        path-charged-direct-insert-time k
      by (simp add: path-charged-direct-insert-time-def)
    show ?case
      using time-le-c c-le tb by linarith
  qed
 qed
 then show ?thesis
  unfolding sssp-time-target-def sssp-log-factor-def by simp
qed

end
theory BMSSP-Cost-Totality
 imports BMSSP-Path-Family-Extended
begin
```

9 On the Totality of the Exact-Concrete Cost Relation

This theory records a precise, machine-checked structural fact about the exact-concrete cost relation *unique-shortest-digraph.exact-concrete-bmssp* and its companion loop relation *unique-shortest-digraph.exact-concrete-partition-loop-state*.

The intended goal was to discharge the run-existence hypothesis *runs-exist* of *bmssp-path-family-runtime-bigo-size* unconditionally, by constructing a terminating partition-loop witness for the path family. The construction reduces, via *unique-shortest-digraph.exact-concrete-bmssp-Suc-exists-from-finite-initial-loop-cases*, to building a partition-loop state whose loop iterates over the successive half-open distance ranges of the pivot tree.

The lemmas below show why that loop cannot iterate more than once under a sound label. The single inductive rule *unique-shortest-digraph.exact-concrete-partition-loop-state* uses for a genuine step (Exact-Concrete-State-Step) carries the hypothesis that the recursive child output equals the incremental range tree *range-tree P a* (*Fin b*). But child correctness, available whenever the carried label is sound, pins that same child output to the

cumulative bounded tree *bound-tree P* (*Fin b*). Since *range-tree P a* (*Fin b*) is by definition the set difference *bound-tree P* (*Fin b*) − *bound-tree P* (*Fin a*), the two can agree only when the lower prefix *bound-tree P* (*Fin a*) is empty. For a reachable singleton source rooted at s this prefix is empty exactly when the lower bound is zero, because the source itself has distance zero and therefore lies in every prefix with a strictly positive bound.

The consequence is that a sound exact-concrete partition loop admits a genuine step only at lower bound zero; any second genuine step, which would necessarily occur at a strictly positive lower bound, is impossible. A complete top-level run, whose returned bound is *Infinity*, requires the loop to terminate in the Done case with an emptied partition; with non-empty pivots that demands the single permitted step to leave no residual frontier, i.e. its one child must already solve the entire bounded tree. These observations isolate the obstruction to the unconditional totality result precisely in the shape of the State-Step rule.

Consequence for the path-family proof route. On the unit-weight directed path P_k, a multi-step exact-concrete construction would require a second genuine loop step at a positive lower bound. The two rigidity lemmas below rule out precisely that step. We therefore retain *bmssp-path-family-runtime-bigo-size* only as a conditional theorem and do not use it as evidence for a non-vacuous path-family runtime claim.

The unconditional floor *path-size-cost-bigo-size*, which bounds the closed cost expression with the graph cardinalities substituted, is unaffected. The non-vacuous theorem in `BMSSP_NonVacuous_Family.thy` instead uses the charged relation and a decoupled schedule. Relaxing the exact-concrete State-Step rule to mirror the more permissive operational loop would be a separate change to the imported cost relation and is not undertaken here.

context *unique-shortest-digraph*
begin

Step rigidity. In a sound exact-concrete loop step the recursive child output is the cumulative bounded tree, so equating it with the incremental range tree forces the lower prefix tree to be empty.

lemma *exact-concrete-step-forces-empty-prefix*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
    and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
    and *pull-split*: *S-pull* = *split-below d P beta*
    and *beta-bound*: *bound-le* (*Fin beta*) (*Fin Bmax*)
    and *child-pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
    and *a-le-b*: $a \leq b$
    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-eq*: *U-child* = *range-tree P a* (*Fin b*)

```
  shows bound-tree P (Fin a) = {}
proof −
  have S-pull-reaches: ⋀x. x ∈ S-pull ⟹ reachable s x
    using pull-split P-reaches unfolding split-below-def by blast
  have child-post:
    bmssp-post-full d S-pull (Fin beta) d-child (Fin b) U-child
    by (rule exact-concrete-bmssp-correct[OF child sound child-pre S-pull-reaches])
  have pre-beta: bmssp-pre-full d P (Fin beta)
    using bmssp-pre-full-bound-mono[OF pre beta-bound] .
  have cover-beta: complete-tree-cover d P (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pre-beta] .
  have child-post-split:
    bmssp-post-full d (split-below d P beta) (Fin beta) d-child (Fin b) U-child
    using child-post pull-split by simp
  have U-child-tree: U-child = bound-tree P (Fin b)
    using pull-recursive-post-lifts-bound-tree[OF cover-beta child-post-split] .
  have eq: range-tree P a (Fin b) = bound-tree P (Fin b)
    using U-child-eq U-child-tree by simp
  have sub: bound-tree P (Fin a) ⊆ bound-tree P (Fin b)
    using a-le-b by (intro bound-tree-bound-mono) simp
  show ?thesis
    using eq sub unfolding range-tree-eq-bound-diff by blast
qed

end
```

Two locale-free interface facts about *pull-separates*. The first is cap monotonicity: weakening the pull capacity preserves a pull. The second is the existence of a singleton pull at any capacity $M \geq 1$, given a key whose value strictly dominates every other key's value. The residual queue in the existential is annotated *'k partition-view* so that the record extension slot is fixed to *unit*; without the annotation the concrete record witness fails to unify with the fresh extension type variable that the bare existential introduces.

```
lemma pull-separates-mono-cap:
  assumes pull-separates D M B S x D' and M ≤ M'
  shows pull-separates D M' B S x D'
  using assms unfolding pull-separates-def by (meson le-trans)

lemma pull-separates-singleton-exists-abstract:
  assumes fin: finite (keys-of D)
    and u: u ∈ keys-of D
    and umin: ⋀v. v ∈ keys-of D ⟹ v ≠ u ⟹ value-of D u < value-of D v
    and M1: 1 ≤ M
    and upper: value-of D u < B
  shows ∃beta (D'::'k partition-view).
        pull-separates (D::'k partition-view) M B {u} beta D'
proof −
  have le-rem: ⋀v. v ∈ keys-of D − {u} ⟹ value-of D u ≤ value-of D v
    using umin by (auto intro: less-imp-le)
```

```
  show ?thesis
  proof (cases keys-of D − {u} = {})
    case True
    have pull-separates D M B {u} B
          (| keys-of = keys-of D − {u}, value-of = value-of D |)
      unfolding pull-separates-def using u M1 True le-rem by simp
    thus ?thesis by blast
  next
    case False
    let ?A = value-of D ‘ (keys-of D − {u})
    have finA: finite ?A using fin by simp
    have neA: ?A ≠ {} using False by simp
    have lo: ⋀v. v ∈ keys-of D − {u} ⟹ Min ?A ≤ value-of D v
      using finA by (auto intro: Min-le)
    have hi: value-of D u < Min ?A
      using finA neA umin by (auto simp: Min-gr-iff)
    have pull-separates D M B {u} (Min ?A)
          (| keys-of = keys-of D − {u}, value-of = value-of D |)
      unfolding pull-separates-def using u M1 le-rem lo hi False by simp
    thus ?thesis by blast
  qed
qed

context strict-tie-breaking-digraph
begin
```

Source persistence. The source lies in every prefix tree with a strictly positive bound, because its true distance is zero.

```
lemma source-in-positive-prefix:
  assumes a-pos: 0 < a
  shows s ∈ bound-tree {s} (Fin a)
proof −
  have reach: reachable s s
    by (rule reachable-refl[OF source-in-V])
  have dist0: dist s s = 0
    by (rule dist-refl-zero[OF source-in-V])
  have below: below-bound (dist s s) (Fin a)
    using dist0 a-pos by simp
  show ?thesis
    by (rule source-in-own-bound-tree[OF - reach below]) simp
qed
```

The obstruction, combining step rigidity with source persistence. For the singleton source a genuine exact-concrete loop step is impossible at any strictly positive lower bound. Equivalently every genuine step must occur at lower bound zero, so a sound loop performs at most one genuine step.

```
lemma no-genuine-exact-concrete-step-at-positive-lower-bound:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d {s} (Fin Bmax)
```

```
    and pull-split: S-pull = split-below d {s} beta
    and beta-bound: bound-le (Fin beta) (Fin Bmax)
    and child-pre: bmssp-pre-full d S-pull (Fin beta)
    and a-pos: 0 < a
    and a-le-b: a ≤ b
    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-eq: U-child = range-tree {s} a (Fin b)
  shows False
proof −
  have P-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using reachable-refl[OF source-in-V] by simp
  have empty: bound-tree {s} (Fin a) = {}
    by (rule exact-concrete-step-forces-empty-prefix
      [OF sound pre P-reaches pull-split beta-bound child-pre a-le-b child
        U-child-eq])
  have s ∈ bound-tree {s} (Fin a)
    by (rule source-in-positive-prefix[OF a-pos])
  then show False
    using empty by simp
qed
```

The relaxed (charged) model: isolating the remaining obligation. The rigidity analysis above explains why the *exact-concrete* relation admits no genuine step at a positive lower bound, and hence why its size-parametric running-time bound is vacuous on the path family. The permissive *charged-direct-insert-costed-bmssp* relation removes that rigidity — its State-Step rule (rule *Charged-Direct-Insert-Step*) leaves the child output *U-child* free and only records the incremental range tree *range-tree P a* (*Fin b*) as the cost-accounting slice *charged-child*. Runs therefore exist; the open question is whether the charged per-level *cost* bound goes through.

That bound is gated on a single arithmetic obligation, the charged *source-progress*: for each loop child the pulled source set must be dominated in cardinality by its range-tree slice, *card* (*split-below d P beta*) ≤ *card* (*range-tree P a* (*Fin b*)). Summed along the loop via *sum-card-dominated-by-range-tree-child-list-le-chain* this is exactly what bounds the cumulative pull cost by the chain cardinality, i.e. linearly in the vertex count.

The direct-insert development discharges this for free only because its rigid rule equates the child output with the slice; by the same rigidity that route is vacuous (the lemmas above). In the charged model the child output is the cumulative *bound-tree P* (*Fin b*) (no lower bound), so the slice bound is genuine new information. The lemmas below reduce it to a single named side-condition — *pivot settledness*: the pulled sources avoid the already-settled prefix *bound-tree P* (*Fin a*) below the step's lower bound *a*. Geometrically: a pivot pulled at the band starting at *a* has true distance at least *a*.

This side-condition is *not* derivable from the local State-Step hypotheses

(*complete-on* d' constrains the output label d', not the input label d that selects the pulled sources; a settled pivot below a carrying a small input label could be pulled). Establishing it requires a frontier/monotonicity invariant threaded top-down through the charged inductive relation — the same band-monotonicity that underlies the correctness proof, but re-expressed on the separate cost relation. That is the design step the introduction to this theory records as not undertaken. The reduction below makes the obligation precise, single, and reusable: once pivot settledness is available as a loop invariant, the charged top-level bound becomes unconditional by mirroring the direct-insert chain.

lemma *split-below-card-le-range-tree-if-settled*:
  assumes *within*: *split-below d P beta* $\subseteq$ *bound-tree P* (*Fin b*)
    and *settled*: *split-below d P beta* $\cap$ *bound-tree P* (*Fin a*) = {}
  shows *card* (*split-below d P beta*) $\leq$ *card* (*range-tree P a* (*Fin b*))
proof −
  have *split-below d P beta* $\subseteq$ *bound-tree P* (*Fin b*) − *bound-tree P* (*Fin a*)
    using *within settled* **by** *blast*
  also have ... = *range-tree P a* (*Fin b*)
    by (*simp add*: *range-tree-eq-bound-diff*)
  finally show *?thesis*
    by (*rule card-mono*[*OF finite-range-tree*])
qed

The within-bound inclusion is the standard source-subset fact: pivots whose input label is correct and below the returned bound b lie in the child's bounded tree, hence (by monotonicity of *bound-tree* in the source set) in *bound-tree P* (*Fin b*).

lemma *split-below-within-bound-tree-if-complete-label-below*:
  assumes *complete*: *complete-on d* (*split-below d P beta*)
    and *reaches*: $\bigwedge x.$ $x \in$ *split-below d P beta* $\Longrightarrow$ *reachable s x*
    and *below*: $\bigwedge x.$ $x \in$ *split-below d P beta* $\Longrightarrow$ *d x* $<$ *b*
  shows *split-below d P beta* $\subseteq$ *bound-tree P* (*Fin b*)
proof −
  have *split-below d P beta* $\subseteq$ *bound-tree* (*split-below d P beta*) (*Fin b*)
  proof (*rule sources-subset-bound-tree-if-label-below*[*OF complete reaches*])
    fix *x* **assume** $x \in$ *split-below d P beta*
    then show *below-bound* (*d x*) (*Fin b*) **using** *below* **by** *simp*
  qed
  also have ... $\subseteq$ *bound-tree P* (*Fin b*)
    by (*rule bound-tree-mono*[*OF split-below-subset*])
  finally show *?thesis* **.**
qed

The tight reduction. The charged per-child source-card domination follows from the standard pivot-completeness and label bound (both available in the direct-insert template) together with the single genuinely-new pivot settledness side-condition. This is the charged counterpart of the direct *source-card-le* step and feeds the existing chain summation.

lemma *split-below-card-le-range-tree-if-complete-and-settled*:
assumes *complete*: *complete-on d* (*split-below d P beta*)
and *reaches*: $\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *d x* $< b$
and *settled*: *split-below d P beta* $\cap$ *bound-tree P* (*Fin a*) $= \{\}$
shows *card* (*split-below d P beta*) $\leq$ *card* (*range-tree P a* (*Fin b*))
by (*rule split-below-card-le-range-tree-if-settled*
[*OF split-below-within-bound-tree-if-complete-label-below*
[*OF complete reaches below*] *settled*])

Reducing the settledness side-condition to threaded invariants. The lemmas above take pivot settledness *split-below d P beta* $\cap$ *bound-tree P* (*Fin a*) $= \{\}$ as a hypothesis. The authors' note (above) flags that settledness is *not* derivable from the local State-Step hypotheses alone: it needs a frontier/monotonicity invariant threaded top-down through the recursion. The next three lemmas isolate *exactly* that obligation by reducing settledness to two recognizable, modular invariants of the input label d at a loop step:

- **label-floor** $\forall\, x{\in}P.\ a \leq d\ x$ — the current pivot labels lie at or above the step's lower band a (the structural fact that earlier bands have already consumed the sources below a), and
- **prefix-exactness** $\forall\, v{\in}$*tree-set P*. *local.dist s v* $< a \longrightarrow d\ v =$ *local.dist s v* — d is exact on the settled tree-set prefix below a (the band-monotonicity invariant the descending recursion maintains). Under those two, a pulled pivot with *local.dist s x* $< a$ would have exact label $d\ x =$ *local.dist s x* and hence $d\ x < a$, contradicting the label floor; so the pull is settled. This turns the opaque settledness side-condition into the single precise invariant a top-down threading must establish, and feeds the existing card / chain machinery unchanged.

lemma *split-below-inter-bound-tree-empty-if-label-ge-exact*:
assumes *ge*: $\bigwedge x.\ x \in P \Longrightarrow a \leq d\ x$
and *exact*: $\bigwedge v.\ v \in$ *tree-set P* $\Longrightarrow$ *dist s v* $< a \Longrightarrow d\ v =$ *dist s v*
shows *split-below d P beta* $\cap$ *bound-tree P* (*Fin a*) $= \{\}$
using *ge exact* **by** (*force simp*: *split-below-def bound-tree-eq-tree-set*)

lemma *split-below-card-le-range-tree-if-complete-label-ge-exact*:
assumes *complete*: *complete-on d* (*split-below d P beta*)
and *reaches*: $\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *d x* $< b$
and *ge*: $\bigwedge x.\ x \in P \Longrightarrow a \leq d\ x$
and *exact*: $\bigwedge v.\ v \in$ *tree-set P* $\Longrightarrow$ *dist s v* $< a \Longrightarrow d\ v =$ *dist s v*
shows *card* (*split-below d P beta*) $\leq$ *card* (*range-tree P a* (*Fin b*))
by (*rule split-below-card-le-range-tree-if-complete-and-settled*
[*OF complete reaches below*

split-below-inter-bound-tree-empty-if-label-ge-exact[*OF ge exact*]])

lemma *split-below-subset-range-tree-if-complete-label-ge-exact*:
assumes *complete*: *complete-on d* (*split-below d P beta*)
and *reaches*: $\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *d x* $<$ *b*
and *ge*: $\bigwedge x.\ x \in P \Longrightarrow a \leq d\ x$
and *exact*: $\bigwedge v.\ v \in$ *tree-set P* $\Longrightarrow$ *dist s v* $<$ *a* $\Longrightarrow$ *d v* $=$ *dist s v*
shows *split-below d P beta* $\subseteq$ *range-tree P a* (*Fin b*)
proof −
have *within*: *split-below d P beta* $\subseteq$ *bound-tree P* (*Fin b*)
by (*rule split-below-within-bound-tree-if-complete-label-below*[*OF complete reaches below*])
have *settled*: *split-below d P beta* $\cap$ *bound-tree P* (*Fin a*) $= \{\}$
by (*rule split-below-inter-bound-tree-empty-if-label-ge-exact*[*OF ge exact*])
have *split-below d P beta* $\subseteq$ *bound-tree P* (*Fin b*) − *bound-tree P* (*Fin a*)
using *within settled* **by** *blast*
also have ... $=$ *range-tree P a* (*Fin b*)
by (*simp add*: *range-tree-eq-bound-diff*)
finally show *?thesis* **.**
qed

Strong base case of the charged totality induction. At level *0* the charged relation can fire only through the *Charged-Direct-Insert-Base* rule, which is restricted to singleton source sets. For such a singleton we exhibit the complete base-case witness: a charged run exists, the returned bound does not exceed the input bound, the output set is exactly the bounded tree of the returned bound, and the returned label is complete on that output. This is the leaf in which the level induction for charged-run existence (obligation *O1* of the path-family bridge) bottoms out. The completeness certificate is read straight off *base-case-result-correct*; note we reduce that fact with *base-case-result-def* and *prod.case only* – a full simplification would orient the $U =$ *bound-tree S B′* conjunct into a rewrite rule and corrupt the label witness, which itself mentions *base-case-vertices*.

lemma *charged-bmssp-zero-total*:
assumes $S = \{x\}$
shows $\exists\, d'\ B'\ U\ c.$
charged-direct-insert-costed-bmssp Δ *M-of t h k cap 0 d S B d′ B′ U c* $\wedge$
bound-le B′ B $\wedge$
$U =$ *bound-tree S B′* $\wedge$
complete-on d′ U
proof −
let *?d′* $= \lambda v.$ *if v* $\in$ *base-case-vertices k x B then dist s v else d v*
let *?B′* $=$ *base-case-bound k x B*
let *?U* $=$ *base-case-vertices k x B*
have *run*: *charged-direct-insert-costed-bmssp* Δ *M-of t h k cap 0 d S B ?d′ ?B′ ?U*
(*base-case-scan-cost* Δ *k x B*)
by (*rule Charged-Direct-Insert-Base*[*OF assms*])

```
  have corr: ?U = bound-tree S ?B' ∧ complete-on ?d' ?U
    using base-case-result-correct[OF assms, where k=k and B=B and d=d]
    unfolding base-case-result-def prod.case .
  have le: bound-le ?B' B by (rule base-case-bound-le)
  show ?thesis using run le corr by blast
qed
```

Full coverage at the leaf. When the bounded tree of the input bound has at most k vertices the truncating base case keeps *all* of them, so the level-*0* charged run returns the input bound itself ($B' = B$) and its output set is the entire bounded tree *bound-tree S B*. This is the sharpened leaf used by the coverage half of the level induction: a small enough source band is completely solved in one base call without losing the residual frontier.

```
lemma charged-bmssp-zero-full-cover:
  assumes S: S = {x}
    and small: card (bound-tree S B) ≤ k
  shows ∃ d' U c.
    charged-direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B U c ∧
    U = bound-tree S B ∧ complete-on d' U
proof −
  have order-len: length (base-case-order x B) = card (bound-tree {x} B)
  proof −
    have distinct (base-case-order x B)
      by (rule base-case-order-distinct)
    moreover have set (base-case-order x B) = bound-tree {x} B
      by (rule base-case-order-set)
    ultimately show ?thesis
      by (metis distinct-card)
  qed
  have len: length (base-case-order x B) ≤ k
    using small order-len S by simp
  have U: base-case-vertices k x B = bound-tree {x} B
    by (rule base-case-success[OF len])
  have B': base-case-bound k x B = B
    using len unfolding base-case-bound-def by (simp add: Let-def)
  let ?d' = λv. if v ∈ base-case-vertices k x B then dist s v else d v
  have run:
    charged-direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B
      ?d' (base-case-bound k x B) (base-case-vertices k x B)
      (base-case-scan-cost Δ k x B)
    by (rule Charged-Direct-Insert-Base[OF S])
  have complete: complete-on ?d' (base-case-vertices k x B)
    unfolding complete-on-def by simp
  have run2:
    charged-direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B
      ?d' B (bound-tree S B) (base-case-scan-cost Δ k x B)
    using run by (simp add: B' U S)
  have complete2: complete-on ?d' (bound-tree S B)
    using complete by (simp add: U S)
```

show *?thesis*
using *run2 complete2* **by** *blast*
qed

The output-label collapse $d' = dist\ s$. Fixing the run's output label to the true distance *local.dist s* trivialises every completeness obligation in the charged relation: *complete-on* (*local.dist s*) *?U* gives *complete-on* (*local.dist s*) *U* for all *U*, and the next fact gives *complete-preserved d* (*local.dist s*) *U* for all *d*, *U*. Since *charged-direct-insert-top-level-cost* existentially quantifies the output label, choosing *local.dist s* is legitimate (the empty-pivot witness already does so), and it removes the band-monotonicity exactness invariant from the run-existence problem entirely, leaving a purely structural induction.

lemma *complete-preserved-dist*: *complete-preserved d* (*dist s*) *U*
unfolding *complete-preserved-def* **using** *complete-on-dist* **by** *blast*

The $d' = local.dist\ s$ specialisation of the band-loop driver *charged-direct-insert-partition-loop-state-cases-exists-with-complete*: with the output label pinned to *local.dist s* every *complete-on* and *complete-preserved* side-condition is discharged automatically, so a loop state exists as soon as the current band admits a Done, Stop, or Step case (the Step case still carrying a child run and a recursive tail loop state).

lemma *charged-loop-state-exists-dist*:
fixes *a Bmax* :: *real*
assumes *cases*:
(*keys-of D* = {} ∧ *bound-le* (*Fin a*) *B*) ∨
(*bound-le* (*Fin a*) *B* ∧ *k* ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*))) ∨
(∃ *S-pull beta D-pull B'*.
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*) ∧
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ∧
(∃ *d-child b U-child c-child betas bs charged-tail*
child-outputs-tail U-tail c-tail child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B'* ∧
charged-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* ∧
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
d P B (*dist s*)
(*batch-min-update D-pull*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta B* @
edge-relaxation-pairs-between d-child
(*range-tree P a* (*Fin b*)) *b beta* @
label-pairs-between d S-pull b beta))
b betas bs B' charged-tail child-outputs-tail U-tail c-tail

```
      child-costs-tail))
  shows ∃ betas bs B' charged-Us child-outputs U c child-costs.
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
      d P B (dist s) D a betas bs B' charged-Us child-outputs U c child-costs
proof −
  have ∃ betas bs B' charged-Us child-outputs U c child-costs.
      charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l
        d P B (dist s) D a betas bs B' charged-Us child-outputs U c child-costs ∧
      complete-on (dist s) ((λ-::bound. {}) B')
    by (rule charged-direct-insert-partition-loop-state-cases-exists-with-complete
        [where W = λ-::bound. {} and d' = dist s and Bmax = Bmax])
      (use cases in ‹blast intro: complete-on-dist complete-preserved-dist›)
  thus ?thesis by blast
qed
```

One genuine step, then Done, under $d' = local.dist\ s$. Specialising the combined step/Done loop witness *charged-direct-insert-partition-loop-state-step-done-exists* to the true-distance output label discharges all three completeness side-conditions automatically (via *complete-on* (*local.dist s*) *?U* and *complete-preserved ?d* (*local.dist s*) *?U*) and fixes the four partition-operation costs to zero. What is left is purely structural: a pull of the current band, a recursive child run, the resulting residual partition view being empty, and the band/threshold bookkeeping. The loop returned terminates at the input bound B – its output bound is B itself – so it is exactly the loop shape consumed by the top-level existence reduction when $B = Infinity$.

```
lemma charged-loop-single-step-then-done-exists-dist:
  fixes a b beta Bmax :: real and B :: bound and D D-pull :: 'a partition-view
  assumes pull: pull-separates D (M-of l) Bmax S-pull beta D-pull
    and beta-bound: bound-le (Fin beta) B
    and pre: bmssp-pre-full d S-pull (Fin beta)
    and split: S-pull = split-below d P beta
    and reaches: ∀ x∈S-pull. reachable s x
    and a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B
    and small: card (bound-tree P (Fin a)) < k * cap
    and child:
      charged-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
    and tail-empty:
      keys-of (batch-min-update D-pull
        (edge-relaxation-pairs-in-bound d-child (range-tree P a (Fin b)) beta B @
         edge-relaxation-pairs-between d-child (range-tree P a (Fin b)) b beta @
         label-pairs-between d S-pull b beta)) = {}
    and b-bound: bound-le (Fin b) B
  shows ∃ betas bs charged-Us child-outputs U c child-costs.
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B
      (dist s) D a betas bs B charged-Us child-outputs U c child-costs
proof −
```

```
  let ?cc = range-tree P a (Fin b)
  let ?direct = edge-relaxation-pairs-in-bound d-child ?cc beta B
  let ?lower = edge-relaxation-pairs-between d-child ?cc b beta
  let ?sources = label-pairs-between d S-pull b beta
  let ?batch = ?direct @ ?lower @ ?sources
  let ?D-next = batch-min-update D-pull ?batch
  have tail:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B
      (dist s) ?D-next b [] [] B [range-tree P b B] []
      (bound-tree P (Fin b) ∪ ⋃(set [range-tree P b B])) 0 []
    by (rule Charged-Direct-Insert-State-Done)
       (use tail-empty b-bound in ‹auto intro: complete-on-dist›)
  have run:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B
      (dist s) D a [beta] [b] B (?cc # [range-tree P b B]) [U-child]
      (bound-tree P (Fin a) ∪ ⋃(set (?cc # [range-tree P b B])))
      (0 + 0 + 0 + 0 + c-child + 0) [c-child]
    by (rule Charged-Direct-Insert-State-Step
        [where Bmax = Bmax and S-pull = S-pull and D-pull = D-pull
          and charged-child = ?cc and direct-edge-batch = ?direct
          and lower-edge-batch = ?lower and source-batch = ?sources
          and batch = ?batch and D-next = ?D-next
          and c-pull = 0 and c-direct = 0 and c-lower = 0 and c-sources = 0
          and c-tail = 0])
       (use pull beta-bound pre split reaches a-le-b a-bound small child tail in
         ‹auto intro: complete-on-dist complete-preserved-dist
           simp: partition-pull-cost-bound-def partition-batch-cost-bound-def›)
  then show ?thesis by blast
qed
```

Cost-exposing form of the single-step-then-Done loop. Identical to *charged-loop-single-step-then-done-exists-dist* but returning the concrete loop state with its cost read off as the lone child cost *c-child* (all four partition costs and the empty Done tail are zero). This is the brick the cheap charged top-level run on the path family threads level by level, so that the whole run cost telescopes to a sum of the FindPivots scans plus the single base-case scan, with no edge-range or chain terms.

```
lemma charged-loop-single-step-then-done-costed-dist:
  fixes a b beta Bmax :: real and B :: bound and D D-pull :: 'a partition-view
  assumes pull: pull-separates D (M-of l) Bmax S-pull beta D-pull
    and beta-bound: bound-le (Fin beta) B
    and pre: bmssp-pre-full d S-pull (Fin beta)
    and split: S-pull = split-below d P beta
    and reaches: ∀x∈S-pull. reachable s x
    and a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B
    and small: card (bound-tree P (Fin a)) < k * cap
    and child:
      charged-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
    and tail-empty:
      keys-of (batch-min-update D-pull
        (edge-relaxation-pairs-in-bound d-child (range-tree P a (Fin b)) beta B @
         edge-relaxation-pairs-between d-child (range-tree P a (Fin b)) b beta @
         label-pairs-between d S-pull b beta)) = {}
    and b-bound: bound-le (Fin b) B
  shows charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B
      (dist s) D a [beta] [b] B
      (range-tree P a (Fin b) # [range-tree P b B]) [U-child]
      (bound-tree P (Fin a) ∪ ⋃(set (range-tree P a (Fin b) # [range-tree P b B])))
      c-child [c-child]
proof −
  let ?cc = range-tree P a (Fin b)
  let ?direct = edge-relaxation-pairs-in-bound d-child ?cc beta B
  let ?lower = edge-relaxation-pairs-between d-child ?cc b beta
  let ?sources = label-pairs-between d S-pull b beta
  let ?batch = ?direct @ ?lower @ ?sources
  let ?D-next = batch-min-update D-pull ?batch
  have tail:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B
      (dist s) ?D-next b [] [] B [range-tree P b B] []
      (bound-tree P (Fin b) ∪ ⋃(set [range-tree P b B])) 0 []
    by (rule Charged-Direct-Insert-State-Done)
       (use tail-empty b-bound in ‹auto intro: complete-on-dist›)
  show ?thesis
    by (rule Charged-Direct-Insert-State-Step
        [where Bmax = Bmax and S-pull = S-pull and D-pull = D-pull
          and charged-child = ?cc and direct-edge-batch = ?direct
          and lower-edge-batch = ?lower and source-batch = ?sources
          and batch = ?batch and D-next = ?D-next
          and c-pull = 0 and c-direct = 0 and c-lower = 0 and c-sources = 0
          and c-tail = 0])
       (use pull beta-bound pre split reaches a-le-b a-bound small child tail in
         ‹auto intro: complete-on-dist complete-preserved-dist
           simp: partition-pull-cost-bound-def partition-batch-cost-bound-def›)
qed
```

One genuine step, then an arbitrary tail, under $d' = local.dist\ s$. This is the multi-step companion of *charged-loop-single-step-then-done-exists-dist*: rather than forcing the residual partition to be empty (Done) it consumes a ready-made loop state for the band $[b,\ B')$ and stacks one more *Charged-Direct-Insert-State-Step* on top, pulling the band $[a,\ b)$ with a recursive child run. All three completeness obligations and the four partition costs collapse via *complete-on* (*local.dist s*) *?U* / *complete-preserved ?d* (*local.dist s*) *?U*. Iterating this brick is how the path band is covered step by step: each step fans out one level-l child of at most $k * cap$ vertices, so a whole-path cover follows by chaining the cons over the frontier.

lemma *charged-loop-step-cons-exists-dist*:
fixes *a b beta Bmax* :: *real* **and** *B B′* :: *bound* **and** *D D-pull* :: *'a partition-view*
assumes *pull*: *pull-separates D* (*M-of l*) *Bmax S-pull beta D-pull*
and *beta-bound*: *bound-le* (*Fin beta*) *B*
and *pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
and *split*: *S-pull* = *split-below d P beta*
and *reaches*: ∀ *x*∈*S-pull*. *reachable s x*
and *a-le-b*: *a* ≤ *b*
and *a-bound*: *bound-le* (*Fin a*) *B′*
and *small*: *card* (*bound-tree P* (*Fin a*)) < *k* ∗ *cap*
and *child*:
charged-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*
and *D-next*:
D-next = *batch-min-update D-pull*
(*edge-relaxation-pairs-in-bound d-child* (*range-tree P a* (*Fin b*)) *beta B* @
edge-relaxation-pairs-between d-child (*range-tree P a* (*Fin b*)) *b beta* @
label-pairs-between d S-pull b beta)
and *tail*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B*
(*dist s*) *D-next b betas bs B′ charged-tail child-outputs-tail U-tail c-tail*
child-costs-tail
shows *charged-direct-insert-costed-partition-loop-state* Δ *M-of t h k cap l d P B*
(*dist s*) *D a* (*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *charged-tail*) (*U-child* # *child-outputs-tail*)
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* (*range-tree P a* (*Fin b*) # *charged-tail*)))
(*0* + *0* + *0* + *0* + *c-child* + *c-tail*) (*c-child* # *child-costs-tail*)
by (*rule Charged-Direct-Insert-State-Step*
[**where** *Bmax* = *Bmax* **and** *S-pull* = *S-pull* **and** *D-pull* = *D-pull*
and *charged-child* = *range-tree P a* (*Fin b*)
and *direct-edge-batch* = *edge-relaxation-pairs-in-bound d-child* (*range-tree P a* (*Fin b*)) *beta B*
and *lower-edge-batch* = *edge-relaxation-pairs-between d-child* (*range-tree P a* (*Fin b*)) *b beta*
and *source-batch* = *label-pairs-between d S-pull b beta*
and *D-next* = *D-next* **and** *c-pull* = *0* **and** *c-direct* = *0* **and** *c-lower* = *0*
and *c-sources* = *0*])
(*use pull beta-bound pre split reaches a-le-b a-bound small child D-next tail* **in**
‹*auto intro*: *complete-on-dist complete-preserved-dist*
simp: *partition-pull-cost-bound-def partition-batch-cost-bound-def*›)

Charged per-call progress: the success-or-threshold dichotomy. The next five facts are the charged-cost analogue of the direct-insert source-card chain. For every charged run the number of sources is bounded by the cardinality of the run's *own* output U: either the bound was not advanced (success, $B' = B$), so the sources sit inside their own bounded tree, or the FindPivots cap is saturated (threshold, $k * cap \leq card\ U$). This is the stranding-free progress step. It transfers the direct-insert dichotomy to the permissive charged relation *without* the rigid *U-child* = *range-tree P a* (*Fin*

b) step rule, and is therefore non-vacuous on the path family. Note, however, that the bound is against the run's *cumulative* own output, not the disjoint range slice; summing it over the loop children does not on its own telescope to the linear chain (see the loop-level companion below).

PIECE A: charged source-card chain (amortized, stranding-free via the success-or-threshold dichotomy).

theorem *charged-direct-insert-costed-bmssp-source-card-le-if-sound-label-below-output*:
  assumes *run*:
    charged-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′*
  shows *card S* $\leq$ *card U*
proof −
  have *post*: *bmssp-post-full d S B d′ B′ U*
   by (*rule charged-direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
  have *U-eq*: *U* = *bound-tree S B′*
   using *post* **unfolding** *bmssp-post-full-def* **by** *blast*
  have *S-subset-U*: *S* $\subseteq$ *U*
  proof
   fix *x*
   assume *xS*: *x* $\in$ *S*
   have *S-subset*: *S* $\subseteq$ *V*
    using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*
   have *xV*: *x* $\in$ *V*
    using *S-subset xS* **by** *blast*
   have *reach-x*: *reachable s x*
    by (*rule S-reaches*[*OF xS*])
   have *dist-le*: *dist s x* $\leq$ *d x*
    using *sound xV reach-x* **unfolding** *sound-label-def* **by** *blast*
   have *below-dist*: *below-bound* (*dist s x*) *B′*
    using *below-bound-le-trans*[*OF dist-le below*[*OF xS*]] **.**
   show *x* $\in$ *U*
    unfolding *U-eq*
    by (*rule source-in-own-bound-tree*[*OF xS reach-x below-dist*])
  qed
  have *finite-U*: *finite U*
   unfolding *U-eq bound-tree-def* **using** *finite-V* **by** *auto*
  show *?thesis*
   by (*rule card-mono*[*OF finite-U S-subset-U*])
qed

theorem *charged-direct-insert-costed-bmssp-Suc-success-or-threshold*:
  assumes *run*:
    charged-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: ⋀x. x ∈ S ⟹ reachable s x
  shows B' = B ∨ k * cap ≤ card U
  using run
proof cases
  case (Charged-Direct-Insert-Step D c-insert a betas bs charged-Us
      child-outputs U-loop c-loop child-costs-loop)
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have loop-class:
    B' = B ∨ k * cap ≤ card U-loop
    by (rule charged-direct-insert-costed-partition-loop-state-success-or-threshold
      [OF Charged-Direct-Insert-Step(3) sound-fp pivot-pre P-reaches])
  have loop-trace:
    concrete-partition-loop-trace ?P B a bs d' B' charged-Us U-loop
    by (rule charged-direct-insert-costed-partition-loop-state-trace
      [OF Charged-Direct-Insert-Step(3) sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF loop-trace])
  have finite-U-loop: finite U-loop
    using loop-post unfolding bmssp-post-full-def by simp
  show ?thesis
    using loop-class
  proof
    assume B' = B
    then show ?thesis
      by blast
  next
    assume threshold-loop: k * cap ≤ card U-loop
    have U-eq: U = U-loop ∪ ?W
      using Charged-Direct-Insert-Step(5) by simp
    have finite-U: finite U
      unfolding U-eq using finite-U-loop by simp
    have U-loop-subset: U-loop ⊆ U
      unfolding U-eq by blast
    have card U-loop ≤ card U
      by (rule card-mono[OF finite-U U-loop-subset])
    then show ?thesis
```

```
    using threshold-loop by linarith
  qed
qed

theorem charged-direct-insert-costed-bmssp-zero-source-card-le-from-label-below:
  assumes run:
      charged-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
  using run
proof cases
  case (Charged-Direct-Insert-Base x)
  have S-eq: S = {x}
    using Charged-Direct-Insert-Base by simp
  have xS: x ∈ S
    using S-eq by simp
  have xV: x ∈ V
    using pre xS unfolding bmssp-pre-full-def by blast
  have reach-x: reachable s x
    by (rule S-reaches[OF xS])
  have dist-le: dist s x ≤ d x
    using sound xV reach-x unfolding sound-label-def by blast
  have below-dist: below-bound (dist s x) B
    using below-bound-le-trans[OF dist-le below[OF xS]] .
  have x-bound: x ∈ bound-tree {x} B
    by (rule source-in-own-bound-tree[OF - reach-x below-dist]) simp
  have xU: x ∈ U
    using source-in-base-case-vertices[OF x-bound k-pos]
    using Charged-Direct-Insert-Base by simp
  have singleton-subset: {x} ⊆ U
    using xU by blast
  have finite-U: finite U
    using Charged-Direct-Insert-Base by simp
  have card {x} ≤ card U
    by (rule card-mono[OF finite-U singleton-subset])
  then show ?thesis
    using S-eq by simp
qed

theorem charged-direct-insert-costed-bmssp-Suc-source-card-le-from-label-below:
  assumes run:
      charged-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: ⋀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 −
  have call-class: B′ = B ∨ k ∗ cap ≤ card U
    by (rule charged-direct-insert-costed-bmssp-Suc-success-or-threshold
      [OF run sound pre S-reaches])
  then show ?thesis
  proof
    assume B′-eq: B′ = B
    show ?thesis
   proof (rule charged-direct-insert-costed-bmssp-source-card-le-if-sound-label-below-
output
        [OF run sound pre S-reaches])
      fix x
      assume x ∈ S
      then show below-bound (d x) B′
        using below B′-eq by simp
    qed
  next
    assume threshold: k ∗ cap ≤ card U
    have cap ≤ k ∗ cap
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-cap threshold by linarith
  qed
qed

theorem charged-direct-insert-costed-bmssp-source-card-le-from-label-below-all-
levels:
  assumes run:
      charged-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: ⋀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 (cases l)
  case 0
  have run0:
    charged-direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d′ B′ U c
    using run 0 by simp
  show ?thesis
    by (rule charged-direct-insert-costed-bmssp-zero-source-card-le-from-label-below
      [OF run0 sound pre S-reaches below k-pos])
next
```

case (*Suc l'*)
have *run-suc*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap* (*Suc l'*) *d S B d' B' U c*
using *run Suc* **by** *simp*
show *?thesis*
by (*rule charged-direct-insert-costed-bmssp-Suc-source-card-le-from-label-below*
[*OF run-suc sound pre S-reaches below S-cap k-pos*])
qed

Loop-level companion: the charged partition-loop cost is bounded by the linear range-tree chain cardinality as soon as the pulled child sources land in their range slices (the transparent no-stranding/settledness invariant).

lemma *list-all2-card-mono-if-finite-right*:
assumes *list-all2* (⊆) *Xs Ys*
and ⋀*Y*. *Y* ∈ *set Ys* ⟹ *finite Y*
shows *list-all2* (λ*X Y*. *card X* ≤ *card Y*) *Xs Ys*
using *assms*
proof (*induction rule*: *list-all2-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons X Xs Y Ys*)
have *fY*: *finite Y* **using** *Cons.prems* **by** *simp*
have *tail-fin*: ⋀*Yy*. *Yy* ∈ *set Ys* ⟹ *finite Yy* **using** *Cons.prems* **by** *simp*
have *card X* ≤ *card Y* **using** *card-mono*[*OF fY Cons.hyps*(*1*)] **.**
moreover have *list-all2* (λ*X Y*. *card X* ≤ *card Y*) *Xs Ys*
using *Cons.IH*[*OF tail-fin*] **.**
ultimately show *?case* **by** *simp*
qed

lemma *finite-range-tree-child-list-elem*:
Y ∈ *set* (*range-tree-child-list S a bs*) ⟹ *finite Y*
by (*induction bs arbitrary*: *a*) *auto*

lemma *list-all2-subset-imp-card-le-range-tree-child-list*:
assumes *list-all2* (⊆) *Xs* (*range-tree-child-list S a bs*)
shows *list-all2* (λ*X R*. *card X* ≤ *card R*) *Xs* (*range-tree-child-list S a bs*)
by (*rule list-all2-card-mono-if-finite-right*[*OF assms*])
(*rule finite-range-tree-child-list-elem*)

theorem *charged-direct-insert-costed-partition-loop-state-source-sum-le-chain-if-settled*:
assumes *loop*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
betas bs B' charged-Us child-outputs U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
shows ∃ *child-sources*.

length child-sources = length bs ∧
list-all2
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
(∀ *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-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 ∗ *sum-list* (*map card child-sources*) ∧
(*list-all2* (⊆) *child-sources* (*range-tree-child-list P a bs*) ⟶
sum-list (*map card child-sources*) ≤ *card* (*range-tree-chain P a bs B′*))
proof −
have *mono*: *nondecreasing-from a bs*
using *charged-direct-insert-costed-partition-loop-state-mono*[*OF loop*] **.**
obtain *child-sources* **where**
L: *length child-sources = length bs* **and**
A2: *list-all2*
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
(∀ *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*) **and**
C: *c* ≤ *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 ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
h ∗ *sum-list* (*map card child-sources*)
using *charged-direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-and-edge-ranges*[*OF loop sound pre reaches*]
by *blast*
have *settle-imp*:
list-all2 (⊆) *child-sources* (*range-tree-child-list P a bs*) ⟶
sum-list (*map card child-sources*) ≤ *card* (*range-tree-chain P a bs B′*)
proof
assume *sub*: *list-all2* (⊆) *child-sources* (*range-tree-child-list P a bs*)
have *dom*: *list-all2* (λ*X R*. *card X* ≤ *card R*) *child-sources* (*range-tree-child-list P a bs*)
using *list-all2-subset-imp-card-le-range-tree-child-list*[*OF sub*] **.**

```
    show sum-list (map card child-sources) ≤ card (range-tree-chain P a bs B')
      by (rule sum-card-dominated-by-range-tree-child-list-le-chain[OF mono dom])
  qed
  show ?thesis
    using L A2 C settle-imp by blast
qed
```

Explicit linear per-loop cost. Substituting the settled source bound into the cost decomposition turns the only non-telescoping source term $(1 + h) * \mathit{sum\text{-}list}\ (\mathit{map\ card\ child\text{-}sources})$ into a linear function of the produced range-tree chain. Under the transparent settledness invariant the charged partition-loop cost is therefore bounded by the recursive child costs plus $(1 + h) * \mathit{card}\ (\mathit{range\text{-}tree\text{-}chain\ P\ a\ bs\ B'})$ plus the edge-range terms (which telescope linearly through the shared *range-tree-child-list* machinery, exactly as in the direct-insert level bound). This is the cost half of the charged per-level closer; the only remaining obligation to discharge the charged top-level bound is the settledness hypothesis itself, i.e. the global amortized no-stranding invariant.

```
theorem charged-direct-insert-costed-partition-loop-state-cost-le-linear-if-settled:
  assumes loop:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' charged-Us child-outputs U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and reaches: ⋀x. x ∈ P ⟹ reachable s x
  shows ∃ child-sources.
    length child-sources = length bs ∧
    list-all2
      (λS-child charged-child. ∃ c-child U-child B-child d-child B-child'.
        charged-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 ∧
        (∀ 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) ∧
    (list-all2 (⊆) child-sources (range-tree-child-list P a bs) ⟶
      c ≤ sum-list child-costs
          + (1 + h) * card (range-tree-chain P a bs B')
          + 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 −
  obtain child-sources where
    L: length child-sources = length bs and
    A2: list-all2
      (λS-child charged-child. ∃ c-child U-child B-child d-child B-child'.
```

charged-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 ∧
(∀ *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*) **and**
C: *c* ≤ *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 ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
h ∗ *sum-list* (*map card child-sources*) **and**
settle-imp:
list-all2 (⊆) *child-sources* (*range-tree-child-list P a bs*) ⟶
sum-list (*map card child-sources*) ≤ *card* (*range-tree-chain P a bs B′*)
using *charged-direct-insert-costed-partition-loop-state-source-sum-le-chain-if-settled*
[*OF loop sound pre reaches*]
by *blast*
have *linear-imp*:
list-all2 (⊆) *child-sources* (*range-tree-child-list P a bs*) ⟶
c ≤ *sum-list child-costs*
+ (*1* + *h*) ∗ *card* (*range-tree-chain P a bs B′*)
+ *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
assume *sub*: *list-all2* (⊆) *child-sources* (*range-tree-child-list P a bs*)
let *?SC* = *sum-list* (*map card child-sources*)
let *?K* = *card* (*range-tree-chain P a bs B′*)
have *SCK*: *?SC* ≤ *?K* **using** *settle-imp sub* **by** *blast*
have *c* ≤ *sum-list child-costs* + *?SC* + *h* ∗ *?SC* +
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*))
using *C* **by** *simp*
also have *sum-list child-costs* + *?SC* + *h* ∗ *?SC* = *sum-list child-costs* + (*1* + *h*) ∗ *?SC*
by *simp*
finally have *step1*:
c ≤ *sum-list child-costs* + (*1* + *h*) ∗ *?SC*
+ *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*))
by *simp*

```
  have (1 + h) * ?SC ≤ (1 + h) * ?K
    by (rule mult-le-mono2[OF SCK])
  then show c ≤ sum-list child-costs
        + (1 + h) * ?K
        + 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))
    using step1 by linarith
 qed
 show ?thesis using L A2 linear-imp by blast
qed
```

Fully-linear per-loop local budget under settledness. The two edge-range sums telescope to *edge-count* through the shared *range-tree-child-edge-range-list* / *range-tree-child-direct-edge-range-list* disjointness lemmas (the same ones the direct-insert level bound uses), so under the settledness invariant the charged partition-loop cost reduces to the canonical local budget *sum-list child-costs* + (1 + *h*) ∗ *card* (*range-tree-chain P a bs B′*) + (*t* + *h*) ∗ *edge-count*: recursive child costs, a linear function of the produced range-tree chain, and a linear function of the edge count. This is precisely the per-level local-budget shape consumed by the direct-insert linchpin induction; the charged top-level bound now reduces to threading the single settledness hypothesis through that induction.

```
theorem charged-direct-insert-costed-partition-loop-state-cost-local-budget-if-settled:
 assumes loop:
   charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
     betas bs B′ charged-Us child-outputs U c child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
   and reaches: ⋀x. x ∈ P ⟹ reachable s x
 shows ∃ child-sources.
   length child-sources = length bs ∧
   list-all2
     (λS-child charged-child. ∃ c-child U-child B-child d-child B-child′.
       charged-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 ∧
       (∀ 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) ∧
   (list-all2 (⊆) child-sources (range-tree-child-list P a bs) ⟶
     c ≤ sum-list child-costs
       + (1 + h) * card (range-tree-chain P a bs B′)
       + (t + h) * edge-count)
proof −
 have mono: nondecreasing-from a bs
```

using *charged-direct-insert-costed-partition-loop-state-mono*[*OF loop*] .
obtain *child-sources* **where**
L: *length child-sources* = *length bs* **and**
A2: *list-all2*
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
(∀ *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*) **and**
lin-imp:
list-all2 (⊆) *child-sources* (*range-tree-child-list P a bs*) ⟶
c ≤ *sum-list child-costs*
+ (*1* + *h*) ∗ *card* (*range-tree-chain P a bs B′*)
+ *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*))
using *charged-direct-insert-costed-partition-loop-state-cost-le-linear-if-settled*
[*OF loop sound pre reaches*]
by *blast*
have *edge-direct*:
sum-list (*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*))
≤ *edge-count*
by (*rule sum-card-range-tree-child-direct-edge-range-list-le-edge-count*[*OF mono*])
have *edge-lower*:
sum-list (*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *edge-count*
by (*rule sum-card-range-tree-child-edge-range-list-le-edge-count*[*OF mono*])
have *budget-imp*:
list-all2 (⊆) *child-sources* (*range-tree-child-list P a bs*) ⟶
c ≤ *sum-list child-costs*
+ (*1* + *h*) ∗ *card* (*range-tree-chain P a bs B′*)
+ (*t* + *h*) ∗ *edge-count*
proof
assume *sub*: *list-all2* (⊆) *child-sources* (*range-tree-child-list P a bs*)
let *?K* = *card* (*range-tree-chain P a bs B′*)
have *c-le*:
c ≤ *sum-list child-costs* + (*1* + *h*) ∗ *?K*
+ *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*))
using *lin-imp sub* **by** *blast*
have *t-bound*: *t* ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*))
≤ *t* ∗ *edge-count*

```
    by (rule mult-le-mono2[OF edge-direct])
  have h-bound: h * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs))
      ≤ h * edge-count
    by (rule mult-le-mono2[OF edge-lower])
  have dist: (t + h) * edge-count = t * edge-count + h * edge-count
    by (simp add: algebra-simps)
  show c ≤ sum-list child-costs + (1 + h) * ?K + (t + h) * edge-count
    using c-le t-bound h-bound unfolding dist by linarith
 qed
 show ?thesis using L A2 budget-imp by blast
qed
```

Increment monotonicity bridge. A child run's cost is bounded by its *own* range-tree increment *range-tree* (*split-below d P beta*) *a B*; because the pulled source *split-below d P beta* sits inside the parent pivot tree, that increment is contained in the parent slice *range-tree P a B*, and *level-range-cost-bound* is monotone in the set argument. Hence the intrinsic child bound transfers to the slice bound the loop closer consumes — using only the easy containment direction, with no cut-equality obligation.

```
lemma tree-set-mono: X ⊆ Y ⟹ tree-set X ⊆ tree-set Y
 unfolding tree-set-def by blast

lemma range-tree-mono-tree-set:
 tree-set S ⊆ tree-set P ⟹ range-tree S a B ⊆ range-tree P a B
 unfolding range-tree-def by blast

lemma range-tree-split-below-subset:
 range-tree (split-below d P beta) a B ⊆ range-tree P a B
 by (rule range-tree-mono-tree-set, rule tree-set-mono, rule split-below-subset)

lemma level-range-cost-bound-mono:
 assumes sub: X ⊆ Y and finY: finite Y
 shows level-range-cost-bound A R L X ≤ level-range-cost-bound A R L Y
proof −
 have card X ≤ card Y by (rule card-mono[OF finY sub])
 moreover have card (outgoing-edges X) ≤ card (outgoing-edges Y)
   by (rule card-mono[OF finite-outgoing-edges outgoing-edges-mono[OF sub]])
 ultimately show ?thesis
   unfolding level-range-cost-bound-def by (intro add-mono mult-le-mono2)
qed

lemma level-range-cost-bound-range-tree-split-below-le:
 assumes finP: finite (range-tree P a B)
 shows level-range-cost-bound A R L (range-tree (split-below d P beta) a B)
     ≤ level-range-cost-bound A R L (range-tree P a B)
 by (rule level-range-cost-bound-mono[OF range-tree-split-below-subset finP])
```

Increment-faithful charged loop closer. The same per-loop

accounting as *charged-direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-ranges*, but the produced bound is kept against the range-tree *chain range-tree-chain P a bs B′* — the freshly-settled increment of the loop — rather than being loosened to the loop output *U*. Keeping the chain is what lets the bound telescope across the recursion: a child's loop charges only its own increment, and sibling increments are disjoint. The source term *Suc h ∗ card* (*range-tree-chain P a bs B′*) is absorbed into the vertex coefficient through *Suc h ≤ A*.

theorem *charged-direct-insert-costed-partition-loop-state-closes-increment-bound-from-child-bound-with-invariants*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs 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 *P-k-cap*: *k ∗ card P ≤ cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:
⋀*c-child charged-child U-child S-child B-child d-child B-child′*.
⟦*charged-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;
⋀*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 charged-child
and *source-factor*: *Suc h ≤ A*
and *source-progress*:
⋀*child-sources*.
list-all2
(*λS-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
(∀ *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*) *⟹*
sum-list (*map card child-sources*) ≤
card (*range-tree-chain P a bs B′*)
shows *c* ≤
A ∗ Suc 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-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 −
  have P-subset: P ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have child-cost-bounds:
    list-all2 (λc-child charged-child.
      c-child ≤ level-range-cost-bound A R L charged-child)
      child-costs (range-tree-child-list P a bs)
    by (rule charged-direct-insert-costed-partition-loop-state-child-cost-bounds-with-
invariants
      [OF run P-subset P-k-cap P-anti child-bound])
  obtain child-sources where sources:
      list-all2
        (λS-child charged-child. ∃ c-child U-child B-child d-child B-child′.
          charged-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 ∧
          (∀ 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)
    and cost:
      c ≤ 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 * sum-list
          (map card (range-tree-child-edge-range-list P a betas bs)) +
        h * sum-list (map card child-sources)
   using charged-direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-
and-edge-ranges
      [OF run sound pre P-reaches] by blast
  have source-sum:
    sum-list (map card child-sources) ≤ card (range-tree-chain P a bs B′)
    by (rule source-progress[OF sources])
  have mono: nondecreasing-from a bs
    by (rule charged-direct-insert-costed-partition-loop-state-mono[OF run])
  have child-sum:
    sum-list child-costs ≤
      A * L * card (range-tree-chain P a bs B′) +
      R * card (outgoing-edges (range-tree-chain P a bs B′))
    by (rule child-costs-le-level-range-child-list-bound[OF mono child-cost-bounds])
  let ?K = card (range-tree-chain P a bs B′)
  let ?OE = card (outgoing-edges (range-tree-chain P a bs B′))
  let ?D = sum-list (map card (range-tree-child-direct-edge-range-list P B a betas
bs))
  let ?E = sum-list (map card (range-tree-child-edge-range-list P a betas bs))
  let ?SC = sum-list (map card child-sources)
  have src-abs: ?SC + h * ?SC ≤ A * ?K
```

```
  proof −
    have ?SC + h * ?SC = Suc h * ?SC by simp
    also have ... ≤ Suc h * ?K using source-sum by (rule mult-le-mono2)
    also have ... ≤ A * ?K using source-factor by (rule mult-le-mono1)
    finally show ?thesis .
  qed
  have c ≤ (A * L * ?K + R * ?OE) + ?SC + t * ?D + h * ?E + h * ?SC
    using cost child-sum by linarith
  also have ... = A * L * ?K + R * ?OE + t * ?D + h * ?E + (?SC + h *
?SC)
    by (simp add: algebra-simps)
  also have ... ≤ A * L * ?K + R * ?OE + t * ?D + h * ?E + A * ?K
    using src-abs by linarith
  also have ... = A * Suc L * ?K + R * ?OE + t * ?D + h * ?E
    by (simp add: algebra-simps)
  finally show ?thesis .
qed
```

Charged amortized accounting (B2). The charged counterpart of the direct-insert amortized chain (*direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs*). The pulled child sources are absorbed into the produced range-tree chain via the per-loop *source-progress* settledness oracle, and the edge-range sums telescope through the shared disjointness lemmas exactly as in the direct-insert development. These reduce the charged top-level refined bound to the single settledness side-condition.

Increment-monotonicity bridge for the amortized budget. The amortized cost budget is monotone in its vertex-set argument (all three of *card*, *outgoing-edges*, and *outgoing-edges-range* are), so a child run's intrinsic amortized bound against its own range-tree increment *range-tree* (*split-below d P beta*) *a B′* transfers to the parent slice *range-tree P a B′* that the loop closer consumes — using only the easy containment direction *range-tree-split-below-subset*.

```
lemma bmssp-amortized-cost-bound-mono:
  assumes sub: X ⊆ Y and finY: finite Y
  shows bmssp-amortized-cost-bound A R h t q L B X
      ≤ bmssp-amortized-cost-bound A R h t q L B Y
proof −
  have cX: card X ≤ card Y by (rule card-mono[OF finY sub])
  have cOE: card (outgoing-edges X) ≤ card (outgoing-edges Y)
    by (rule card-mono[OF finite-outgoing-edges outgoing-edges-mono[OF sub]])
  have cOER: card (outgoing-edges-range X 0 B) ≤ card (outgoing-edges-range Y
0 B)
    by (rule card-mono[OF finite-outgoing-edges-range outgoing-edges-range-mono-
sources[OF sub]])
  show ?thesis
    unfolding bmssp-amortized-cost-bound-def
    using cX cOE cOER by (intro add-mono mult-le-mono2)
```

qed

lemma *bmssp-amortized-cost-bound-range-tree-split-below-le*:
assumes *finP*: *finite* (*range-tree P a B′*)
shows *bmssp-amortized-cost-bound A R h t q L B*
(*range-tree* (*split-below d P beta*) *a B′*)
≤ *bmssp-amortized-cost-bound A R h t q L B* (*range-tree P a B′*)
by (*rule bmssp-amortized-cost-bound-mono*[*OF range-tree-split-below-subset finP*])

theorem *charged-cost-from-child-source-and-amortized-bounds*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ charged-Us child-outputs 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 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 P a betas bs*)
and *source-progress*:
⋀*child-sources*.
list-all2
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
(∀ *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*) ⟹
sum-list (*map card child-sources*) ≤
card (*range-tree-chain P a bs B′*)
shows *c* ≤
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 −
obtain *child-sources* **where** *sources*:
list-all2
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-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 ∧
      (∀ 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)
 and cost:
  c ≤ 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 * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs)) +
    h * sum-list (map card child-sources)
 using charged-direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-
and-edge-ranges
   [OF run sound pre P-reaches] by blast
 have mono: nondecreasing-from a bs
  by (rule charged-direct-insert-costed-partition-loop-state-mono[OF run])
 have source-sum:
  sum-list (map card child-sources) ≤ card (range-tree-chain P a bs B′)
  by (rule source-progress[OF sources])
 have child-sum-raw:
  sum-list child-costs ≤
    A * L * sum-list (map card (range-tree-child-list P a bs)) +
    (R + q * h) *
      sum-list (map (λU. card (outgoing-edges U))
        (range-tree-child-list P a bs)) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs))
  by (rule sum-bmssp-amortized-child-bounds-le[OF child-cost-bounds])
 have child-card-sum:
  sum-list (map card (range-tree-child-list P a bs)) ≤
    card (range-tree-chain P a bs B′)
  by (rule card-range-tree-child-list-le-chain[OF mono])
 have child-out-sum:
  sum-list (map (λU. card (outgoing-edges U))
      (range-tree-child-list P a bs)) ≤
    card (outgoing-edges (range-tree-chain P a bs B′))
  by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
 have child-sum:
  sum-list child-costs ≤
    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))
 proof −
  have vterm:
    A * L * sum-list (map card (range-tree-child-list P a bs)) ≤
     A * L * card (range-tree-chain P a bs B′)
    using child-card-sum by simp
```

```
  have eterm:
    (R + q * h) *
      sum-list (map (λU. card (outgoing-edges U))
        (range-tree-child-list P a bs)) ≤
     (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B'))
    using child-out-sum by simp
  show ?thesis
    using child-sum-raw vterm eterm by linarith
 qed
 have c ≤
    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)) +
    sum-list (map card child-sources) +
    h * sum-list (map card child-sources)
  using cost child-sum by linarith
 also have ... ≤
    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 −
  have h-part:
    h * sum-list (map card child-sources) ≤
      h * card (range-tree-chain P a bs B')
    using source-sum by simp
  have sum-list (map card child-sources) +
      h * sum-list (map card child-sources) ≤
    card (range-tree-chain P a bs B') +
      h * card (range-tree-chain P a bs B')
    using source-sum h-part by linarith
  then have sum-list (map card child-sources) +
      h * sum-list (map card child-sources) ≤
    Suc h * card (range-tree-chain P a bs B')
    by (simp add: algebra-simps)
  then show ?thesis by linarith
 qed
 finally show ?thesis .
qed
```

theorem *charged-direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs 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*:
$\bigwedge$*child-sources*.
list-all2
(λ*S-child charged-child*. $\exists$ *c-child U-child B-child d-child B-child′*.
charged-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*) $\Longrightarrow$
sum-list (*map card child-sources*) $\leq$
card (*range-tree-chain P a bs B′*)
shows *c* $\leq$ *bmssp-amortized-cost-bound A R h t* (*Suc q*) (*Suc L*) *B U*
proof −
have *trace*: *concrete-partition-loop-trace P B a bs d′ B′ charged-Us U*
by (*rule charged-direct-insert-costed-partition-loop-state-trace* [*OF run sound pre P-reaches*])
have *children*:
list-all2 (λ*U X. U* = *X* $\wedge$ *complete-on d′ U*) *charged-Us* (*range-tree-chain-list P a bs B′*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *U-def*: *U* = *bound-tree P* (*Fin a*) $\cup$ $\bigcup$(*set charged-Us*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *Union-eq*:
$\bigcup$(*set charged-Us*) = $\bigcup$(*set* (*range-tree-chain-list P a bs B′*))
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
U = *bound-tree P* (*Fin a*) $\cup$ *range-tree-chain P a bs B′*
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B′*] **by** *simp*
have *finite-U*: *finite U*
using *U-eq-chain* **by** *simp*
have *chain-subset*: *range-tree-chain P a bs B′* $\subseteq$ *U*
using *U-eq-chain* **by** *blast*
have *card-chain-le*: *card* (*range-tree-chain P a bs B′*) $\leq$ *card U*

by (*rule card-mono*[*OF finite-U chain-subset*])
have *outgoing-subset*:
outgoing-edges (*range-tree-chain P a bs B′*) ⊆ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*:
card (*outgoing-edges* (*range-tree-chain P a bs B′*)) ≤
card (*outgoing-edges U*)
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *range-subset*:
outgoing-edges-range (*range-tree-chain P a bs B′*) *0 B* ⊆
outgoing-edges-range U 0 B
by (*rule outgoing-edges-range-mono-sources*[*OF chain-subset*])
have *card-range-le*:
card (*outgoing-edges-range* (*range-tree-chain P a bs B′*) *0 B*) ≤
card (*outgoing-edges-range U 0 B*)
by (*rule card-mono*[*OF finite-outgoing-edges-range range-subset*])
have *mono*: *nondecreasing-from a bs*
by (*rule charged-direct-insert-costed-partition-loop-state-mono*[*OF run*])
have *beta-bounds*: *bounds-le B betas*
by (*rule charged-direct-insert-costed-partition-loop-state-beta-bounds*[*OF run*])
have *cost*:
c ≤
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′*)
by (*rule charged-cost-from-child-source-and-amortized-bounds*
[*OF run sound pre P-reaches child-cost-bounds source-progress*])
have *lower-sum*:
sum-list (*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*))
by (*rule sum-card-range-tree-child-edge-range-list-le-outgoing-edges-chain*
[*OF mono*])
have *direct-sum*:
sum-list
(*map card* (*range-tree-child-zero-edge-range-list P a betas bs*)) +
sum-list
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*))
≤ *card* (*outgoing-edges-range* (*range-tree-chain P a bs B′*) *0 B*)
by (*rule sum-card-range-tree-child-zero-direct-edge-ranges-le-outgoing-edges-range-chain*
[*OF mono beta-bounds*])

```
have vertex-part:
  A * L * card (range-tree-chain P a bs B') +
    Suc h * card (range-tree-chain P a bs B') ≤
   A * Suc L * card U
proof −
  have source-term:
    Suc h * card (range-tree-chain P a bs B') ≤
     A * card U
  proof −
    have Suc h * card (range-tree-chain P a bs B') ≤
        A * card (range-tree-chain P a bs B')
      by (rule mult-right-mono[OF source-factor]) simp
    also have ... ≤ A * card U
      using card-chain-le by simp
    finally show ?thesis .
  qed
  have child-term:
    A * L * card (range-tree-chain P a bs B') ≤
     A * L * card U
    using card-chain-le by simp
  have A * L * card U + A * card U = A * Suc L * card U
    by (simp add: algebra-simps)
  then show ?thesis
    using child-term source-term by linarith
qed
have edge-part:
  (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs))
   ≤ (R + Suc q * h) * card (outgoing-edges U)
proof −
  have lower-term:
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs))
     ≤ h * card (outgoing-edges (range-tree-chain P a bs B'))
    using lower-sum by simp
  have chain-term:
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
      h * card (outgoing-edges (range-tree-chain P a bs B')) =
     (R + Suc q * h) *
      card (outgoing-edges (range-tree-chain P a bs B'))
    by (simp add: algebra-simps)
  have to-chain:
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
      h * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs))
     ≤ (R + Suc q * h) *
      card (outgoing-edges (range-tree-chain P a bs B'))
    using lower-term chain-term by linarith
```

```
    have to-U:
      (R + Suc q * h) *
        card (outgoing-edges (range-tree-chain P a bs B')) ≤
       (R + Suc q * h) * card (outgoing-edges U)
      using card-out-le by simp
    show ?thesis
      using to-chain to-U by linarith
  qed
  have direct-part:
    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))
     ≤ t * card (outgoing-edges-range U 0 B)
  proof −
    have t * (sum-list
        (map card (range-tree-child-zero-edge-range-list P a betas bs)) +
      sum-list
        (map card (range-tree-child-direct-edge-range-list P B a betas bs)))
      ≤ t * card (outgoing-edges-range (range-tree-chain P a bs B') 0 B)
      using direct-sum by simp
    also have ... ≤ t * card (outgoing-edges-range U 0 B)
      using card-range-le by simp
    finally show ?thesis
      by (simp add: algebra-simps)
  qed
  have c ≤
      A * Suc L * card U +
      (R + Suc q * h) * card (outgoing-edges U) +
      t * card (outgoing-edges-range U 0 B)
    using cost vertex-part edge-part direct-part by linarith
  then show ?thesis
    unfolding bmssp-amortized-cost-bound-def .
qed

theorem charged-direct-insert-costed-nonbase-step-closes-amortized-bound-from-
child-costs:
  assumes loop:
    charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' charged-Us child-outputs U-loop c-loop 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 (λc-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 P a betas bs)
    and source-factor: Suc h ≤ A
    and source-progress:
```

```
    ⋀child-sources.
      list-all2
        (λS-child charged-child. ∃ c-child U-child B-child d-child B-child'.
          charged-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 ∧
          (∀ 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) ⟹
      sum-list (map card child-sources) ≤
        card (range-tree-chain P a bs B')
  and U-def: U = U-loop ∪ W
  and finite-U: finite U
  and scan-insert: c-scan-insert ≤ A * card U
  and c-def: c = c-scan-insert + c-loop
 shows c ≤
  bmssp-amortized-cost-bound A R h t (Suc q) (Suc (Suc L)) B U
proof −
 have loop-bound:
  c-loop ≤
    bmssp-amortized-cost-bound A R h t (Suc q) (Suc L) B U-loop
  by (rule charged-direct-insert-costed-partition-loop-state-closes-amortized-bound-
from-child-costs
    [OF loop sound pre P-reaches child-cost-bounds source-factor source-progress])
 have U-loop-subset: U-loop ⊆ U
  using U-def by blast
 have card-loop-le: card U-loop ≤ card U
  by (rule card-mono[OF finite-U U-loop-subset])
 have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
  by (rule outgoing-edges-mono[OF U-loop-subset])
 have finite-out-U: finite (outgoing-edges U)
  by simp
 have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
  by (rule card-mono[OF finite-out-U outgoing-subset])
 have range-subset:
  outgoing-edges-range U-loop 0 B ⊆ outgoing-edges-range U 0 B
  by (rule outgoing-edges-range-mono-sources[OF U-loop-subset])
 have card-range-le:
  card (outgoing-edges-range U-loop 0 B) ≤
    card (outgoing-edges-range U 0 B)
  by (rule card-mono[OF finite-outgoing-edges-range range-subset])
 have loop-to-U:
  c-loop ≤
    A * Suc L * card U +
    (R + Suc q * h) * card (outgoing-edges U) +
    t * card (outgoing-edges-range U 0 B)
 proof −
  have vterm:
```

```
    A * Suc L * card U-loop ≤ A * Suc L * card U
    using card-loop-le by simp
  have eterm:
    (R + Suc q * h) * card (outgoing-edges U-loop) ≤
     (R + Suc q * h) * card (outgoing-edges U)
    using card-out-le by simp
  have rterm:
    t * card (outgoing-edges-range U-loop 0 B) ≤
     t * card (outgoing-edges-range U 0 B)
    using card-range-le by simp
  show ?thesis
    using loop-bound vterm eterm rterm
    unfolding bmssp-amortized-cost-bound-def by linarith
qed
have c ≤
    A * card U +
    (A * Suc L * card U +
     (R + Suc q * h) * card (outgoing-edges U) +
     t * card (outgoing-edges-range U 0 B))
  using scan-insert loop-to-U c-def by linarith
also have ... =
    A * Suc (Suc L) * card U +
    (R + Suc q * h) * card (outgoing-edges U) +
    t * card (outgoing-edges-range U 0 B)
  by (simp add: algebra-simps)
finally show ?thesis
  unfolding bmssp-amortized-cost-bound-def .
qed
```

Charged amortized child-cost propagator. The amortized counterpart of *charged-direct-insert-costed-partition-loop-state-child-cost-bounds-with-invariants*: it threads an amortized per-child budget (stated against the disjoint range-tree slice *charged-child*) across the partition loop, producing the list the amortized loop closer consumes. In a recursion-level induction the per-child budget is discharged by the inductive hypothesis against the child increment and the increment->slice bridge *bmssp-amortized-cost-bound-range-tree-split-below-le*.

```
theorem charged-direct-insert-costed-partition-loop-state-child-amortized-cost-bounds-with-invariants:
  charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' charged-Us child-outputs U c child-costs ⟹
  P ⊆ V ⟹
  k * card P ≤ cap ⟹
  tree-antichain P ⟹
  (⋀c-child charged-child U-child S-child B-child d-child B-child'.
     ⟦charged-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;
      ⋀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*;
k $*$ *card S-child* $\leq$ *cap*;
tree-antichain S-child$\rrbracket$
$\Longrightarrow$ *c-child* $\leq$ *bmssp-amortized-cost-bound A R h t q L B-child charged-child*)
$\Longrightarrow$
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 *charged-direct-insert-costed-bmssp-child-amortized-cost-bounds-with-invariants-trivial*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$ *True*
proof (*induction rule*:
charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d' P* Δ *M-of t h k cap l d*)
then show *?case* **by** *simp*
next
case (*Charged-Direct-Insert-State-Stop a B d' P k cap* Δ *M-of t h l d D*)
then show *?case* **by** *simp*
next
case (*Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d*
P a b B' d' k cap Δ *t h d-child U-child c-child charged-child*
direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
c-direct c-lower c-sources betas bs charged-tail child-outputs-tail
U-tail c-tail child-costs-tail c)
have *tail*:
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-tail (*range-tree-child-bound-pair-list P b betas bs*)
using *Charged-Direct-Insert-State-Step.IH Charged-Direct-Insert-State-Step.prems*
by *blast*
have *card-pull*: *card S-pull* $\leq$ *M-of l*
using *Charged-Direct-Insert-State-Step* **unfolding** *pull-separates-def* **by** *blast*
have *below-pull*: $\bigwedge x.\ x \in$ *S-pull* $\Longrightarrow$ *below-bound* (*d x*) (*Fin beta*)
proof −
fix *x* **assume** *xS*: *x* $\in$ *S-pull*
have *S-pull* = *split-below d P beta* **using** *Charged-Direct-Insert-State-Step* **by** *blast*
then show *below-bound* (*d x*) (*Fin beta*) **using** *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *pull-k-cap*: *k* $*$ *card S-pull* $\leq$ *cap*
proof −
have *S-pull* = *split-below d P beta* **using** *Charged-Direct-Insert-State-Step* **by** *blast*
then show *?thesis*
using *split-below-scaled-card-le*

```
      [OF Charged-Direct-Insert-State-Step.prems(1)
        Charged-Direct-Insert-State-Step.prems(2), of d beta] by simp
  qed
  have pull-anti: tree-antichain S-pull
  proof −
    have S-pull = split-below d P beta using Charged-Direct-Insert-State-Step by
blast
    then show ?thesis
      using split-below-tree-antichain
        [OF Charged-Direct-Insert-State-Step.prems(3), of d beta] by simp
  qed
  have child-run:
    charged-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
    using Charged-Direct-Insert-State-Step by blast
  have pre-pull: bmssp-pre-full d S-pull (Fin beta)
    using Charged-Direct-Insert-State-Step by blast
  have reaches-pull: ⋀x. x ∈ S-pull ⟹ reachable s x
    using Charged-Direct-Insert-State-Step by blast
  have charged-child-eq: charged-child = range-tree P a (Fin b)
    using Charged-Direct-Insert-State-Step by blast
  have head:
    c-child ≤ bmssp-amortized-cost-bound A R h t q L (Fin beta) charged-child
    by (rule Charged-Direct-Insert-State-Step.prems(4)
      [OF child-run pre-pull reaches-pull card-pull below-pull
        pull-k-cap pull-anti])
  show ?case
    using head tail Charged-Direct-Insert-State-Step charged-child-eq by simp
next
  case (Charged-Direct-Insert-Base S x Δ M-of t h k cap d B)
  then show ?case by simp
next
  case (Charged-Direct-Insert-Step D k cap d S B c-insert t Δ M-of h l
      d′ a betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c)
  then show ?case by simp
qed

end

end
theory BMSSP-NonVacuous-Family
  imports BMSSP-Path-Family-Extended BMSSP-Cost-Totality
begin
```

10 An unconditional, non-vacuous size-parametric BMSSP running-time bound

The conditional charged headline *bmssp-path-family-charged-direct-insert-runtime-bigo-size-if-cost-bounded* runs the path P_k at the *same* index k that also fixes the internal logarithmic schedule. On that diagonal the top-level bucket cap $cap\ k = p\ k \cdot 2^{p\ k \cdot q\ k}$ with $p\ k =$ *sssp-log-one-third-param* k and $q\ k =$ *sssp-log-two-thirds-param* k is sublinear in k, so it eventually fails to cover the $k + 1$ path vertices and no charged top-level run can exist — the diagonal family is vacuous.

This theory *decouples* the graph size from the schedule index: it keeps the genuine growing graph P_n (with $card\ (path\text{-}k\text{-}V\ n) = Suc\ n \to \infty$) but runs it at the *inflated* schedule index $n \cdot n$. Because $cap\ (n \cdot n) \geq (n^2 + 2)^{ln\ 2} \geq n^{2\ ln\ 2}$ and $2\ ln\ 2 > 1$, the cap now covers the path for all but finitely many n, while $ln\ (n \cdot n) = 2\ ln\ n$ keeps the schedule polylog in $\Theta(ln\ n)$. This is milestone **B0**: the arithmetic that de-risks the whole construction.

10.1 Elementary analytic facts

A power with exponent > 1 eventually dominates the successor function.

```
lemma real-Suc-le-powr-eventually:
  fixes c :: real
  assumes c1: 1 < c
  shows eventually (λn::nat. real (Suc n) ≤ (real n) powr c) at-top
proof −
  have cpos: 0 < c − 1
    using c1 by simp
  have ln-top: filterlim (λn::nat. ln (real n)) at-top at-top
    by (rule filterlim-compose[OF ln-at-top filterlim-real-sequentially])
  have ev-ln: eventually (λn::nat. ln 2 / (c − 1) ≤ ln (real n)) at-top
    using ln-top unfolding filterlim-at-top by blast
  have ev-n: eventually (λn::nat. (1::nat) ≤ n) at-top
    by (rule eventually-ge-at-top)
  from ev-ln ev-n show ?thesis
  proof eventually-elim
    case (elim n)
    have n1: 1 ≤ real n
      using elim(2) by simp
    have npos: 0 < real n
      using n1 by simp
    have two-le: (2::real) ≤ (real n) powr (c − 1)
    proof −
      have le1: ln 2 ≤ (c − 1) ∗ ln (real n)
      proof −
        have ln 2 = (c − 1) ∗ (ln 2 / (c − 1))
          using cpos by simp
```

```
      also have ... ≤ (c − 1) * ln (real n)
        using elim(1) cpos by (intro mult-left-mono) auto
      finally show ?thesis .
    qed
    have ex: (2::real) ≤ exp ((c − 1) * ln (real n))
    proof −
      have exp (ln (2::real)) ≤ exp ((c − 1) * ln (real n))
        using le1 by (simp only: exp-le-cancel-iff)
      thus ?thesis by simp
    qed
    have (real n) powr (c − 1) = exp ((c − 1) * ln (real n))
      using npos by (simp add: powr-def)
    with ex show ?thesis by simp
  qed
  have key: (real n) powr c = real n * (real n) powr (c − 1)
  proof −
    have (real n) powr c = (real n) powr (1 + (c − 1))
      by simp
    also have ... = (real n) powr 1 * (real n) powr (c − 1)
      by (rule powr-add)
    also have ... = real n * (real n) powr (c − 1)
      by (simp add: powr-one n1)
    finally show ?thesis .
  qed
  have sucle: real (Suc n) ≤ real n * 2
  proof −
    have real (Suc n) = real n + 1
      by simp
    thus ?thesis using n1 by linarith
  qed
  have real (Suc n) ≤ real n * 2
    by (rule sucle)
  also have ... ≤ real n * (real n) powr (c − 1)
    by (rule mult-left-mono[OF two-le]) (simp add: npos)
  also have ... = (real n) powr c
    by (simp add: key)
  finally show ?case .
 qed
qed
```

The slope *2 ln 2* obtained from squaring the schedule exceeds *1*.

```
lemma one-less-two-ln-two: 1 < 2 * ln (2::real)
proof −
  have exp-half-lt: exp (1 / 2 :: real) < 2
  proof (rule power-less-imp-less-base[where n = 2])
    have (exp (1 / 2 :: real)) ^ 2 = exp 1
      by (simp add: power2-eq-square flip: exp-add)
    also have ... ≤ 3
      by (rule exp-le)
```

```
  also have ... < 2 ^ 2
    by simp
  finally show (exp (1 / 2 :: real)) ^ 2 < 2 ^ 2 .
  show 0 ≤ (2::real)
    by simp
qed
have exp (1 / 2 :: real) < exp (ln 2)
  using exp-half-lt by simp
then have 1 / 2 < ln (2::real)
  by (simp only: exp-less-cancel-iff)
then show ?thesis
  by simp
qed
```

10.2 A reusable "the source is the sole pivot" brick

The dual of *unique-shortest-digraph.find-pivots-pivots-capped-singleton-empty-if-params-exceed-vertex-count*: when the capped scan does *not* overflow and the source's own tree slice already holds at least k seen vertices, the singleton FindPivots call returns exactly the source. This is the $P = \{s\}$ half of the source-pivot witness, and it is precisely the regime of the decoupled path family: the cap covers the path, yet the source subtree is large.

```
context unique-shortest-digraph
begin

lemma find-pivots-pivots-capped-singleton-source-if-large-tree:
  assumes no-overflow: card (find-pivots-seen-capped k cap d {s} B) ≤ cap
    and large: k ≤ card (tree-of s ∩ find-pivots-seen-capped k cap d {s} B)
  shows find-pivots-pivots-capped k cap d {s} B = {s}
proof −
  let ?seen = find-pivots-seen-capped k cap d {s} B
  have no-overflow': ¬ card ?seen > cap
    using no-overflow by simp
  have qualifies: {u ∈ {s}. k ≤ card (tree-of u ∩ ?seen)} = {s}
    using large by auto
  show ?thesis
    unfolding find-pivots-pivots-capped-def
    using no-overflow' qualifies by simp
qed
```

The source's shortest-path tree is exactly its reachable set: the source is the head of every shortest walk, hence lies on every shortest path. Combined with the FindPivots brick this turns the $P = \{s\}$ obligation on the path family into a pure cardinality count of the capped scan.

```
lemma tree-path-source-iff-reachable:
  tree-path s v ⟷ reachable s v
proof
  assume tree-path s v
```

```
  thus reachable s v
    by (rule tree-pathD)
next
  assume r: reachable s v
  have shortest-walk s (shortest-path-to v) v
    by (rule shortest-path-to-shortest[OF r])
  then have walk-betw s (shortest-path-to v) v
    unfolding shortest-walk-def simple-walk-betw-def by simp
  then have ne: shortest-path-to v ≠ []
    and hd: hd (shortest-path-to v) = s
    unfolding walk-betw-def by auto
  have mem: s ∈ set (shortest-path-to v)
    using ne hd by (metis hd-in-set)
  show tree-path s v
    by (rule tree-pathI[OF r mem])
qed

lemma tree-of-source-eq:
  tree-of s = {v ∈ V. reachable s v}
  unfolding tree-of-def by (simp add: tree-path-source-iff-reachable)
```

Generic per-round bound on the capped FindPivots scan cost. Each round of *fp-iter-capped-scan-cost* contributes *card* (*outgoing-edges F*) for the current frontier F, and every frontier's outgoing-edge set is a subset of the whole edge set, so each round costs at most *edge-count*. Hence the n-round scan costs at most $n * \mathit{edge\text{-}count}$, uniformly in the schedule arguments. This is the only handle we need on the FindPivots cost: the cheap charged top-level run on the path family pays exactly one such scan per recursion level (all four partition costs being zero), so its whole cost is a sum of these scans plus a single base-case scan.

```
lemma fp-iter-capped-scan-cost-le-rounds:
  fp-iter-capped-scan-cost n cap d F W B ≤ n * edge-count
proof (induction n arbitrary: d F W)
  case 0
  show ?case by simp
next
  case (Suc n)
  have round-le: card (outgoing-edges F) ≤ edge-count
    by (rule edge-count-outgoing-bound)
  have tail-le:
    (if card (W ∪ fp-next d F B) > cap then 0
      else fp-iter-capped-scan-cost n cap (relax-frontier d F) (fp-next d F B)
            (W ∪ fp-next d F B) B) ≤ n * edge-count
  proof (cases card (W ∪ fp-next d F B) > cap)
    case True
    then show ?thesis by simp
  next
    case False
```

```
  then show ?thesis using Suc.IH by simp
 qed
 have fp-iter-capped-scan-cost (Suc n) cap d F W B
    = card (outgoing-edges F)
      + (if card (W ∪ fp-next d F B) > cap then 0
         else fp-iter-capped-scan-cost n cap (relax-frontier d F) (fp-next d F B)
              (W ∪ fp-next d F B) B)
  by (simp add: Let-def)
 also have ... ≤ edge-count + n * edge-count
  using round-le tail-le by linarith
 also have ... = Suc n * edge-count by simp
 finally show ?case .
qed

end
```

10.3 The genuine shortest-path distance on the path family

The unit-weight path P_k has the identity distance *dist 0 v* = *v*: the canonical walk *0* → *1* → ... → *v* is the unique simple walk to *v* (*path-k-simple-walk-unique*) and carries *v* unit edges. This is the keystone distance fact underlying every later path-tree, bound-tree and FindPivots characterisation on the decoupled family.

10.4 A schedule that dominates the path size (non-vacuous regime)

For the genuinely non-vacuous result we run the growing path P_k at a schedule *σ k* chosen so that the FindPivots round count *p* = *sssp-log-one-third-param* (*σ k*) strictly exceeds the *Suc k* path vertices. Such a schedule exists because *sssp-log-one-third-param* diverges (*sssp-log-one-third-param-at-top*). In this regime the top pivots are empty and a charged top-level run exists unconditionally, while *card* (*path-k-V k*) = *Suc k* → ∞.

```
definition path-sched :: nat ⇒ nat where
 path-sched k = (LEAST N. Suc k < sssp-log-one-third-param N)

lemma path-sched-param-gt:
 Suc k < sssp-log-one-third-param (path-sched k)
proof −
 have ∃N. Suc k < sssp-log-one-third-param N
 proof −
  have eventually (λN. Suc (Suc k) ≤ sssp-log-one-third-param N) at-top
   using sssp-log-one-third-param-at-top unfolding filterlim-at-top by blast
  then obtain N where N: Suc (Suc k) ≤ sssp-log-one-third-param N
   by (meson eventually-at-top-linorder order-refl)
  have Suc k < sssp-log-one-third-param N using N by linarith
  then show ?thesis by blast
```

```
  qed
  then show ?thesis
    unfolding path-sched-def by (rule LeastI-ex)
qed

context fixes k :: nat
begin

interpretation pk: finite-weighted-digraph
  path-k-V k path-k-E k path-k-weight k 0
  by (rule path-k-finite-weighted-digraph)

lemma path-k-walk-weight-upt:
  a + j ≤ k ⟹ pk.walk-weight [a..<Suc (a + j)] = real j
proof (induction j arbitrary: a)
  case 0
  have [a..<Suc (a + 0)] = [a] by simp
  then show ?case by simp
next
  case (Suc j)
  have alt: a < k using Suc.prems by simp
  have w1: path-k-weight k a (Suc a) = 1
    using alt by (simp add: path-k-weight-def path-k-E-iff)
  have dec1: [a..<Suc (a + Suc j)] = a # [Suc a..<Suc (a + Suc j)]
    by (simp add: upt-conv-Cons)
  have dec2: [Suc a..<Suc (a + Suc j)] = Suc a # [Suc (Suc a)..<Suc (a + Suc j)]
    by (simp add: upt-conv-Cons)
  have ih: pk.walk-weight [Suc a..<Suc (Suc a + j)] = real j
    using Suc.IH[of Suc a] Suc.prems by simp
  have step: pk.walk-weight (Suc a # [Suc (Suc a)..<Suc (a + Suc j)]) = real j
    using dec2 ih by simp
  have pk.walk-weight [a..<Suc (a + Suc j)]
      = pk.walk-weight (a # Suc a # [Suc (Suc a)..<Suc (a + Suc j)])
    using dec1 dec2 by simp
  also have ... = path-k-weight k a (Suc a)
      + pk.walk-weight (Suc a # [Suc (Suc a)..<Suc (a + Suc j)])
    by simp
  also have ... = 1 + real j
    using w1 step by simp
  finally show ?case by simp
qed

lemma path-k-dist-eq:
  assumes v ≤ k
  shows pk.dist 0 v = real v
proof −
  have canon: pk.simple-walk-betw 0 [0..<Suc v] v
    using assms by (rule path-k-canonical-walk)
  have uniq: {p. pk.simple-walk-betw 0 p v} = {[0..<Suc v]}
```

```
  proof
    show {p. pk.simple-walk-betw 0 p v} ⊆ {[0..<Suc v]}
    proof
      fix p
      assume p ∈ {p. pk.simple-walk-betw 0 p v}
      then have pk.simple-walk-betw 0 p v by simp
      from path-k-simple-walk-unique[OF this assms]
      show p ∈ {[0..<Suc v]} by simp
    qed
    show {[0..<Suc v]} ⊆ {p. pk.simple-walk-betw 0 p v}
      using canon by simp
  qed
  have ww: pk.walk-weight [0..<Suc v] = real v
  proof −
    have hk: 0 + v ≤ k using assms by simp
    show ?thesis
      using path-k-walk-weight-upt[OF hk] by (simp only: add-0-left)
  qed
  have pk.simple-walk-weights 0 v = {real v}
    unfolding pk.simple-walk-weights-def using uniq ww by simp
  then show ?thesis
    unfolding pk.dist-def by simp
qed

interpretation bpi: bounded-reduced-positive-instance
  path-k-V k path-k-E k path-k-weight k 0 1
  by (rule path-k-bounded-instance)
```

The source's shortest-path tree is the whole path: every path vertex is reachable from *0*, and *0* heads every shortest walk. Hence the source-tree slice in the FindPivots count is the entire vertex set.

```
lemma path-k-tree-of-source:
  bpi.tree-of 0 = path-k-V k
proof −
  have bpi.tree-of 0 = {v ∈ path-k-V k. pk.reachable 0 v}
    by (rule bpi.tree-of-source-eq)
  also have ... = path-k-V k
    using path-k-all-reachable by auto
  finally show ?thesis .
qed
```

The capped FindPivots scan stays inside the graph, so its seen set never exceeds the *Suc k* path vertices. This discharges the no-overflow half of the singleton source-pivot brick whenever the cap covers the path, i.e. eventually at the inflated schedule (see the cap-domination theorem below).

```
lemma path-k-find-pivots-seen-card-le:
  card (bpi.find-pivots-seen-capped K cap d {0} Infinity) ≤ Suc k
proof −
  have s0: {0} ⊆ path-k-V k
```

```
    by (simp add: path-k-V-eq)
  have bpi.find-pivots-seen-capped K cap d {0} Infinity ⊆ path-k-V k
    unfolding bpi.find-pivots-seen-capped-def
    by (rule bpi.fp-iter-capped-seen-subset-V[OF s0 s0])
  then have card (bpi.find-pivots-seen-capped K cap d {0} Infinity)
      ≤ card (path-k-V k)
    by (rule card-mono[OF path-k-finite-V])
  then show ?thesis
    by (simp add: path-k-card-V)
qed
```

Empty top pivots when the schedule dominates the path. If the FindPivots round count K already strictly exceeds the *Suc k* path vertices (and the cap covers the path, so the capped scan does not overflow), then no vertex roots a K-large source-tree slice — the whole source tree has only *Suc* $k < K$ vertices — so the capped FindPivots call from $\{0\}$ returns the *empty* pivot set. This is the genuinely *non-vacuous* regime: with empty pivots the top-level partition loop is immediately Done, so a charged top-level run exists unconditionally, with no recourse to the (walled) multi-step band walk.

```
lemma path-k-top-pivots-empty:
  assumes Kk: Suc k < K and cap: Suc k ≤ cap
  shows bpi.find-pivots-pivots-capped K cap d {0} Infinity = {}
proof −
  let ?seen = bpi.find-pivots-seen-capped K cap d {0} Infinity
  have s0: {0} ⊆ path-k-V k by (simp add: path-k-V-eq)
  have card-le: card ?seen ≤ Suc k
    by (rule path-k-find-pivots-seen-card-le)
  have le-cap: card ?seen ≤ cap
    using card-le cap by simp
  have notgt: ¬ card ?seen > cap
    using le-cap by simp
  have seen-sub: ?seen ⊆ path-k-V k
    unfolding bpi.find-pivots-seen-capped-def
    by (rule bpi.fp-iter-capped-seen-subset-V[OF s0 s0])
  have inter-eq: bpi.tree-of 0 ∩ ?seen = ?seen
    using seen-sub path-k-tree-of-source by auto
  have card-inter: card (bpi.tree-of 0 ∩ ?seen) ≤ Suc k
    using inter-eq card-le by simp
  have nonqual: ¬ K ≤ card (bpi.tree-of 0 ∩ ?seen)
    using card-inter Kk by linarith
  have qual-empty: {u ∈ {0}. K ≤ card (bpi.tree-of u ∩ ?seen)} = {}
    using nonqual by auto
  show ?thesis
    unfolding bpi.find-pivots-pivots-capped-def
    using notgt qual-empty by simp
qed
```

Non-vacuous run existence (empty-pivots regime). Whenever the schedule parameter $p =$ *sssp-log-one-third-param N* exceeds the *Suc k* path

vertices, the top pivots are empty (the cap $p \cdot 2^{p \cdot q} \geq p$ covers the path), so a charged top-level cost exists by the immediate-Done route *bpi.charged-direct-insert-top-level-cost-exists-if-top-pivots-empty*. This discharges run-existence unconditionally on the path in the schedule-dominates-graph regime, the regime where the BMSSP top-level call terminates in a single sweep.

```
lemma path-k-charged-cost-exists-if-param-large:
  assumes Kk: Suc k < sssp-log-one-third-param N
  shows ∃ c. bpi.charged-direct-insert-top-level-cost 1 N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  have cap-ge: Suc k ≤ ?cap
    using Kk bmssp-level-cap-ge-level[of ?p ?q ?p] by linarith
  have empty: bpi.find-pivots-pivots-capped ?p ?cap pk.finite-initial-label {0} Infinity
= {}
    by (rule path-k-top-pivots-empty[OF Kk cap-ge])
  show ?thesis
    by (rule bpi.charged-direct-insert-top-level-cost-exists-if-top-pivots-empty[OF
empty])
qed
```

Unconditional run existence at the dominating schedule. Specialising the empty-pivots route at $N = \mathit{path\text{-}sched}\ k$ discharges run-existence on P_k with *no* side condition: at this schedule the FindPivots round count exceeds the path, so a charged top-level cost always exists.

```
lemma path-k-charged-cost-exists-at-path-sched:
  ∃ c. bpi.charged-direct-insert-top-level-cost 1 (path-sched k) c
  by (rule path-k-charged-cost-exists-if-param-large[OF path-sched-param-gt])
```

Unit out-degree and a linear FindPivots scan bound. Each path vertex has at most one out-edge, so a singleton frontier stays a singleton along the capped FindPivots iteration; hence the scan cost is bounded by the round count times the frontier size — here *linearly* in the frontier, with *no* cap factor. This is the per-call ingredient for the genuine $O(|V|)$ cost of the empty-pivots run (the general library bound carries a spurious *cap* factor).

```
lemma path-k-outdegree-le: bpi.outdegree-le 1
proof (unfold bpi.outdegree-le-def, intro ballI)
  fix u assume u ∈ path-k-V k
  have sub: bpi.out-neighbors {u} ⊆ {Suc u}
    unfolding bpi.out-neighbors-def using path-k-E-iff by auto
  have card (bpi.out-neighbors {u}) ≤ card {Suc u}
    by (rule card-mono[OF - sub]) simp
  then show card (bpi.out-neighbors {u}) ≤ 1 by simp
qed

lemma path-k-fp-iter-capped-scan-cost-le-card-frontier:
```

```
  assumes F ⊆ path-k-V k
  shows bpi.fp-iter-capped-scan-cost n cap d F W B ≤ n * card F
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case by simp
next
  case (Suc n)
  let ?F' = bpi.fp-next d F B
  let ?W' = W ∪ ?F'
  have round-le: card (bpi.outgoing-edges F) ≤ card F
    using bpi.card-outgoing-edges-le[OF Suc.prems bpi.bounded-edge-outdegree] by
simp
  have F'-le: card ?F' ≤ card F
    using bpi.card-fp-next-le[OF Suc.prems path-k-outdegree-le] by simp
  have F'-subset: ?F' ⊆ path-k-V k
    by (rule bpi.fp-next-subset-V)
  show ?case
  proof (cases card ?W' > cap)
    case True
    have c: card (bpi.outgoing-edges F) ≤ Suc n * card F
    proof −
      have card F ≤ Suc n * card F by simp
      then show ?thesis using round-le by linarith
    qed
    show ?thesis using True c by (simp add: Let-def)
  next
    case False
    have tail: bpi.fp-iter-capped-scan-cost n cap (bpi.relax-frontier d F) ?F' ?W' B
        ≤ n * card ?F'
      by (rule Suc.IH[OF F'-subset])
    have step: n * card ?F' ≤ n * card F
      using F'-le by (rule mult-le-mono2)
    have c: card (bpi.outgoing-edges F)
        + bpi.fp-iter-capped-scan-cost n cap (bpi.relax-frontier d F) ?F' ?W' B
        ≤ Suc n * card F
    proof −
      have card (bpi.outgoing-edges F)
          + bpi.fp-iter-capped-scan-cost n cap (bpi.relax-frontier d F) ?F' ?W' B
          ≤ card F + n * card F
        using round-le tail step by linarith
      then show ?thesis by (simp add: algebra-simps)
    qed
    show ?thesis using False c by (simp add: Let-def)
  qed
qed
```

Every prefix vertex $v \leq K \leq k$ is a short-tight witness for the K-round FindPivots scan from the source: the canonical walk $0 \to 1 \to \ldots \to v$ is the (unique) shortest walk to v (*path-k-canonical-walk*, *path-k-dist-eq*), hence

successively tight, starts at the complete source, stays below the *Infinity* bound and uses exactly $v \leq K$ edges.

```
lemma path-k-short-tight-witness-prefix:
  assumes vK: v ≤ K and Kk: K ≤ k
  shows bpi.short-tight-witness K pk.finite-initial-label {0} Infinity v
proof −
  have vk: v ≤ k using vK Kk by simp
  have ne: [0..<Suc v] ≠ [] by simp
  have walk: pk.shortest-walk 0 [0..<Suc v] v
  proof −
    have sw: pk.simple-walk-betw 0 [0..<Suc v] v
      by (rule path-k-canonical-walk[OF vk])
    have ww: pk.walk-weight [0..<Suc v] = real v
      using path-k-walk-weight-upt[of 0 v] vk by simp
    have dd: pk.dist 0 v = real v
      by (rule path-k-dist-eq[OF vk])
    show ?thesis
      unfolding pk.shortest-walk-def using sw ww dd by simp
  qed
  have tight: successively pk.tight-edge-step [0..<Suc v]
    by (rule pk.shortest-walk-successively-tight[OF walk])
  have hd0: hd [0..<Suc v] = 0
    using hd-conv-nth[OF ne] by (simp add: nth-upt del: upt-Suc)
  have last-v: last [0..<Suc v] = v
    using last-conv-nth[OF ne] by (simp add: nth-upt del: upt-Suc)
  have dlab: pk.finite-initial-label (hd [0..<Suc v]) = pk.dist 0 (hd [0..<Suc v])
    using hd0 pk.finite-initial-label-source-complete by simp
  have len: length (pk.path-edges [0..<Suc v]) ≤ K
    using bpi.path-edges-length[of [0..<Suc v]] vK by simp
  show ?thesis
    unfolding bpi.short-tight-witness-def
  proof (intro exI conjI)
    show [0..<Suc v] ≠ [] by (rule ne)
    show hd [0..<Suc v] ∈ {0} using hd0 by simp
    show last [0..<Suc v] = v by (rule last-v)
    show pk.finite-initial-label (hd [0..<Suc v]) = pk.dist 0 (hd [0..<Suc v])
      by (rule dlab)
    show successively pk.tight-edge-step [0..<Suc v] by (rule tight)
    show ∀ y∈set [0..<Suc v]. below-bound (pk.dist 0 y) Infinity by simp
    show length (pk.path-edges [0..<Suc v]) ≤ K by (rule len)
  qed
qed
```

The seen *lower* bound (deliverable (G2) for the $P = \{s\}$ obligation): whenever the round count K is at most the path length and the cap covers the path, the K-round capped scan from $\{0\}$ reaches at least K vertices. The prefix $\{0..K\}$ are short-tight witnesses, hence sit inside the completed (uncapped) seen set (*bpi.find-pivots-completes-short-vertices*); the no-overflow regime collapses the capped scan to the uncapped one (*bpi.fp-iter-capped-eq-*

fp-iter-if-final-within-cap).

lemma *path-k-seen-card-ge*:
assumes *Kk*: $K \leq k$ **and** *cover*: $Suc\ k \leq cap$
shows $K \leq card\ (bpi.find\text{-}pivots\text{-}seen\text{-}capped\ K\ cap\ pk.finite\text{-}initial\text{-}label\ \{0\}\ Infinity)$
proof −
let *?d* = *pk.finite-initial-label*
let *?seen-c* = *bpi.find-pivots-seen-capped K cap ?d* $\{0\}$ *Infinity*
let *?seen-u* = *bpi.find-pivots-seen K ?d* $\{0\}$ *Infinity*
have *su*: $\{0\} \subseteq$ *path-k-V k* **by** (*simp add*: *path-k-V-eq*)
have *sound*: *pk.sound-label pk.finite-initial-label*
by (*rule pk.finite-initial-label-sound*[*OF path-k-all-reachable*])
have *reaches*: $\bigwedge x.\ x \in \{0::nat\} \Longrightarrow$ *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *subset-short*: $\{0..K\} \subseteq$ *bpi.find-pivots-short-vertices K ?d* $\{0\}$ *Infinity*
proof
fix *v* **assume** $v \in \{0..K\}$
then have *vK*: $v \leq K$ **by** *simp*
have *vk*: $v \leq k$ **using** *vK Kk* **by** *simp*
have *wit*: *bpi.short-tight-witness K ?d* $\{0\}$ *Infinity v*
by (*rule path-k-short-tight-witness-prefix*[*OF vK Kk*])
have *vV*: *v* ∈ *path-k-V k* **using** *vk* **by** (*simp add*: *path-k-V-eq*)
show *v* ∈ *bpi.find-pivots-short-vertices K ?d* $\{0\}$ *Infinity*
unfolding *bpi.find-pivots-short-vertices-def* **using** *vV wit* **by** *simp*
qed
have *short-sub-seen*: *bpi.find-pivots-short-vertices K ?d* $\{0\}$ *Infinity* ⊆ *?seen-u*
using *bpi.find-pivots-completes-short-vertices*[*OF sound reaches*]
by *blast*
have *subset-seen*: $\{0..K\} \subseteq$ *?seen-u*
using *subset-short short-sub-seen* **by** *blast*
have *seen-u-sub*: *?seen-u* ⊆ *path-k-V k*
unfolding *bpi.find-pivots-seen-def*
by (*rule bpi.fp-iter-seen-subset-V*[*OF su su*])
have *fin-u*: *finite ?seen-u*
using *seen-u-sub path-k-finite-V* **by** (*rule finite-subset*)
have *card-u-ge*: $K + 1 \leq$ *card ?seen-u*
proof −
have *card* $\{0..K\} \leq$ *card ?seen-u*
by (*rule card-mono*[*OF fin-u subset-seen*])
then show *?thesis* **by** *simp*
qed
have *card-c-le-cap*: *card ?seen-c* ≤ *cap*
proof −
have *card ?seen-c* ≤ *Suc k* **by** (*rule path-k-find-pivots-seen-card-le*)
then show *?thesis* **using** *cover* **by** *simp*
qed
have *eq-seen*: *?seen-c* = *?seen-u*
proof −
have *le*: *card* (*bpi.fp-seen* (*bpi.fp-iter-capped K cap ?d* $\{0\}$ $\{0\}$ *Infinity*)) ≤ *cap*

```
    using card-c-le-cap unfolding bpi.find-pivots-seen-capped-def .
  have bpi.fp-iter-capped K cap ?d {0} {0} Infinity
     = bpi.fp-iter K ?d {0} {0} Infinity
    by (rule bpi.fp-iter-capped-eq-fp-iter-if-final-within-cap[OF le])
  then show ?thesis
    unfolding bpi.find-pivots-seen-capped-def bpi.find-pivots-seen-def by simp
qed
show K ≤ card ?seen-c
  using card-u-ge eq-seen by simp
qed
```

A bound-generic, label-generic version of the short-tight prefix witness: any source-complete finite label ($d\ 0 = 0$) makes every prefix vertex $v \leq K \leq k$ a short-tight witness for the K-round capped scan, at any bound covering the path. This frees the seen lower bound from the specific initial label, which is what the level-l child runs need (their FindPivots input label is the previous level's capped label, not *finite-initial-label*).

```
lemma path-k-short-tight-witness-prefix-fin:
  assumes vK: v ≤ K and Kk: K ≤ k and d0: d 0 = 0
  shows bpi.short-tight-witness K d {0} (Fin (real (Suc k))) v
proof −
  have vk: v ≤ k using vK Kk by simp
  have ne: [0..<Suc v] ≠ [] by simp
  have walk: pk.shortest-walk 0 [0..<Suc v] v
  proof −
    have sw: pk.simple-walk-betw 0 [0..<Suc v] v
      by (rule path-k-canonical-walk[OF vk])
    have ww: pk.walk-weight [0..<Suc v] = real v
      using path-k-walk-weight-upt[of 0 v] vk by simp
    have dd: pk.dist 0 v = real v
      by (rule path-k-dist-eq[OF vk])
    show ?thesis
      unfolding pk.shortest-walk-def using sw ww dd by simp
  qed
  have tight: successively pk.tight-edge-step [0..<Suc v]
    by (rule pk.shortest-walk-successively-tight[OF walk])
  have hd0: hd [0..<Suc v] = 0
    using hd-conv-nth[OF ne] by (simp add: nth-upt del: upt-Suc)
  have last-v: last [0..<Suc v] = v
    using last-conv-nth[OF ne] by (simp add: nth-upt del: upt-Suc)
  have dlab: d (hd [0..<Suc v]) = pk.dist 0 (hd [0..<Suc v])
    using hd0 d0 pk.dist-refl-zero[OF pk.source-in-V] by simp
  have len: length (pk.path-edges [0..<Suc v]) ≤ K
    using bpi.path-edges-length[of [0..<Suc v]] vK by simp
  show ?thesis
    unfolding bpi.short-tight-witness-def
  proof (intro exI conjI)
    show [0..<Suc v] ≠ [] by (rule ne)
    show hd [0..<Suc v] ∈ {0} using hd0 by simp
```

```
    show last [0..<Suc v] = v by (rule last-v)
    show d (hd [0..<Suc v]) = pk.dist 0 (hd [0..<Suc v]) by (rule dlab)
    show successively pk.tight-edge-step [0..<Suc v] by (rule tight)
    show ∀ y∈set [0..<Suc v]. below-bound (pk.dist 0 y) (Fin (real (Suc k)))
    proof
      fix y assume y ∈ set [0..<Suc v]
      then have y ≤ k using vk by auto
      then show below-bound (pk.dist 0 y) (Fin (real (Suc k)))
        by (simp add: path-k-dist-eq)
    qed
    show length (pk.path-edges [0..<Suc v]) ≤ K by (rule len)
  qed
qed
```

The bound-/label-generic seen lower bound: at a bound covering the path, the capped K-round scan from $\{0\}$ reaches at least K vertices for any source-complete sound label, provided the cap covers the path.

```
lemma path-k-seen-card-ge-fin:
  assumes Kk: K ≤ k and cover: Suc k ≤ cap and d0: d 0 = 0
    and sound: pk.sound-label d
  shows K ≤ card (bpi.find-pivots-seen-capped K cap d {0} (Fin (real (Suc k))))
proof −
  let ?B = Fin (real (Suc k))
  let ?seen-c = bpi.find-pivots-seen-capped K cap d {0} ?B
  let ?seen-u = bpi.find-pivots-seen K d {0} ?B
  have su: {0} ⊆ path-k-V k by (simp add: path-k-V-eq)
  have reaches: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
    using pk.reachable-refl[OF pk.source-in-V] by simp
  have subset-short: {0..K} ⊆ bpi.find-pivots-short-vertices K d {0} ?B
  proof
    fix v assume v ∈ {0..K}
    then have vK: v ≤ K by simp
    have vk: v ≤ k using vK Kk by simp
    have wit: bpi.short-tight-witness K d {0} ?B v
      using path-k-short-tight-witness-prefix-fin[where K=K and v=v and d=d,
OF vK Kk d0] .
    have vV: v ∈ path-k-V k using vk by (simp add: path-k-V-eq)
    show v ∈ bpi.find-pivots-short-vertices K d {0} ?B
      unfolding bpi.find-pivots-short-vertices-def using vV wit by simp
  qed
  have short-sub-seen: bpi.find-pivots-short-vertices K d {0} ?B ⊆ ?seen-u
    using bpi.find-pivots-completes-short-vertices[OF sound reaches] by blast
  have subset-seen: {0..K} ⊆ ?seen-u using subset-short short-sub-seen by blast
  have seen-u-sub: ?seen-u ⊆ path-k-V k
    unfolding bpi.find-pivots-seen-def by (rule bpi.fp-iter-seen-subset-V[OF su su])
  have fin-u: finite ?seen-u using seen-u-sub path-k-finite-V by (rule finite-subset)
  have card-u-ge: K + 1 ≤ card ?seen-u
  proof −
    have card {0..K} ≤ card ?seen-u by (rule card-mono[OF fin-u subset-seen])
```

```
    then show ?thesis by simp
  qed
  have card-c-le-cap: card ?seen-c ≤ cap
  proof −
    have sub: ?seen-c ⊆ path-k-V k
      unfolding bpi.find-pivots-seen-capped-def
      by (rule bpi.fp-iter-capped-seen-subset-V[OF su su])
    have card ?seen-c ≤ card (path-k-V k) by (rule card-mono[OF path-k-finite-V
sub])
    then show ?thesis using cover by (simp add: path-k-card-V)
  qed
  have eq-seen: ?seen-c = ?seen-u
  proof −
    have le: card (bpi.fp-seen (bpi.fp-iter-capped K cap d {0} {0} ?B)) ≤ cap
      using card-c-le-cap unfolding bpi.find-pivots-seen-capped-def .
    have bpi.fp-iter-capped K cap d {0} {0} ?B = bpi.fp-iter K d {0} {0} ?B
      by (rule bpi.fp-iter-capped-eq-fp-iter-if-final-within-cap[OF le])
    then show ?thesis
      unfolding bpi.find-pivots-seen-capped-def bpi.find-pivots-seen-def by simp
  qed
  show K ≤ card ?seen-c using card-u-ge eq-seen by simp
qed

lemma path-k-source-is-sole-pivot-fin:
  assumes Kk: K ≤ k and cover: Suc k ≤ cap and d0: d 0 = 0
    and sound: pk.sound-label d
  shows bpi.find-pivots-pivots-capped K cap d {0} (Fin (real (Suc k))) = {0}
proof (rule bpi.find-pivots-pivots-capped-singleton-source-if-large-tree)
  let ?B = Fin (real (Suc k))
  show card (bpi.find-pivots-seen-capped K cap d {0} ?B) ≤ cap
  proof −
    have su: {0} ⊆ path-k-V k by (simp add: path-k-V-eq)
    have bpi.find-pivots-seen-capped K cap d {0} ?B ⊆ path-k-V k
      unfolding bpi.find-pivots-seen-capped-def
      by (rule bpi.fp-iter-capped-seen-subset-V[OF su su])
    then have card (bpi.find-pivots-seen-capped K cap d {0} ?B) ≤ Suc k
      using card-mono[OF path-k-finite-V] by (simp add: path-k-card-V)
    then show ?thesis using cover by simp
  qed
  show K ≤ card (bpi.tree-of 0
     ∩ bpi.find-pivots-seen-capped K cap d {0} ?B)
  proof −
    let ?seen-c = bpi.find-pivots-seen-capped K cap d {0} ?B
    have s0: {0} ⊆ path-k-V k by (simp add: path-k-V-eq)
    have seen-sub: ?seen-c ⊆ path-k-V k
      unfolding bpi.find-pivots-seen-capped-def
      by (rule bpi.fp-iter-capped-seen-subset-V[OF s0 s0])
    have inter-eq: bpi.tree-of 0 ∩ ?seen-c = ?seen-c
      using seen-sub path-k-tree-of-source by auto
```

```
    have K ≤ card ?seen-c by (rule path-k-seen-card-ge-fin[OF Kk cover d0 sound])
    then show ?thesis using inter-eq by simp
  qed
qed
```

The source-is-sole-pivot gateway (deliverable (G3), conjunct $P = \{s\}$): under the same two hypotheses, the singleton FindPivots call on the path returns exactly the source. This combines the seen lower bound above (the K qualifying vertices) with the no-overflow upper bound (*path-k-find-pivots-seen-card-le*) through the reusable brick *bpi.find-pivots-pivots-capped-singleton-source-if-large-tree*; the source tree is the whole path (*path-k-tree-of-source*), so the tree-slice intersection is the entire seen set.

```
lemma path-k-source-is-sole-pivot:
  assumes Kk: K ≤ k and cover: Suc k ≤ cap
  shows bpi.find-pivots-pivots-capped K cap pk.finite-initial-label {0} Infinity =
{0}
proof (rule bpi.find-pivots-pivots-capped-singleton-source-if-large-tree)
  show card (bpi.find-pivots-seen-capped K cap pk.finite-initial-label {0} Infinity)
≤ cap
  proof −
    have card (bpi.find-pivots-seen-capped K cap pk.finite-initial-label {0} Infinity)
≤ Suc k
      by (rule path-k-find-pivots-seen-card-le)
    then show ?thesis using cover by simp
  qed
  show K ≤ card (bpi.tree-of 0
      ∩ bpi.find-pivots-seen-capped K cap pk.finite-initial-label {0} Infinity)
  proof −
    let ?seen-c = bpi.find-pivots-seen-capped K cap pk.finite-initial-label {0} Infinity
    have s0: {0} ⊆ path-k-V k by (simp add: path-k-V-eq)
    have seen-sub: ?seen-c ⊆ path-k-V k
      unfolding bpi.find-pivots-seen-capped-def
      by (rule bpi.fp-iter-capped-seen-subset-V[OF s0 s0])
    have inter-eq: bpi.tree-of 0 ∩ ?seen-c = ?seen-c
      using seen-sub path-k-tree-of-source by auto
    have K ≤ card ?seen-c by (rule path-k-seen-card-ge[OF Kk cover])
    then show ?thesis using inter-eq by simp
  qed
qed
```

The remaining structural conjuncts of *strict-tie-breaking-digraph.charged-direct-insert-source-pivot-finishes* on the path family, discharged once and for all. Together with the source-is-sole-pivot gateway above they reduce the predicate to its single charged child-run obligation (conjunct *6*): a full-cover level-$p-1$ run of the path inside the bound *Fin* (*real* (*Suc k*)).

Conjunct *2*: the source's FindPivots label never rises above *0*.

```
lemma path-k-fp-label-source-le-zero:
  bpi.find-pivots-label-capped K cap pk.finite-initial-label {0} Infinity 0 ≤ 0
```

```
proof −
  have bpi.find-pivots-label-capped K cap pk.finite-initial-label {0} Infinity 0
      ≤ pk.finite-initial-label 0
    by (rule bpi.find-pivots-label-capped-le)
  also have pk.finite-initial-label 0 = pk.dist 0 0
    by (rule pk.finite-initial-label-source-complete)
  also have pk.dist 0 0 = 0
    by (rule pk.dist-refl-zero[OF pk.source-in-V])
  finally show ?thesis .
qed
```

Conjunct *5*: the zero-bounded source tree is empty.

```
lemma path-k-bound-tree-source-Fin0-empty:
  pk.bound-tree {0} (Fin 0) = {}
proof −
  have ⋀v. v ∈ pk.bound-tree {0} (Fin 0) ⟹ False
  proof −
    fix v assume v ∈ pk.bound-tree {0} (Fin 0)
    then have rv: pk.reachable 0 v and bb: below-bound (pk.dist 0 v) (Fin 0)
      unfolding pk.bound-tree-def by auto
    have 0 ≤ pk.dist 0 v by (rule pk.dist-nonneg[OF rv])
    moreover have pk.dist 0 v < 0 using bb by simp
    ultimately show False by simp
  qed
  then show ?thesis by blast
qed
```

Conjunct *4*: the FindPivots source precondition holds at any finite bound.

```
lemma path-k-fp-source-pre-full:
  pk.bmssp-pre-full
    (bpi.find-pivots-label-capped K cap pk.finite-initial-label {0} Infinity)
    {0} (Fin B)
proof −
  have allr: ⋀v. v ∈ path-k-V k ⟹ pk.reachable 0 v
    by (rule path-k-all-reachable)
  have sc: pk.finite-initial-label 0 = pk.dist 0 0
    by (rule pk.finite-initial-label-source-complete)
  have pre0: pk.bmssp-pre-full pk.finite-initial-label {0} Infinity
  proof (rule pk.top-bmssp-pre-full)
    show ⋀v. v ∈ path-k-V k ⟹ pk.reachable 0 v by (rule allr)
    show pk.finite-initial-label 0 = pk.dist 0 0 by (rule sc)
  qed
  have sound: pk.sound-label pk.finite-initial-label
    by (rule pk.finite-initial-label-sound[OF allr])
  have reaches: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
    using pk.reachable-refl[OF pk.source-in-V] by simp
  have pre-inf:
    pk.bmssp-pre-full
      (bpi.find-pivots-label-capped K cap pk.finite-initial-label {0} Infinity)
```

```
    {0} Infinity
  by (rule bpi.find-pivots-label-capped-preserves-source-pre[OF sound pre0 reaches])
 show ?thesis
  using pre-inf unfolding pk.bmssp-pre-full-def by auto
qed
```

Conjunct *7*: with the bound *Fin* (*real* (*Suc k*)) chosen to cover the whole path, the source band is the entire path, so its outgoing boundary edges all relax to a value $\leq k < Suc\ k$ and the direct-edge batch is empty.

```
lemma path-k-direct-batch-empty:
 bpi.edge-relaxation-pairs-in-bound (pk.dist 0)
   (bpi.range-tree {0} 0 (Fin (real (Suc k)))) (real (Suc k)) Infinity = []
proof −
 let ?U = bpi.range-tree {0} 0 (Fin (real (Suc k)))
 have setU: set (bpi.edge-list-of (bpi.outgoing-edges ?U)) = bpi.outgoing-edges ?U
  by (rule bpi.edge-list-of-properties(1)[OF bpi.finite-outgoing-edges])
 have no-edge: ⋀e. e ∈ bpi.outgoing-edges ?U ⟹
   ¬ (real (Suc k) ≤ pk.dist 0 (fst e) + path-k-weight k (fst e) (snd e) ∧
      below-bound (pk.dist 0 (fst e) + path-k-weight k (fst e) (snd e)) Infinity)
 proof −
  fix e assume e ∈ bpi.outgoing-edges ?U
  then have edge: e ∈ path-k-E k
   unfolding bpi.outgoing-edges-def by auto
  obtain u v where e: e = (u, v) by (cases e)
  have euv: (u, v) ∈ path-k-E k using edge e by simp
  then have vSuc: v = Suc u and ult: u < k
   by (simp-all add: path-k-E-iff)
  have du: pk.dist 0 u = real u
   using ult by (intro path-k-dist-eq) simp
  have wuv: path-k-weight k u v = 1
   using euv by (simp add: path-k-weight-def)
  have pk.dist 0 u + path-k-weight k u v = real u + 1
   using du wuv by simp
  also have ... < real (Suc k)
   using ult by simp
  finally show ¬ (real (Suc k) ≤ pk.dist 0 (fst e) + path-k-weight k (fst e) (snd
e) ∧
      below-bound (pk.dist 0 (fst e) + path-k-weight k (fst e) (snd e)) Infinity)
   using e by simp
 qed
 show ?thesis
  unfolding bpi.edge-relaxation-pairs-in-bound-def
  using setU no-edge by (auto simp: filter-empty-conv)
qed
```

Milestone B1 (deliverable (G3)), the decoupled charged reduction. Putting conjuncts *1–5* and *7* together, the explicit-assumption top-level builder *bpi.charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes* turns the existence of the single full-cover level-$p-1$ child run (con-

junct *6*) into the existence of a charged top-level cost. This is the schedule-aware reduction of (G3) to the lone charged child-run obligation: once that run is built for the inflated schedule N, the charged top-level cost on P_k exists unconditionally.

```
lemma path-k-charged-cost-exists-if-child:
  assumes pk-le: sssp-log-one-third-param N ≤ k
    and cover: Suc k ≤ bmssp-level-cap (sssp-log-one-third-param N)
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    and child:
      ∃ U-child c-child.
        bpi.charged-direct-insert-costed-bmssp (Suc 0)
          (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param
N))
          (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
          (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))
          (sssp-log-one-third-param N − 1)
          (bpi.find-pivots-label-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))
            pk.finite-initial-label {0} Infinity)
          {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U-child c-child
  shows ∃ c. bpi.charged-direct-insert-top-level-cost 1 N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  let ?d-fp = bpi.find-pivots-label-capped ?p ?cap pk.finite-initial-label {0} Infinity
  let ?P = bpi.find-pivots-pivots-capped ?p ?cap pk.finite-initial-label {0} Infinity
  have P0: ?P = {0} by (rule path-k-source-is-sole-pivot[OF pk-le cover])
  have d-lt: ?d-fp 0 < real (Suc k)
  proof −
    have ?d-fp 0 ≤ 0 by (rule path-k-fp-label-source-le-zero)
    also have (0::real) < real (Suc k) by simp
    finally show ?thesis .
  qed
  have bn: (0::real) ≤ real (Suc k) by simp
  have pre: pk.bmssp-pre-full ?d-fp {0} (Fin (real (Suc k)))
    by (rule path-k-fp-source-pre-full)
  have cap-pos: 0 < ?p * ?cap
    using sssp-log-one-third-param-pos[of N] by (simp add: bmssp-level-cap-def)
  have card-lt: card (pk.bound-tree ?P (Fin 0)) < ?p * ?cap
  proof −
    have pk.bound-tree ?P (Fin 0) = {}
      using P0 path-k-bound-tree-source-Fin0-empty by simp
    then show ?thesis using cap-pos by simp
  qed
```

```
  have batch: bpi.edge-relaxation-pairs-in-bound (pk.dist 0)
      (bpi.range-tree ?P 0 (Fin (real (Suc k)))) (real (Suc k)) Infinity = []
    using P0 path-k-direct-batch-empty by simp
  have main: ∃ c. bpi.charged-direct-insert-top-level-cost (Suc 0) N c
    using child
  proof (elim exE)
    fix U-child c-child
    assume child':
      bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap ?p ?q) ?q ?p
?p ?cap
        (?p − 1) ?d-fp {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k)))
        U-child c-child
    show ?thesis
    proof (rule bpi.charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes)
      show ?P = {0} by (rule P0)
      show ?d-fp 0 < real (Suc k) by (rule d-lt)
      show (0::real) ≤ real (Suc k) by (rule bn)
      show pk.bmssp-pre-full ?d-fp {0} (Fin (real (Suc k))) by (rule pre)
      show card (pk.bound-tree ?P (Fin 0)) < ?p * ?cap by (rule card-lt)
      show bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap ?p ?q)
?q ?p ?p ?cap
          (?p − 1) ?d-fp {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k)))
          U-child c-child
        by (rule child')
      show bpi.edge-relaxation-pairs-in-bound (pk.dist 0)
          (bpi.range-tree ?P 0 (Fin (real (Suc k)))) (real (Suc k)) Infinity = []
        by (rule batch)
    qed
  qed
  then show ?thesis by simp
qed
```

Cover lift ($l \to Suc\ l$). If a full-cover charged run of the whole path exists at level l for the FindPivots label of a source-complete sound input label d, then one exists at level $Suc\ l$ for d: the FindPivots call on the path returns just the source (cap covers the path), one pull separates the source from the band, the level-l cover discharges the band, and the residual loop terminates immediately. Stacking this from the level-1 base reaches the level $p-1$ required by conjunct 6.

```
lemma path-k-cover-lift:
  fixes p q cap l :: nat and d :: nat ⇒ real
  assumes pk-le: p ≤ k and cover: Suc k ≤ cap
    and ppos: 0 < p
    and sound: pk.sound-label d and d0: d 0 = 0
    and pre0: pk.bmssp-pre-full d {0} Infinity
    and child: ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap l
      (bpi.find-pivots-label-capped p cap d {0} (Fin (real (Suc k))))
      {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U c
```

```
  shows ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap (Suc l) d {0} (Fin (real (Suc k)))
      (pk.dist 0) (Fin (real (Suc k))) U c
proof −
  let ?B = Fin (real (Suc k))
  let ?d2 = bpi.find-pivots-label-capped p cap d {0} ?B
  let ?P = bpi.find-pivots-pivots-capped p cap d {0} ?B
  have reaches0: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
    using pk.reachable-refl[OF pk.source-in-V] by simp
  have P0: ?P = {0} by (rule path-k-source-is-sole-pivot-fin[OF pk-le cover d0
sound])
  have sound2: pk.sound-label ?d2
    unfolding bpi.find-pivots-label-capped-def
    by (rule bpi.fp-iter-capped-label-sound[OF sound reaches0])
  have le2: ?d2 0 ≤ d 0 by (rule bpi.find-pivots-label-capped-le)
  have d2-0: ?d2 0 = 0
  proof −
    have pk.dist 0 0 ≤ ?d2 0 using sound2 pk.source-in-V pk.reachable-refl[OF
pk.source-in-V]
      unfolding pk.sound-label-def by blast
    moreover have pk.dist 0 0 = 0 by (rule pk.dist-refl-zero[OF pk.source-in-V])
    ultimately show ?thesis using le2 d0 by simp
  qed
  have pre2: pk.bmssp-pre-full ?d2 {0} ?B
  proof −
    have preB: pk.bmssp-pre-full d {0} ?B
      using pre0 unfolding pk.bmssp-pre-full-def by auto
    show ?thesis
      by (rule bpi.find-pivots-label-capped-preserves-source-pre[OF sound preB
reaches0])
  qed
  obtain U-c c-c where childrun:
    bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q p p cap l
      ?d2 {0} ?B (pk.dist 0) ?B U-c c-c
    using child by blast
  have Mpos: 0 < bmssp-level-cap p q l using ppos by (simp add: bmssp-level-cap-
def)
  have pull: pull-separates (bpi.label-partition-view ?d2 {0}) (bmssp-level-cap p q l)
    (real (Suc k)) {0} (real (Suc k)) (bpi.label-partition-view ?d2 {} :: nat partition-
view)
    unfolding pull-separates-def bpi.label-partition-view-def
    using Mpos d2-0 by simp
  have split: {0} = bpi.split-below ?d2 {0} (real (Suc k))
    using d2-0 by (auto simp: bpi.split-below-def)
  have small: card (pk.bound-tree {0} (Fin 0)) < p ∗ cap
  proof −
    have pk.bound-tree {0} (Fin 0) = {} by (rule path-k-bound-tree-source-Fin0-
empty)
    moreover have 0 < p ∗ cap using ppos cover by simp
```

ultimately show *?thesis* **by** *simp*
qed
have *tail-empty*:
keys-of (*batch-min-update* (*bpi.label-partition-view ?d2* {} :: *nat partition-view*)
(*bpi.edge-relaxation-pairs-in-bound* (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) *?B* @
bpi.edge-relaxation-pairs-between (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) (*real* (*Suc k*)) @
bpi.label-pairs-between ?d2 {*0*} (*real* (*Suc k*)) (*real* (*Suc k*)))) = {}
proof −
have *lower*: *bpi.edge-relaxation-pairs-between* (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) (*real* (*Suc k*)) = []
unfolding *bpi.edge-relaxation-pairs-between-def*
by (*auto simp*: *filter-empty-conv split*: *prod.splits*)
have *src*: *bpi.label-pairs-between ?d2* {*0*} (*real* (*Suc k*)) (*real* (*Suc k*)) = []
unfolding *bpi.label-pairs-between-def* **by** (*simp add*: *filter-empty-conv*)
have *direct*: *bpi.edge-relaxation-pairs-in-bound* (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) *?B* = []
unfolding *bpi.edge-relaxation-pairs-in-bound-def* **by** (*auto simp*: *filter-empty-conv*)
show *?thesis* **using** *lower src direct* **unfolding** *batch-min-update-def* **by** *simp*
qed
have *loop*:
∃ *betas bs charged-Us child-outputs U c child-costs*.
bpi.charged-direct-insert-costed-partition-loop-state (*Suc 0*)
(*bmssp-level-cap p q*) *q p p cap l ?d2* {*0*} *?B* (*pk.dist 0*)
(*bpi.label-partition-view ?d2* {*0*}) *0 betas bs ?B charged-Us child-outputs U c child-costs*
by (*rule bpi.charged-loop-single-step-then-done-exists-dist*
[**where** *Bmax* = *real* (*Suc k*) **and** *beta* = *real* (*Suc k*) **and** *b* = *real* (*Suc k*)
and *S-pull* = {*0*} **and** *D-pull* = *bpi.label-partition-view ?d2* {} :: *nat partition-view*
and *d-child* = *pk.dist 0* **and** *U-child* = *U-c* **and** *c-child* = *c-c*])
(*use pull pre2 split reaches0 small childrun tail-empty* **in** *simp-all*)
then obtain *betas bs charged-Us child-outputs U c child-costs* **where** *looprun*:
bpi.charged-direct-insert-costed-partition-loop-state (*Suc 0*)
(*bmssp-level-cap p q*) *q p p cap l ?d2* {*0*} *?B* (*pk.dist 0*)
(*bpi.label-partition-view ?d2* {*0*}) *0 betas bs ?B charged-Us child-outputs U c child-costs*
by *blast*
let *?U* = *U* ∪ {*v* ∈ *pk.bound-tree* {*0*} *?B*. *?d2 v* = *pk.dist 0 v*}
have *run*: *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*) (*bmssp-level-cap p q*) *q p p cap*
(*Suc l*) *d* {*0*} *?B* (*pk.dist 0*) *?B ?U*
(*bpi.fp-iter-capped-scan-cost p cap d* {*0*} {*0*} *?B* + *0* + *c*)
by (*rule bpi.charged-direct-insert-costed-bmssp-step-with-insert-cost*
[**where** *D* = *bpi.label-partition-view ?d2* {*0*} **and** *c-insert* = *0*])
(*use looprun P0* **in** ‹*simp-all add*: *bpi.partition-initial-insert-cost-bound-def pk.complete-on-def*›)

```
  show ?thesis using run by blast
qed
```

Level chain. Stacking the cover lift from a level-1 base, a whole-path cover exists at every level $l \geq 1$ for any source-complete sound input label. Specialising at $l = p - 1$ supplies conjunct 6 of *strict-tie-breaking-digraph.charged-direct-insert-source-pivot-finishes* unconditionally once the level-1 base is available.

```
lemma path-k-cover-ge1:
  fixes p q cap :: nat
  assumes pk-le: p ≤ k and cover: Suc k ≤ cap and ppos: 0 < p
    and base1: ⋀d. pk.sound-label d ⟹ d 0 = 0 ⟹ pk.bmssp-pre-full d {0}
Infinity
        ⟹ ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
            (bmssp-level-cap p q) q p p cap (Suc 0) d {0} (Fin (real (Suc k)))
            (pk.dist 0) (Fin (real (Suc k))) U c
  shows ⋀d. pk.sound-label d ⟹ d 0 = 0 ⟹ pk.bmssp-pre-full d {0} Infinity
⟹
      ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q p
p
        cap (Suc l) d {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U c
proof (induct l)
  case 0
  show ?case using base1[OF 0(1) 0(2) 0(3)] .
next
  case (Suc l)
  let ?d2 = bpi.find-pivots-label-capped p cap d {0} (Fin (real (Suc k)))
  have reaches0: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
    using pk.reachable-refl[OF pk.source-in-V] by simp
  have sound2: pk.sound-label ?d2
    unfolding bpi.find-pivots-label-capped-def
    by (rule bpi.fp-iter-capped-label-sound[OF Suc(2) reaches0])
  have d2-0: ?d2 0 = 0
  proof −
    have pk.dist 0 0 ≤ ?d2 0 using sound2 pk.source-in-V pk.reachable-refl[OF
pk.source-in-V]
      unfolding pk.sound-label-def by blast
    moreover have pk.dist 0 0 = 0 by (rule pk.dist-refl-zero[OF pk.source-in-V])
    ultimately show ?thesis
     using bpi.find-pivots-label-capped-le[of p cap d {0} Fin (real (Suc k)) 0] Suc(3)
by simp
  qed
  have pre2: pk.bmssp-pre-full ?d2 {0} Infinity
  proof (rule pk.top-bmssp-pre-full)
    show ⋀v. v ∈ path-k-V k ⟹ pk.reachable 0 v by (rule path-k-all-reachable)
    show ?d2 0 = pk.dist 0 0 using d2-0 pk.dist-refl-zero[OF pk.source-in-V] by
simp
  qed
  have child: ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
```

```
    (bmssp-level-cap p q) q p p cap (Suc l) ?d2 {0} (Fin (real (Suc k)))
    (pk.dist 0) (Fin (real (Suc k))) U c
   by (rule Suc(1)[OF sound2 d2-0 pre2])
  show ?case
   by (rule path-k-cover-lift[OF pk-le cover ppos Suc(2) Suc(3) Suc(4) child])
qed
```

The path band tree. Every vertex $a \le v \le k$ lies on the unique shortest walk from the source, so the source band starting at a below the finite bound $Fin\ r$ is exactly the prefix $\{a..k\}$ truncated by r.

```
lemma path-k-through-iff:
  assumes v ≤ k
  shows pk.through {a} v ⟷ a ≤ v
proof
  assume pk.through {a} v
  then obtain pp where sw: pk.shortest-walk 0 pp v and au: a ∈ set pp
    unfolding pk.through-def by auto
  have pk.simple-walk-betw 0 pp v using sw unfolding pk.shortest-walk-def by
simp
  then have pp = [0..<Suc v] by (rule path-k-simple-walk-unique[OF - assms])
  then show a ≤ v using au by auto
next
  assume a ≤ v
  then have aset: a ∈ set [0..<Suc v] by auto
  have ww: pk.walk-weight [0..<Suc v] = real v
    using path-k-walk-weight-upt[of 0 v] assms by simp
  have sw: pk.shortest-walk 0 [0..<Suc v] v
    unfolding pk.shortest-walk-def
    using path-k-canonical-walk[OF assms] ww path-k-dist-eq[OF assms] by simp
  show pk.through {a} v unfolding pk.through-def using sw aset by auto
qed

lemma path-k-bound-tree-band:
  pk.bound-tree {a} (Fin r) = {v. a ≤ v ∧ v ≤ k ∧ real v < r}
proof (intro set-eqI iffI)
  fix v assume v ∈ pk.bound-tree {a} (Fin r)
  then have vV: v ∈ path-k-V k and thr: pk.through {a} v
    and lt: pk.dist 0 v < r
    unfolding pk.bound-tree-def by auto
  have vk: v ≤ k using vV by (simp add: path-k-V-eq)
  show v ∈ {v. a ≤ v ∧ v ≤ k ∧ real v < r}
    using thr vk lt path-k-through-iff[OF vk] path-k-dist-eq[OF vk] by simp
next
  fix v assume v ∈ {v. a ≤ v ∧ v ≤ k ∧ real v < r}
  then have av: a ≤ v and vk: v ≤ k and lt: real v < r by auto
  show v ∈ pk.bound-tree {a} (Fin r)
    unfolding pk.bound-tree-def
    using av vk lt path-k-through-iff[OF vk] path-k-dist-eq[OF vk]
      path-k-all-reachable by (simp add: path-k-V-eq)
```

qed

The base-case order on a path source band $\{a\}$ below *Suc k* is just the increasing run $[a, \ldots, k]$: distances equal indices, so the unique distinct distance-sorted listing of $\{a..k\}$ is $[a..<Suc\ k]$.

lemma *path-k-base-order*:
shows *pk.base-case-order a* (*Fin* (*real* (*Suc k*))) = [*a..<Suc k*]
proof −
have *set-xs*: *set* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*)))) = {*a..k*}
proof −
have *set* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*))))
= {*v*. *a* ≤ *v* ∧ *v* ≤ *k* ∧ *real v* < *real* (*Suc k*)}
by (*simp add*: *pk.base-case-order-set path-k-bound-tree-band*)
also have ... = {*a..k*} **by** *auto*
finally show *?thesis* **.**
qed
have *dist-xs*: *distinct* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*))))
by (*rule pk.base-case-order-distinct*)
have *sorted-xs*: *sorted* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*))))
proof −
have *sorted-wrt* (≤) (*pk.base-case-order a* (*Fin* (*real* (*Suc k*))))
proof (*rule sorted-wrt-mono-rel*[*OF - pk.base-case-order-sorted*])
fix *u v*
assume *u* ∈ *set* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*))))
and *v* ∈ *set* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*))))
then have *u* ≤ *k* **and** *v* ≤ *k* **using** *set-xs* **by** *auto*
moreover assume *pk.dist 0 u* ≤ *pk.dist 0 v*
ultimately show *u* ≤ *v* **by** (*simp add*: *path-k-dist-eq*)
qed
then show *?thesis* **by** *simp*
qed
show *?thesis*
proof (*rule sorted-distinct-set-unique*[*OF sorted-xs dist-xs*, *of* [*a..<Suc k*]])
show *set* (*pk.base-case-order a* (*Fin* (*real* (*Suc k*)))) = *set* [*a..<Suc k*]
by (*simp only*: *set-xs set-upt atLeastLessThanSuc-atLeastAtMost*)
show *sorted* [*a..<Suc k*] **by** (*rule sorted-upt*)
show *distinct* [*a..<Suc k*] **by** (*rule distinct-upt*)
qed
qed

One base step. A level-0 base case on a singleton path band $\{a\}$ truncated just above a fully settles a: the bounded tree is $\{a\}$, of card *1* ≤ p, so the base call keeps it whole. Witnesses are given explicitly (the truncating label, returned bound, vertex set) to avoid any search in the large path simpset.

lemma *path-k-base-step*:
fixes *a p q cap* :: *nat* **and** *d* :: *nat* ⇒ *real*
assumes *ak*: *a* ≤ *k* **and** *ppos*: *0* < *p*
shows ∃ *d′ c*. *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*)

```
    (bmssp-level-cap p q) q p p cap 0 d {a} (Fin (real (Suc a)))
    d' (Fin (real (Suc a))) {a} c
proof −
  have bt: pk.bound-tree {a} (Fin (real (Suc a))) = {a}
  proof −
    have pk.bound-tree {a} (Fin (real (Suc a)))
       = {v. a ≤ v ∧ v ≤ k ∧ real v < real (Suc a)}
     by (rule path-k-bound-tree-band)
    also have ... = {a} using ak by (auto simp only: of-nat-less-iff Suc-le-eq
atMost-iff)
    finally show ?thesis .
  qed
  have small: card (pk.bound-tree {a} (Fin (real (Suc a)))) ≤ p
    using bt ppos by simp
  have len: length (pk.base-case-order a (Fin (real (Suc a)))) ≤ p
  proof −
    have set (pk.base-case-order a (Fin (real (Suc a)))) = {a}
      using pk.base-case-order-set[of a Fin (real (Suc a))] bt by simp
    moreover have distinct (pk.base-case-order a (Fin (real (Suc a))))
      by (rule pk.base-case-order-distinct)
    ultimately have length (pk.base-case-order a (Fin (real (Suc a)))) = 1
      by (simp add: distinct-card[symmetric])
    then show ?thesis using ppos by simp
  qed
  have U: pk.base-case-vertices p a (Fin (real (Suc a))) = {a}
    using pk.base-case-success[OF len] bt by simp
  have B': pk.base-case-bound p a (Fin (real (Suc a))) = Fin (real (Suc a))
    using len by (simp add: pk.base-case-bound-def)
  let ?d' = λv. if v ∈ pk.base-case-vertices p a (Fin (real (Suc a)))
      then pk.dist 0 v else d v
  have run: bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q
p p
      cap 0 d {a} (Fin (real (Suc a))) ?d' (Fin (real (Suc a))) {a}
      (bpi.base-case-scan-cost (Suc 0) p a (Fin (real (Suc a))))
      using bpi.Charged-Direct-Insert-Base[OF refl, where Δ=Suc 0 and M-
of=bmssp-level-cap p q
       and t=q and h=p and k=p and cap=cap and d=d and x=a and B=Fin
(real (Suc a))] U B' by simp
  show ?thesis by (intro exI[of - ?d'] exI) (rule run)
qed
```

Top base step. A level-0 base case on the whole source band $\{0\}$ below $Fin\ (real\ (Suc\ k))$ settles exactly the prefix $\{0..<p\}$ and advances the returned bound to $Fin\ (real\ p)$: the path order is $[0..<Suc\ k]$, the capped scan stops at the p-th vertex p, and only strictly-smaller vertices survive. This is the per-block brick that the multi-step loop chains.

Singleton frontier pull. A one-element bucket $\{a\}$ with label view d is pulled whole (fits under the fanout $0 < p$), leaving the empty view; the separator threshold is the loop bound $Bmax$. Each multi-step block iteration

uses this pull.

lemma *path-k-pull-singleton*:
fixes *a p* :: *nat* **and** *Bmax* :: *real* **and** *d* :: *nat* ⇒ *real*
assumes *ppos*: *0* < *p*
shows *pull-separates* (*bpi.label-partition-view d* {*a*}) *p Bmax* {*a*} *Bmax*
(*bpi.label-partition-view d* {} :: *nat partition-view*)
unfolding *pull-separates-def bpi.label-partition-view-def* **using** *ppos* **by** *simp*

Block range tree. The slice of the source tree at heights $[a,\ a{+}p)$ is the block $\{a..{<}a{+}p\}$; this is the charged child the loop discharges at each step.

lemma *path-k-range-block*:
assumes *apk*: *a* + *p* ≤ *k*
shows *bpi.range-tree* {*a*} (*real a*) (*Fin* (*real* (*a* + *p*))) = {*a*..<*a*+*p*}
proof −
have *bpi.range-tree* {*a*} (*real a*) (*Fin* (*real* (*a* + *p*)))
= {*v* ∈ *pk.bound-tree* {*a*} (*Fin* (*real* (*a* + *p*))). *real a* ≤ *pk.dist 0 v*}
unfolding *bpi.range-tree-def bpi.bound-tree-eq-tree-set* **by** *auto*
also have *pk.bound-tree* {*a*} (*Fin* (*real* (*a* + *p*))) = {*v*. *a* ≤ *v* ∧ *v* ≤ *k* ∧ *real v* < *real* (*a* + *p*)}
by (*rule path-k-bound-tree-band*)
also have {*v* ∈ {*v*. *a* ≤ *v* ∧ *v* ≤ *k* ∧ *real v* < *real* (*a* + *p*)}. *real a* ≤ *pk.dist 0 v*} = {*a*..<*a*+*p*}
using *apk* **by** (*auto simp*: *path-k-dist-eq*)
finally show *?thesis* **.**
qed

Source-pivot range slice. Read off the source tree $\{0\}$ the path range $[a,\ b)$ is exactly the integer block $\{a..{<}b\}$ whenever $b \le \mathit{Suc}\ k$. Unlike *path-k-range-block* (which slices from the band start $\{a\}$), the pivot here is the *global* source $\{0\}$ that the top-level partition loop carries, which is what the loop's relaxation batches *bpi.range-tree* $\{0\}$ *a B* use.

lemma *path-k-range-zero*:
assumes *bk*: *b* ≤ *Suc k*
shows *bpi.range-tree* {*0*} (*real a*) (*Fin* (*real b*)) = {*a*..<*b*}
proof −
have *bpi.range-tree* {*0*} (*real a*) (*Fin* (*real b*))
= {*v* ∈ *pk.bound-tree* {*0*} (*Fin* (*real b*)). *real a* ≤ *pk.dist 0 v*}
unfolding *bpi.range-tree-def bpi.bound-tree-eq-tree-set* **by** *auto*
also have *pk.bound-tree* {*0*} (*Fin* (*real b*)) = {*v*. *0* ≤ *v* ∧ *v* ≤ *k* ∧ *real v* < *real b*}
by (*rule path-k-bound-tree-band*)
also have {*v* ∈ {*v*. *0* ≤ *v* ∧ *v* ≤ *k* ∧ *real v* < *real b*}. *real a* ≤ *pk.dist 0 v*} = {*a*..<*b*}
using *bk* **by** (*auto simp*: *path-k-dist-eq*)
finally show *?thesis* **.**
qed

The source-pivot pull on the path. Because the source carries the genuine zero label, the loop's threshold pull *bpi.split-below d* $\{0\}$ *beta* of

the sole pivot set $\{0\}$ is exactly $\{0\}$ below a positive threshold and empty otherwise. This pins down the only pivot the top-level loop can ever pull.

```
lemma path-k-split-below-zero:
  assumes d0: d (0::nat) = 0
  shows bpi.split-below d {0} beta = (if 0 < beta then {0} else {})
  using d0 unfolding bpi.split-below-def by auto
```

Pivot settledness at the source band (the B2 cost brick). At the loop's initial band $a = 0$ the already-settled prefix *pk.bound-tree* $\{0\}$ (*Fin* 0) is empty, so the source pull is trivially settled, and lands inside its own range slice $\{0..<b\}$ whenever the child advanced the bound ($0 < b \leq$ *Suc* k). This is the per-step ingredient of the no-stranding invariant of *bpi.charged-direct-insert-costed-partition-loop-state-cost-local-budget-if-settled*, specialised to the unit-out-degree path where the loop runs at band *0*.

```
lemma path-k-pull-settled-at-zero:
  bpi.split-below d {0} beta ∩ pk.bound-tree {0} (Fin 0) = {}
  using path-k-bound-tree-source-Fin0-empty by simp

lemma path-k-pull-in-first-slice:
  assumes bpos: 0 < b and bk: b ≤ Suc k
  shows bpi.split-below d {0} beta ⊆ bpi.range-tree {0} (real 0) (Fin (real b))
proof −
  have sub0: bpi.split-below d {0} beta ⊆ {0}
    unfolding bpi.split-below-def by auto
  have bpi.range-tree {0} (real 0) (Fin (real b)) = {0..<b}
    by (rule path-k-range-zero[OF bk])
  then have {0} ⊆ bpi.range-tree {0} (real 0) (Fin (real b))
    using bpos by auto
  with sub0 show ?thesis by blast
qed
```

The relaxation never re-creates the source. Every relaxation target on the path is the head *Suc* u of a forward edge (u, *Suc* u), hence strictly positive. So a loop step's batch keys avoid *0*: once the sole pivot *0* is pulled out of the partition it is never pushed back. This is the frontier-monotonicity fact behind pivot settledness on the unit-out-degree path — the loop can advance the frontier but can never return it to the source band.

```
lemma path-k-relax-in-bound-keys-pos:
  0 ∉ fst ` set (bpi.edge-relaxation-pairs-in-bound d U L B')
proof −
  have setU: set (bpi.edge-list-of (bpi.outgoing-edges U)) = bpi.outgoing-edges U
    by (rule bpi.edge-list-of-properties(1)[OF bpi.finite-outgoing-edges])
  have tgt-pos: ⋀u v. (u, v) ∈ bpi.outgoing-edges U ⟹ 0 < v
  proof −
    fix u v assume (u, v) ∈ bpi.outgoing-edges U
    then have (u, v) ∈ path-k-E k
      unfolding bpi.outgoing-edges-def by auto
```

```
    then have v = Suc u by (simp add: path-k-E-iff)
    then show 0 < v by simp
  qed
  show ?thesis
  proof
    assume 0 ∈ fst ‘ set (bpi.edge-relaxation-pairs-in-bound d U L B′)
    then obtain val where
      (0, val) ∈ set (bpi.edge-relaxation-pairs-in-bound d U L B′)
      by auto
    then obtain u v where uv: (u, v) ∈ bpi.outgoing-edges U and v0: v = 0
      unfolding bpi.edge-relaxation-pairs-in-bound-def
      using setU by auto
    from tgt-pos[OF uv] v0 show False by simp
  qed
qed
```

The same frontier-positivity for the *between* relaxation batch: every pushed key is an edge target $Suc\ u > 0$.

```
lemma path-k-relax-between-keys-pos:
  0 ∉ fst ‘ set (bpi.edge-relaxation-pairs-between d U L H)
proof −
  have setU: set (bpi.edge-list-of (bpi.outgoing-edges U)) = bpi.outgoing-edges U
    by (rule bpi.edge-list-of-properties(1)[OF bpi.finite-outgoing-edges])
  have tgt-pos: ⋀u v. (u, v) ∈ bpi.outgoing-edges U ⟹ 0 < v
  proof −
    fix u v assume (u, v) ∈ bpi.outgoing-edges U
    then have (u, v) ∈ path-k-E k
      unfolding bpi.outgoing-edges-def by auto
    then have v = Suc u by (simp add: path-k-E-iff)
    then show 0 < v by simp
  qed
  show ?thesis
  proof
    assume 0 ∈ fst ‘ set (bpi.edge-relaxation-pairs-between d U L H)
    then obtain val where
      (0, val) ∈ set (bpi.edge-relaxation-pairs-between d U L H)
      by auto
    then obtain u v where uv: (u, v) ∈ bpi.outgoing-edges U and v0: v = 0
      unfolding bpi.edge-relaxation-pairs-between-def
      using setU by auto
    from tgt-pos[OF uv] v0 show False by simp
  qed
qed
```

Block frontier. Relaxing the block $\{a..<a+p\}$ across its single boundary edge $(a+p-1,\ a+p)$ pushes exactly the next frontier vertex $a+p$; the lower and source batches are empty. Hence the partition advances $\{a\} \mapsto \{a+p\}$.

```
lemma path-k-block-next-keys:
  assumes apk: a + p ≤ k and ppos: 0 < p
```

shows *keys-of* (*batch-min-update* (*bpi.label-partition-view* (*pk.dist 0*) {} :: *nat partition-view*)
(*bpi.edge-relaxation-pairs-in-bound* (*pk.dist 0*)
(*bpi.range-tree* {*a*} (*real a*) (*Fin* (*real* (*a*+*p*)))) (*real* (*a*+*p*)) (*Fin* (*real* (*Suc k*))) @
bpi.edge-relaxation-pairs-between (*pk.dist 0*)
(*bpi.range-tree* {*a*} (*real a*) (*Fin* (*real* (*a*+*p*)))) (*real* (*a*+*p*)) (*real* (*a*+*p*)) @
bpi.label-pairs-between (*pk.dist 0*) {*a*} (*real* (*a*+*p*)) (*real* (*a*+*p*)))) = {*a*+*p*}
proof −
let *?U* = *bpi.range-tree* {*a*} (*real a*) (*Fin* (*real* (*a*+*p*)))
have *outset*: *set* (*bpi.edge-list-of* (*bpi.outgoing-edges ?U*)) = *bpi.outgoing-edges ?U*
by (*rule bpi.edge-list-of-properties*(*1*)[*OF bpi.finite-outgoing-edges*])
have *Ueq*: *?U* = {*a*..<*a*+*p*} **by** (*rule path-k-range-block*[*OF apk*])
have *direct-fst*:
fst ‘ *set* (*bpi.edge-relaxation-pairs-in-bound* (*pk.dist 0*) *?U* (*real* (*a*+*p*)) (*Fin* (*real* (*Suc k*)))) = {*a*+*p*}
proof −
have *set* (*bpi.edge-relaxation-pairs-in-bound* (*pk.dist 0*) *?U* (*real* (*a*+*p*)) (*Fin* (*real* (*Suc k*))))
= (λ(*u*,*v*). (*v*, *pk.dist 0 u* + *path-k-weight k u v*)) ‘
{*e* ∈ *bpi.outgoing-edges ?U*. *real* (*a*+*p*) ≤ *pk.dist 0* (*fst e*) + *path-k-weight k* (*fst e*) (*snd e*)
∧ *pk.dist 0* (*fst e*) + *path-k-weight k* (*fst e*) (*snd e*) < *real* (*Suc k*)}
unfolding *bpi.edge-relaxation-pairs-in-bound-def* **using** *outset*
by (*auto split*: *prod.splits*)
moreover have {*e* ∈ *bpi.outgoing-edges ?U*. *real* (*a*+*p*) ≤ *pk.dist 0* (*fst e*) + *path-k-weight k* (*fst e*) (*snd e*)
∧ *pk.dist 0* (*fst e*) + *path-k-weight k* (*fst e*) (*snd e*) < *real* (*Suc k*)} = {(*a*+*p*−*1*, *a*+*p*)}
using *apk ppos Ueq*
by (*auto simp*: *bpi.outgoing-edges-def path-k-E-iff path-k-weight-def path-k-dist-eq of-nat-less-iff*)
ultimately show *?thesis* **using** *ppos* **by** *auto*
qed
have *lower*: *bpi.edge-relaxation-pairs-between* (*pk.dist 0*) *?U* (*real* (*a*+*p*)) (*real* (*a*+*p*)) = []
unfolding *bpi.edge-relaxation-pairs-between-def* **by** (*auto simp*: *filter-empty-conv*)
have *src*: *bpi.label-pairs-between* (*pk.dist 0*) {*a*} (*real* (*a*+*p*)) (*real* (*a*+*p*)) = []
unfolding *bpi.label-pairs-between-def* **by** (*simp add*: *filter-empty-conv*)
show *?thesis*
using *direct-fst lower src* **by** (*simp add*: *batch-min-update-keys image-Un*)
qed

lemma *path-k-base-top*:
fixes *p q cap* :: *nat* **and** *d* :: *nat* ⇒ *real*
assumes *pk-le*: *p* ≤ *k* **and** *ppos*: *0* < *p*
shows *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*)

```
    (bmssp-level-cap p q) q p p cap 0 d {0} (Fin (real (Suc k)))
    (λv. if v ∈ pk.base-case-vertices p 0 (Fin (real (Suc k))) then pk.dist 0 v else
d v)
    (Fin (real p)) {0..<p}
    (bpi.base-case-scan-cost (Suc 0) p 0 (Fin (real (Suc k))))
proof −
  let ?xs = pk.base-case-order 0 (Fin (real (Suc k)))
  have order: ?xs = [0..<Suc k] by (rule path-k-base-order)
  have len: length ?xs = Suc k using order by simp
  have not-le: ¬ length ?xs ≤ p using len pk-le by simp
  have nthp: ?xs ! p = p using order pk-le by (simp add: nth-append)
  have distp: pk.dist 0 (?xs ! p) = real p
    using nthp pk-le by (simp add: path-k-dist-eq)
  have vert: pk.base-case-vertices p 0 (Fin (real (Suc k))) = {0..<p}
  proof −
    have take-eq: take (Suc p) ?xs = [0..<Suc p]
      using order pk-le by (simp add: take-upt del: upt-Suc)
    have setp: set (take (Suc p) ?xs) = {0..p}
      using take-eq by auto
    have dvals: ⋀v. v ≤ p ⟹ pk.dist 0 v = real v
      using pk-le by (intro path-k-dist-eq) simp
    have pk.base-case-vertices p 0 (Fin (real (Suc k)))
        = {v ∈ set (take (Suc p) ?xs). pk.dist 0 v < pk.dist 0 (?xs ! p)}
      using not-le unfolding pk.base-case-vertices-def by (simp add: Let-def)
    also have ... = {v ∈ {0..p}. real v < real p}
      using setp distp dvals by (auto simp: atLeastAtMost-iff)
    also have ... = {0..<p}
      by (auto simp only: of-nat-less-iff atLeastLessThan-iff atLeastAtMost-iff)
    finally show ?thesis .
  qed
  have bnd: pk.base-case-bound p 0 (Fin (real (Suc k))) = Fin (real p)
    using not-le distp unfolding pk.base-case-bound-def by (simp add: Let-def)
  show ?thesis
    using bpi.Charged-Direct-Insert-Base[OF refl, where Δ=Suc 0
      and M-of=bmssp-level-cap p q and t=q and h=p and k=p and cap=cap
      and d=d and x=0 and B=Fin (real (Suc k))]
    by (simp only: vert bnd)
qed
```

Generic block base step. From any band start a with $a + p \le k$, the level-0 base call on $\{a\}$ below the full bound settles exactly the block $\{a..<a+p\}$ and advances the returned bound to *Fin* (*real* $(a + p)$). Iterating this brick over $a = 0,\ p,\ 2p,\ \ldots$ is the multi-step frontier walk.

```
lemma path-k-base-block:
  fixes a p q cap :: nat and d :: nat ⇒ real
  assumes apk: a + p ≤ k and ppos: 0 < p
  shows bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap 0 d {a} (Fin (real (Suc k)))
      (λv. if v ∈ pk.base-case-vertices p a (Fin (real (Suc k))) then pk.dist 0 v else d
```

```
v)
      (Fin (real (a + p))) {a..<a+p}
      (bpi.base-case-scan-cost (Suc 0) p a (Fin (real (Suc k))))
proof −
  let ?xs = pk.base-case-order a (Fin (real (Suc k)))
  have ak: a ≤ k using apk by simp
  have order: ?xs = [a..<Suc k] by (rule path-k-base-order)
  have len: length ?xs = Suc k − a using order ak by simp
  have not-le: ¬ length ?xs ≤ p using len apk by simp
  have nthp: ?xs ! p = a + p using order apk by (auto simp add: nth-append)
  have distp: pk.dist 0 (?xs ! p) = real (a + p)
    using nthp apk by (simp add: path-k-dist-eq)
  have vert: pk.base-case-vertices p a (Fin (real (Suc k))) = {a..<a+p}
  proof −
    have take-eq: take (Suc p) ?xs = [a..<Suc (a + p)]
      using order apk by (simp add: take-upt del: upt-Suc)
    have setp: set (take (Suc p) ?xs) = {a..a+p}
      using take-eq by auto
    have dvals: ⋀v. v ≤ a + p ⟹ pk.dist 0 v = real v
      using apk by (intro path-k-dist-eq) simp
    have pk.base-case-vertices p a (Fin (real (Suc k)))
        = {v ∈ set (take (Suc p) ?xs). pk.dist 0 v < pk.dist 0 (?xs ! p)}
      using not-le unfolding pk.base-case-vertices-def by (simp add: Let-def)
    also have ... = {v ∈ {a..a+p}. real v < real (a + p)}
      using setp distp dvals by (auto simp: atLeastAtMost-iff)
    also have ... = {a..<a+p}
      by (auto simp only: of-nat-less-iff atLeastLessThan-iff atLeastAtMost-iff)
    finally show ?thesis .
  qed
  have bnd: pk.base-case-bound p a (Fin (real (Suc k))) = Fin (real (a + p))
    using not-le distp unfolding pk.base-case-bound-def by (simp add: Let-def)
  show ?thesis
    using bpi.Charged-Direct-Insert-Base[OF refl, where Δ=Suc 0
        and M-of=bmssp-level-cap p q and t=q and h=p and k=p and cap=cap
        and d=d and x=a and B=Fin (real (Suc k))]
    by (simp only: vert bnd)
qed
```

Terminal block. When the residual band $\{a..k\}$ fits within p, the base call below the full bound covers it whole and keeps the bound, settling the loop.

```
lemma path-k-base-last:
  fixes a p q cap :: nat and d :: nat ⇒ real
  assumes ak: a ≤ k and ple: Suc k − a ≤ p
  shows bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap 0 d {a} (Fin (real (Suc k)))
      (λv. if v ∈ pk.base-case-vertices p a (Fin (real (Suc k))) then pk.dist 0 v else d
v)
      (Fin (real (Suc k))) {a..k}
```

```
    (bpi.base-case-scan-cost (Suc 0) p a (Fin (real (Suc k))))
proof −
  have bt: pk.bound-tree {a} (Fin (real (Suc k))) = {a..k}
    by (auto simp: path-k-bound-tree-band of-nat-less-iff atLeastAtMost-iff)
  have ord: pk.base-case-order a (Fin (real (Suc k))) = [a..<Suc k]
    by (rule path-k-base-order)
  have len: length (pk.base-case-order a (Fin (real (Suc k)))) ≤ p
    using ord ak ple by simp
  have U: pk.base-case-vertices p a (Fin (real (Suc k))) = {a..k}
    using pk.base-case-success[OF len] bt by simp
  have bnd: pk.base-case-bound p a (Fin (real (Suc k))) = Fin (real (Suc k))
    using len by (simp add: pk.base-case-bound-def)
  show ?thesis
    using bpi.Charged-Direct-Insert-Base[OF refl, where Δ=Suc 0
       and M-of=bmssp-level-cap p q and t=q and h=p and k=p and cap=cap
       and d=d and x=a and B=Fin (real (Suc k))]
    by (simp only: U bnd)
qed
```

Level-0 whole-path cover (fanout regime). When the FindPivots round count p already dominates the *Suc k* path vertices, a single level-0 base call on $\{0\}$ below *Fin* (*real* (*Suc k*)) settles the entire band $\{0..k\}$: the bounded tree has card *Suc k* $\leq p$, so the capped scan keeps it whole and the bound is preserved. This is the genuine level-0 child the cover lift needs.

```
lemma path-k-zero-cover:
  fixes p q cap :: nat and d :: nat ⇒ real
  assumes small: Suc k ≤ p
  shows ∃ d' U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap 0 d {0} (Fin (real (Suc k)))
      d' (Fin (real (Suc k))) U c
proof −
  have card-le: card (pk.bound-tree {0} (Fin (real (Suc k)))) ≤ p
  proof −
    have pk.bound-tree {0} (Fin (real (Suc k))) = {0..k}
      by (auto simp: path-k-bound-tree-band of-nat-less-iff atLeastAtMost-iff)
    then have card (pk.bound-tree {0} (Fin (real (Suc k)))) = Suc k by simp
    then show ?thesis using small by simp
  qed
  show ∃ d' U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap 0 d {0} (Fin (real (Suc k)))
      d' (Fin (real (Suc k))) U c
    using bpi.charged-bmssp-zero-full-cover[OF refl card-le,
        where Δ=Suc 0 and M-of=bmssp-level-cap p q and t=q and cap=cap
and d=d]
    by blast
qed
```

Level-1 base (fanout regime). Lifting the level-0 whole cover by one pull gives the level-1 whole-path cover required as the base of *path-k-cover-ge1*,

whenever p dominates the path.

lemma *path-k-base1*:
fixes *p q cap* :: *nat* **and** *d* :: *nat* ⇒ *real*
assumes *pk-le*: $p \leq k$ **and** *cover*: $Suc\ k \leq cap$ **and** *ppos*: $0 < p$
and *small*: $Suc\ k \leq p$
and *sound*: *pk.sound-label d* **and** *d0*: $d\ 0 = 0$
and *pre0*: *pk.bmssp-pre-full d* {*0*} *Infinity*
shows ∃ *U c*. *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*)
(*bmssp-level-cap p q*) *q p p cap* (*Suc 0*) *d* {*0*} (*Fin* (*real* (*Suc k*)))
(*pk.dist 0*) (*Fin* (*real* (*Suc k*))) *U c*
proof −
let *?B* = *Fin* (*real* (*Suc k*))
let *?d2* = *bpi.find-pivots-label-capped p cap d* {*0*} *?B*
let *?P* = *bpi.find-pivots-pivots-capped p cap d* {*0*} *?B*
have *reaches0*: ⋀*x*. *x* ∈ {*0*::*nat*} ⟹ *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *P0*: *?P* = {*0*} **by** (*rule path-k-source-is-sole-pivot-fin*[*OF pk-le cover d0 sound*])
have *sound2*: *pk.sound-label ?d2*
unfolding *bpi.find-pivots-label-capped-def*
by (*rule bpi.fp-iter-capped-label-sound*[*OF sound reaches0*])
have *le2*: *?d2 0* ≤ *d 0* **by** (*rule bpi.find-pivots-label-capped-le*)
have *d2-0*: *?d2 0* = *0*
proof −
have *pk.dist 0 0* ≤ *?d2 0* **using** *sound2 pk.source-in-V pk.reachable-refl*[*OF pk.source-in-V*]
unfolding *pk.sound-label-def* **by** *blast*
moreover have *pk.dist 0 0* = *0* **by** (*rule pk.dist-refl-zero*[*OF pk.source-in-V*])
ultimately show *?thesis* **using** *le2 d0* **by** *simp*
qed
have *pre2*: *pk.bmssp-pre-full ?d2* {*0*} *?B*
proof −
have *preB*: *pk.bmssp-pre-full d* {*0*} *?B*
using *pre0* **unfolding** *pk.bmssp-pre-full-def* **by** *auto*
show *?thesis*
by (*rule bpi.find-pivots-label-capped-preserves-source-pre*[*OF sound preB reaches0*])
qed
obtain *d-c U-c c-c* **where** *childrun*:
bpi.charged-direct-insert-costed-bmssp (*Suc 0*) (*bmssp-level-cap p q*) *q p p cap 0 ?d2* {*0*} *?B d-c ?B U-c c-c*
using *path-k-zero-cover*[*OF small*] **by** *blast*
have *Mpos*: *0* < *bmssp-level-cap p q 0* **using** *ppos* **by** (*simp add*: *bmssp-level-cap-def*)
have *pull*: *pull-separates* (*bpi.label-partition-view ?d2* {*0*}) (*bmssp-level-cap p q 0*)
(*real* (*Suc k*)) {*0*} (*real* (*Suc k*)) (*bpi.label-partition-view ?d2* {} :: *nat partition-view*)
unfolding *pull-separates-def bpi.label-partition-view-def*

```
    using Mpos d2-0 by simp
  have split: {0} = bpi.split-below ?d2 {0} (real (Suc k))
    using d2-0 by (auto simp: bpi.split-below-def)
  have small-tree: card (pk.bound-tree {0} (Fin 0)) < p * cap
  proof −
    have pk.bound-tree {0} (Fin 0) = {} by (rule path-k-bound-tree-source-Fin0-empty)
    moreover have 0 < p * cap using ppos cover by simp
    ultimately show ?thesis by simp
  qed
  have tail-empty:
    keys-of (batch-min-update (bpi.label-partition-view ?d2 {} :: nat partition-view)
       (bpi.edge-relaxation-pairs-in-bound d-c
          (bpi.range-tree {0} 0 ?B) (real (Suc k)) ?B @
        bpi.edge-relaxation-pairs-between d-c
          (bpi.range-tree {0} 0 ?B) (real (Suc k)) (real (Suc k)) @
        bpi.label-pairs-between ?d2 {0} (real (Suc k)) (real (Suc k)))) = {}
  proof −
    have lower: bpi.edge-relaxation-pairs-between d-c
       (bpi.range-tree {0} 0 ?B) (real (Suc k)) (real (Suc k)) = []
      unfolding bpi.edge-relaxation-pairs-between-def
      by (auto simp: filter-empty-conv split: prod.splits)
    have src: bpi.label-pairs-between ?d2 {0} (real (Suc k)) (real (Suc k)) = []
      unfolding bpi.label-pairs-between-def by (simp add: filter-empty-conv)
    have direct: bpi.edge-relaxation-pairs-in-bound d-c
       (bpi.range-tree {0} 0 ?B) (real (Suc k)) ?B = []
      unfolding bpi.edge-relaxation-pairs-in-bound-def by (auto simp: filter-empty-conv)
    show ?thesis using lower src direct unfolding batch-min-update-def by simp
  qed
  have loop:
    ∃ betas bs charged-Us child-outputs U c child-costs.
       bpi.charged-direct-insert-costed-partition-loop-state (Suc 0)
        (bmssp-level-cap p q) q p p cap 0 ?d2 {0} ?B (pk.dist 0)
        (bpi.label-partition-view ?d2 {0}) 0 betas bs ?B charged-Us child-outputs U c child-costs
    by (rule bpi.charged-loop-single-step-then-done-exists-dist
        [where Bmax = real (Suc k) and beta = real (Suc k) and b = real (Suc k)
          and S-pull = {0} and D-pull = bpi.label-partition-view ?d2 {} :: nat partition-view
          and d-child = d-c and U-child = U-c and c-child = c-c])
      (use pull pre2 split reaches0 small-tree childrun tail-empty in simp-all)
  then obtain betas bs charged-Us child-outputs U c child-costs where looprun:
    bpi.charged-direct-insert-costed-partition-loop-state (Suc 0)
      (bmssp-level-cap p q) q p p cap 0 ?d2 {0} ?B (pk.dist 0)
      (bpi.label-partition-view ?d2 {0}) 0 betas bs ?B charged-Us child-outputs U c child-costs
    by blast
  let ?U = U ∪ {v ∈ pk.bound-tree {0} ?B. ?d2 v = pk.dist 0 v}
```

have *run*: *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*) (*bmssp-level-cap p q*) *q p p cap*
(*Suc 0*) *d* {*0*} *?B* (*pk.dist 0*) *?B ?U*
(*bpi.fp-iter-capped-scan-cost p cap d* {*0*} {*0*} *?B + 0 + c*)
by (*rule bpi.charged-direct-insert-costed-bmssp-step-with-insert-cost*
[**where** *D* = *bpi.label-partition-view ?d2* {*0*} **and** *c-insert* = *0*])
(*use looprun P0* **in** ‹*simp-all add*: *bpi.partition-initial-insert-cost-bound-def pk.complete-on-def*›)
show *?thesis* **using** *run* **by** *blast*
qed

Level-1 base in the genuine $p \le k$ regime (empty-band / free-prefix construction). Unlike *path-k-base1*, which needs the round count to dominate the path (*Suc k* ≤ *p*), this builds a level-1 whole-path cover for *any* $p \le k$. The permissive charged loop is started at the band $a = p$ (claiming the already-settled prefix {*0*..<*p*} via the true-distance output label), pulls the sole pivot {*0*}, runs a single level-0 base child that returns *Fin* (*real p*), and finishes over the *empty* band {*p*..<*p*}: with $a = b = p$ the range tree is empty, so the relaxation pushes nothing and the residual partition is empty, giving an immediate *Done* back at the input bound *Fin* (*real* (*Suc k*)). This is the level-1 base the cover chain *path-k-cover-ge1* consumes.

lemma *path-k-base1-general*:
fixes *p q cap* :: *nat* **and** *d* :: *nat* ⇒ *real*
assumes *pk-le*: *p* ≤ *k* **and** *cover*: *Suc k* ≤ *cap* **and** *ppos*: *0* < *p*
and *capgt*: *1* < *cap*
and *sound*: *pk.sound-label d* **and** *d0*: *d 0* = *0*
and *pre0*: *pk.bmssp-pre-full d* {*0*} *Infinity*
shows ∃ *U c*. *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*)
(*bmssp-level-cap p q*) *q p p cap* (*Suc 0*) *d* {*0*} (*Fin* (*real* (*Suc k*)))
(*pk.dist 0*) (*Fin* (*real* (*Suc k*))) *U c*
proof −
let *?B* = *Fin* (*real* (*Suc k*))
let *?d2* = *bpi.find-pivots-label-capped p cap d* {*0*} *?B*
let *?P* = *bpi.find-pivots-pivots-capped p cap d* {*0*} *?B*
let *?dc* = λ*v*. *if v* ∈ *pk.base-case-vertices p 0 ?B then pk.dist 0 v else ?d2 v*
have *reaches0*: ⋀*x*. *x* ∈ {*0*::*nat*} ⟹ *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *P0*: *?P* = {*0*} **by** (*rule path-k-source-is-sole-pivot-fin*[*OF pk-le cover d0 sound*])
have *sound2*: *pk.sound-label ?d2*
unfolding *bpi.find-pivots-label-capped-def*
by (*rule bpi.fp-iter-capped-label-sound*[*OF sound reaches0*])
have *le2*: *?d2 0* ≤ *d 0* **by** (*rule bpi.find-pivots-label-capped-le*)
have *d2-0*: *?d2 0* = *0*
proof −
have *pk.dist 0 0* ≤ *?d2 0* **using** *sound2 pk.source-in-V pk.reachable-refl*[*OF pk.source-in-V*]
unfolding *pk.sound-label-def* **by** *blast*

```
    moreover have pk.dist 0 0 = 0 by (rule pk.dist-refl-zero[OF pk.source-in-V])
    ultimately show ?thesis using le2 d0 by simp
  qed
  have pre2: pk.bmssp-pre-full ?d2 {0} ?B
  proof −
    have preB: pk.bmssp-pre-full d {0} ?B
      using pre0 unfolding pk.bmssp-pre-full-def by auto
    show ?thesis
        by (rule bpi.find-pivots-label-capped-preserves-source-pre[OF sound preB
reaches0])
  qed
  have childrun:
    bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q p p cap 0
       ?d2 {0} ?B ?dc (Fin (real p)) {0..<p}
       (bpi.base-case-scan-cost (Suc 0) p 0 ?B)
    by (rule path-k-base-top[OF pk-le ppos])
  have Mpos: 0 < bmssp-level-cap p q 0 using ppos by (simp add: bmssp-level-
cap-def)
  have pull: pull-separates (bpi.label-partition-view ?d2 {0}) (bmssp-level-cap p q
0)
     (real (Suc k)) {0} (real (Suc k)) (bpi.label-partition-view ?d2 {} :: nat partition-
view)
    unfolding pull-separates-def bpi.label-partition-view-def
    using Mpos d2-0 by simp
  have split: {0} = bpi.split-below ?d2 {0} (real (Suc k))
    using d2-0 by (auto simp: bpi.split-below-def)
  have rt-empty: bpi.range-tree {0} (real p) (Fin (real p)) = {}
    by (auto simp: bpi.range-tree-def)
  have small-tree: card (pk.bound-tree {0} (Fin (real p))) < p ∗ cap
  proof −
    have pk.bound-tree {0} (Fin (real p)) = {0..<p}
          using pk-le by (auto simp: path-k-bound-tree-band of-nat-less-iff
atLeast0LessThan)
    then have card (pk.bound-tree {0} (Fin (real p))) = p by simp
    moreover have p < p ∗ cap using ppos capgt by simp
    ultimately show ?thesis by simp
  qed
  have tail-empty:
    keys-of (batch-min-update (bpi.label-partition-view ?d2 {} :: nat partition-view)
      (bpi.edge-relaxation-pairs-in-bound ?dc
        (bpi.range-tree {0} (real p) (Fin (real p))) (real (Suc k)) ?B @
       bpi.edge-relaxation-pairs-between ?dc
        (bpi.range-tree {0} (real p) (Fin (real p))) (real p) (real (Suc k)) @
       bpi.label-pairs-between ?d2 {0} (real p) (real (Suc k)))) = {}
  proof −
    have el: bpi.edge-list-of (bpi.outgoing-edges {}) = []
    proof −
      have set (bpi.edge-list-of (bpi.outgoing-edges {})) = bpi.outgoing-edges {}
        by (rule bpi.edge-list-of-properties(1)[OF bpi.finite-outgoing-edges])
```

```
    then show ?thesis by (simp add: bpi.outgoing-edges-def)
  qed
  have direct: bpi.edge-relaxation-pairs-in-bound ?dc
      (bpi.range-tree {0} (real p) (Fin (real p))) (real (Suc k)) ?B = []
    using rt-empty el by (simp add: bpi.edge-relaxation-pairs-in-bound-def)
  have lower: bpi.edge-relaxation-pairs-between ?dc
      (bpi.range-tree {0} (real p) (Fin (real p))) (real p) (real (Suc k)) = []
    using rt-empty el by (simp add: bpi.edge-relaxation-pairs-between-def)
  have src: bpi.label-pairs-between ?d2 {0} (real p) (real (Suc k)) = []
  proof −
    have fin: finite (keys-of (bpi.label-partition-view ?d2 {0})) by simp
    have set (partition-key-order (bpi.label-partition-view ?d2 {0})) = {0}
      using partition-key-order-properties(1)[OF fin] by simp
    then show ?thesis
      unfolding bpi.label-pairs-between-def using d2-0 ppos
      by (simp add: filter-empty-conv)
  qed
  show ?thesis using direct lower src unfolding batch-min-update-def by simp
qed
have loop:
  ∃ betas bs charged-Us child-outputs U c child-costs.
    bpi.charged-direct-insert-costed-partition-loop-state (Suc 0)
      (bmssp-level-cap p q) q p p cap 0 ?d2 {0} ?B (pk.dist 0)
    (bpi.label-partition-view ?d2 {0}) (real p) betas bs ?B charged-Us child-outputs
U c child-costs
  by (rule bpi.charged-loop-single-step-then-done-exists-dist
      [where Bmax = real (Suc k) and beta = real (Suc k) and b = real p
        and a = real p and S-pull = {0}
        and D-pull = bpi.label-partition-view ?d2 {} :: nat partition-view
        and d-child = ?dc and U-child = {0..<p}
        and c-child = bpi.base-case-scan-cost (Suc 0) p 0 ?B])
    (use pull pre2 split reaches0 small-tree childrun tail-empty pk-le in simp-all)
then obtain betas bs charged-Us child-outputs U c child-costs where looprun:
  bpi.charged-direct-insert-costed-partition-loop-state (Suc 0)
    (bmssp-level-cap p q) q p p cap 0 ?d2 {0} ?B (pk.dist 0)
    (bpi.label-partition-view ?d2 {0}) (real p) betas bs ?B charged-Us child-outputs
U c child-costs
  by blast
let ?U = U ∪ {v ∈ pk.bound-tree {0} ?B. ?d2 v = pk.dist 0 v}
have run: bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q
p p cap
    (Suc 0) d {0} ?B (pk.dist 0) ?B ?U
    (bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B + 0 + c)
  by (rule bpi.charged-direct-insert-costed-bmssp-step-with-insert-cost
      [where D = bpi.label-partition-view ?d2 {0} and c-insert = 0])
    (use looprun P0 in ‹simp-all add: bpi.partition-initial-insert-cost-bound-def
pk.complete-on-def›)
show ?thesis using run by blast
qed
```

Milestone B1, discharged: a charged top-level cost exists on P_k at the inflated schedule. Stacking the empty-band level-1 base *path-k-base1-general* through the cover chain *path-k-cover-ge1* to level $p - 1$ supplies the single charged child run (conjunct 6) that the source-pivot reduction *path-k-charged-cost-exists-if-child* turns into a charged top-level cost — with *no* run-existence assumption, in the genuine $2 \le p \le k$ regime where the cap covers the path.

```
theorem path-k-charged-cost-exists:
  assumes pk-le: sssp-log-one-third-param N ≤ k
    and cover: Suc k ≤ bmssp-level-cap (sssp-log-one-third-param N)
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    and p2: 2 ≤ sssp-log-one-third-param N
  shows ∃ c. bpi.charged-direct-insert-top-level-cost 1 N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  let ?d-fp = bpi.find-pivots-label-capped ?p ?cap pk.finite-initial-label {0} Infinity
  have ppos: 0 < ?p using p2 by simp
  have capge: ?p ≤ ?cap by (rule bmssp-level-cap-ge-level)
  have capgt: 1 < ?cap using p2 capge by simp
  have base1: ⋀d. pk.sound-label d ⟹ d 0 = 0 ⟹ pk.bmssp-pre-full d {0}
Infinity ⟹
      ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap ?p ?q)
?q ?p ?p ?cap
        (Suc 0) d {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U c
    by (rule path-k-base1-general[OF pk-le cover ppos capgt])
  have allr: ⋀v. v ∈ path-k-V k ⟹ pk.reachable 0 v by (rule path-k-all-reachable)
  have reaches0: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
    using pk.reachable-refl[OF pk.source-in-V] by simp
  have fsound: pk.sound-label pk.finite-initial-label
    by (rule pk.finite-initial-label-sound[OF allr])
  have sc: pk.finite-initial-label 0 = pk.dist 0 0
    by (rule pk.finite-initial-label-source-complete)
  have pre0: pk.bmssp-pre-full pk.finite-initial-label {0} Infinity
    by (rule pk.top-bmssp-pre-full) (use allr sc in auto)
  have dfp-sound: pk.sound-label ?d-fp
    unfolding bpi.find-pivots-label-capped-def
    by (rule bpi.fp-iter-capped-label-sound[OF fsound reaches0])
  have dfp-pre: pk.bmssp-pre-full ?d-fp {0} Infinity
      by (rule bpi.find-pivots-label-capped-preserves-source-pre[OF fsound pre0
reaches0])
  have dfp-0: ?d-fp 0 = 0
  proof −
    have pk.dist 0 0 ≤ ?d-fp 0
      using dfp-sound pk.source-in-V pk.reachable-refl[OF pk.source-in-V]
      unfolding pk.sound-label-def by blast
    moreover have ?d-fp 0 ≤ pk.finite-initial-label 0
```

```
    by (rule bpi.find-pivots-label-capped-le)
  moreover have pk.finite-initial-label 0 = 0
    using sc pk.dist-refl-zero[OF pk.source-in-V] by simp
  moreover have pk.dist 0 0 = 0 by (rule pk.dist-refl-zero[OF pk.source-in-V])
  ultimately show ?thesis by simp
 qed
 have chain: ⋀d. pk.sound-label d ⟹ d 0 = 0 ⟹ pk.bmssp-pre-full d {0}
Infinity ⟹
    ∃U c. bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap ?p ?q)
?q ?p ?p ?cap
      (Suc (?p − 2)) d {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U
c
   by (rule path-k-cover-ge1[OF pk-le cover ppos base1])
 have child: ∃U-child c-child. bpi.charged-direct-insert-costed-bmssp (Suc 0)
    (bmssp-level-cap ?p ?q) ?q ?p ?p ?cap (?p − 1) ?d-fp {0} (Fin (real (Suc k)))
    (pk.dist 0) (Fin (real (Suc k))) U-child c-child
 proof −
   have pe: Suc (?p − 2) = ?p − 1 using p2 by simp
   show ?thesis using chain[OF dfp-sound dfp-0 dfp-pre] pe by simp
 qed
 show ?thesis
   by (rule path-k-charged-cost-exists-if-child[OF pk-le cover child])
qed
```

Cost-bounded level-1 base. The cost-tracking twin of *path-k-base1-general*: the level-1 charged run built by the empty-band single-step-then-Done loop has cost exactly one FindPivots scan plus one base-case scan, hence at most $p * bpi.edge\text{-}count + p$. All four partition costs and the Done tail are zero, so no edge-range or chain terms enter — this is what keeps the cheap run polylogarithmic.

```
lemma path-k-base1-costed:
 fixes p q cap :: nat and d :: nat ⇒ real
 assumes pk-le: p ≤ k and cover: Suc k ≤ cap and ppos: 0 < p
   and capgt: 1 < cap
   and sound: pk.sound-label d and d0: d 0 = 0
   and pre0: pk.bmssp-pre-full d {0} Infinity
 shows ∃U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
    (bmssp-level-cap p q) q p p cap (Suc 0) d {0} (Fin (real (Suc k)))
    (pk.dist 0) (Fin (real (Suc k))) U c
    ∧ c ≤ p * bpi.edge-count + p
proof −
 let ?B = Fin (real (Suc k))
 let ?d2 = bpi.find-pivots-label-capped p cap d {0} ?B
 let ?P = bpi.find-pivots-pivots-capped p cap d {0} ?B
 let ?dc = λv. if v ∈ pk.base-case-vertices p 0 ?B then pk.dist 0 v else ?d2 v
 have reaches0: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
   using pk.reachable-refl[OF pk.source-in-V] by simp
 have P0: ?P = {0} by (rule path-k-source-is-sole-pivot-fin[OF pk-le cover d0
sound])
```

```
  have sound2: pk.sound-label ?d2
    unfolding bpi.find-pivots-label-capped-def
    by (rule bpi.fp-iter-capped-label-sound[OF sound reaches0])
  have le2: ?d2 0 ≤ d 0 by (rule bpi.find-pivots-label-capped-le)
  have d2-0: ?d2 0 = 0
  proof −
    have pk.dist 0 0 ≤ ?d2 0 using sound2 pk.source-in-V pk.reachable-refl[OF
pk.source-in-V]
      unfolding pk.sound-label-def by blast
    moreover have pk.dist 0 0 = 0 by (rule pk.dist-refl-zero[OF pk.source-in-V])
    ultimately show ?thesis using le2 d0 by simp
  qed
  have pre2: pk.bmssp-pre-full ?d2 {0} ?B
  proof −
    have preB: pk.bmssp-pre-full d {0} ?B
      using pre0 unfolding pk.bmssp-pre-full-def by auto
    show ?thesis
        by (rule bpi.find-pivots-label-capped-preserves-source-pre[OF sound preB
reaches0])
  qed
  have childrun:
    bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q p p cap 0
      ?d2 {0} ?B ?dc (Fin (real p)) {0..<p}
      (bpi.base-case-scan-cost (Suc 0) p 0 ?B)
    by (rule path-k-base-top[OF pk-le ppos])
  have Mpos: 0 < bmssp-level-cap p q 0 using ppos by (simp add: bmssp-level-
cap-def)
  have pull: pull-separates (bpi.label-partition-view ?d2 {0}) (bmssp-level-cap p q
0)
     (real (Suc k)) {0} (real (Suc k)) (bpi.label-partition-view ?d2 {} :: nat partition-
view)
    unfolding pull-separates-def bpi.label-partition-view-def
    using Mpos d2-0 by simp
  have split: {0} = bpi.split-below ?d2 {0} (real (Suc k))
    using d2-0 by (auto simp: bpi.split-below-def)
  have rt-empty: bpi.range-tree {0} (real p) (Fin (real p)) = {}
    by (auto simp: bpi.range-tree-def)
  have small-tree: card (pk.bound-tree {0} (Fin (real p))) < p ∗ cap
  proof −
    have pk.bound-tree {0} (Fin (real p)) = {0..<p}
          using pk-le by (auto simp: path-k-bound-tree-band of-nat-less-iff
atLeast0LessThan)
    then have card (pk.bound-tree {0} (Fin (real p))) = p by simp
    moreover have p < p ∗ cap using ppos capgt by simp
    ultimately show ?thesis by simp
  qed
  have tail-empty:
    keys-of (batch-min-update (bpi.label-partition-view ?d2 {} :: nat partition-view)
      (bpi.edge-relaxation-pairs-in-bound ?dc
```

```
      (bpi.range-tree {0} (real p) (Fin (real p))) (real (Suc k)) ?B @
     bpi.edge-relaxation-pairs-between ?dc
      (bpi.range-tree {0} (real p) (Fin (real p))) (real p) (real (Suc k)) @
     bpi.label-pairs-between ?d2 {0} (real p) (real (Suc k)))) = {}
 proof −
  have el: bpi.edge-list-of (bpi.outgoing-edges {}) = []
  proof −
   have set (bpi.edge-list-of (bpi.outgoing-edges {})) = bpi.outgoing-edges {}
    by (rule bpi.edge-list-of-properties(1)[OF bpi.finite-outgoing-edges])
   then show ?thesis by (simp add: bpi.outgoing-edges-def)
  qed
  have direct: bpi.edge-relaxation-pairs-in-bound ?dc
     (bpi.range-tree {0} (real p) (Fin (real p))) (real (Suc k)) ?B = []
   using rt-empty el by (simp add: bpi.edge-relaxation-pairs-in-bound-def)
  have lower: bpi.edge-relaxation-pairs-between ?dc
     (bpi.range-tree {0} (real p) (Fin (real p))) (real p) (real (Suc k)) = []
   using rt-empty el by (simp add: bpi.edge-relaxation-pairs-between-def)
  have src: bpi.label-pairs-between ?d2 {0} (real p) (real (Suc k)) = []
  proof −
   have fin: finite (keys-of (bpi.label-partition-view ?d2 {0})) by simp
   have set (partition-key-order (bpi.label-partition-view ?d2 {0})) = {0}
    using partition-key-order-properties(1)[OF fin] by simp
   then show ?thesis
    unfolding bpi.label-pairs-between-def using d2-0 ppos
    by (simp add: filter-empty-conv)
  qed
  show ?thesis using direct lower src unfolding batch-min-update-def by simp
 qed
 have beta-bd: bound-le (Fin (real (Suc k))) ?B by simp
 have a-le-b: (real p :: real) ≤ real p by simp
 have a-bd: bound-le (Fin (real p)) ?B using pk-le by simp
 have reaches-ball: ∀ x∈{0::nat}. pk.reachable 0 x using reaches0 by simp
 have looprun:
  bpi.charged-direct-insert-costed-partition-loop-state (Suc 0)
   (bmssp-level-cap p q) q p p cap 0 ?d2 {0} ?B (pk.dist 0)
   (bpi.label-partition-view ?d2 {0}) (real p) [real (Suc k)] [real p] ?B
   (bpi.range-tree {0} (real p) (Fin (real p)) # [bpi.range-tree {0} (real p) ?B])
   [{0..<p}]
   (pk.bound-tree {0} (Fin (real p)) ∪
     ⋃(set (bpi.range-tree {0} (real p) (Fin (real p)) # [bpi.range-tree {0} (real
p) ?B])))
   (bpi.base-case-scan-cost (Suc 0) p 0 ?B) [bpi.base-case-scan-cost (Suc 0) p 0
?B]
  by (rule bpi.charged-loop-single-step-then-done-costed-dist
      [where Bmax = real (Suc k) and beta = real (Suc k) and b = real p
       and a = real p and S-pull = {0}
       and D-pull = bpi.label-partition-view ?d2 {} :: nat partition-view
       and d-child = ?dc and U-child = {0..<p}
       and c-child = bpi.base-case-scan-cost (Suc 0) p 0 ?B])
```

```
    (use pull beta-bd pre2 split reaches-ball a-le-b a-bd small-tree childrun
      tail-empty a-bd in simp-all)
  let ?U = (pk.bound-tree {0} (Fin (real p)) ∪
      ⋃(set (bpi.range-tree {0} (real p) (Fin (real p)) # [bpi.range-tree {0} (real
p) ?B])))
    ∪ {v ∈ pk.bound-tree {0} ?B. ?d2 v = pk.dist 0 v}
  have run: bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q
p p cap
    (Suc 0) d {0} ?B (pk.dist 0) ?B ?U
    (bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B + 0
      + bpi.base-case-scan-cost (Suc 0) p 0 ?B)
   by (rule bpi.charged-direct-insert-costed-bmssp-step-with-insert-cost
      [where D = bpi.label-partition-view ?d2 {0} and c-insert = 0])
    (use looprun P0 in ‹simp-all add: bpi.partition-initial-insert-cost-bound-def
pk.complete-on-def›)
  have cost-le: bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B + 0
      + bpi.base-case-scan-cost (Suc 0) p 0 ?B ≤ p * bpi.edge-count + p
  proof −
   have fp: bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B ≤ p * bpi.edge-count
     by (rule bpi.fp-iter-capped-scan-cost-le-rounds)
   have base: bpi.base-case-scan-cost (Suc 0) p 0 ?B ≤ p
     using bpi.base-case-scan-cost-le[OF bpi.bounded-edge-outdegree] by simp
   show ?thesis using fp base by simp
  qed
  show ?thesis using run cost-le by blast
qed
```

Cost-bounded cover lift ($l \to Suc\ l$). The cost-tracking twin of *path-k-cover-lift*: lifting a level-l whole-path cover of cost at most Cb to level $Suc\ l$ adds exactly one FindPivots scan (cost $\leq p \cdot$ *edge-count*); all four partition costs and the Done tail are zero. Stacking this from the level-1 base accumulates one scan per level, giving the polylogarithmic cheap-run cost.

```
lemma path-k-cover-lift-costed:
  fixes p q cap l Cb :: nat and d :: nat ⇒ real
  assumes pk-le: p ≤ k and cover: Suc k ≤ cap
   and ppos: 0 < p
   and sound: pk.sound-label d and d0: d 0 = 0
   and pre0: pk.bmssp-pre-full d {0} Infinity
   and child: ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap l
      (bpi.find-pivots-label-capped p cap d {0} (Fin (real (Suc k))))
      {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U c
      ∧ c ≤ Cb
  shows ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
     (bmssp-level-cap p q) q p p cap (Suc l) d {0} (Fin (real (Suc k)))
     (pk.dist 0) (Fin (real (Suc k))) U c
     ∧ c ≤ Cb + p * bpi.edge-count
proof −
  let ?B = Fin (real (Suc k))
```

let *?d2* = *bpi.find-pivots-label-capped p cap d* {*0*} *?B*
let *?P* = *bpi.find-pivots-pivots-capped p cap d* {*0*} *?B*
have *reaches0*: ⋀*x*. *x* ∈ {*0*::*nat*} ⟹ *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *P0*: *?P* = {*0*} **by** (*rule path-k-source-is-sole-pivot-fin*[*OF pk-le cover d0 sound*])
have *sound2*: *pk.sound-label ?d2*
unfolding *bpi.find-pivots-label-capped-def*
by (*rule bpi.fp-iter-capped-label-sound*[*OF sound reaches0*])
have *le2*: *?d2 0* ≤ *d 0* **by** (*rule bpi.find-pivots-label-capped-le*)
have *d2-0*: *?d2 0* = *0*
proof −
have *pk.dist 0 0* ≤ *?d2 0* **using** *sound2 pk.source-in-V pk.reachable-refl*[*OF pk.source-in-V*]
unfolding *pk.sound-label-def* **by** *blast*
moreover have *pk.dist 0 0* = *0* **by** (*rule pk.dist-refl-zero*[*OF pk.source-in-V*])
ultimately show *?thesis* **using** *le2 d0* **by** *simp*
qed
have *pre2*: *pk.bmssp-pre-full ?d2* {*0*} *?B*
proof −
have *preB*: *pk.bmssp-pre-full d* {*0*} *?B*
using *pre0* **unfolding** *pk.bmssp-pre-full-def* **by** *auto*
show *?thesis*
by (*rule bpi.find-pivots-label-capped-preserves-source-pre*[*OF sound preB reaches0*])
qed
obtain *U-c c-c* **where** *childrun*:
bpi.charged-direct-insert-costed-bmssp (*Suc 0*) (*bmssp-level-cap p q*) *q p p cap l ?d2* {*0*} *?B* (*pk.dist 0*) *?B U-c c-c*
and *c-c-le*: *c-c* ≤ *Cb*
using *child* **by** *blast*
have *Mpos*: *0* < *bmssp-level-cap p q l* **using** *ppos* **by** (*simp add*: *bmssp-level-cap-def*)
have *pull*: *pull-separates* (*bpi.label-partition-view ?d2* {*0*}) (*bmssp-level-cap p q l*) (*real* (*Suc k*)) {*0*} (*real* (*Suc k*)) (*bpi.label-partition-view ?d2* {} :: *nat partition-view*)
unfolding *pull-separates-def bpi.label-partition-view-def*
using *Mpos d2-0* **by** *simp*
have *split*: {*0*} = *bpi.split-below ?d2* {*0*} (*real* (*Suc k*))
using *d2-0* **by** (*auto simp*: *bpi.split-below-def*)
have *small*: *card* (*pk.bound-tree* {*0*} (*Fin 0*)) < *p* ∗ *cap*
proof −
have *pk.bound-tree* {*0*} (*Fin 0*) = {} **by** (*rule path-k-bound-tree-source-Fin0-empty*)
moreover have *0* < *p* ∗ *cap* **using** *ppos cover* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
have *tail-empty*:
keys-of (*batch-min-update* (*bpi.label-partition-view ?d2* {} :: *nat partition-view*)

```
      (bpi.edge-relaxation-pairs-in-bound (pk.dist 0)
        (bpi.range-tree {0} 0 (Fin (real (Suc k)))) (real (Suc k)) ?B @
       bpi.edge-relaxation-pairs-between (pk.dist 0)
        (bpi.range-tree {0} 0 (Fin (real (Suc k)))) (real (Suc k)) (real (Suc k)) @
       bpi.label-pairs-between ?d2 {0} (real (Suc k)) (real (Suc k)))) = {}
  proof −
   have lower: bpi.edge-relaxation-pairs-between (pk.dist 0)
      (bpi.range-tree {0} 0 (Fin (real (Suc k)))) (real (Suc k)) (real (Suc k)) = []
     unfolding bpi.edge-relaxation-pairs-between-def
     by (auto simp: filter-empty-conv split: prod.splits)
   have src: bpi.label-pairs-between ?d2 {0} (real (Suc k)) (real (Suc k)) = []
     unfolding bpi.label-pairs-between-def by (simp add: filter-empty-conv)
   have direct: bpi.edge-relaxation-pairs-in-bound (pk.dist 0)
      (bpi.range-tree {0} 0 (Fin (real (Suc k)))) (real (Suc k)) ?B = []
     unfolding bpi.edge-relaxation-pairs-in-bound-def by (auto simp: filter-empty-
conv)
   show ?thesis using lower src direct unfolding batch-min-update-def by simp
  qed
  have beta-bd: bound-le (Fin (real (Suc k))) ?B by simp
  have a-le-b: (0::real) ≤ real (Suc k) by simp
  have a0-bd: bound-le (Fin (0::real)) ?B by simp
  have reaches-ball: ∀ x∈{0::nat}. pk.reachable 0 x using reaches0 by simp
  have looprun:
   bpi.charged-direct-insert-costed-partition-loop-state (Suc 0)
     (bmssp-level-cap p q) q p p cap l ?d2 {0} ?B (pk.dist 0)
     (bpi.label-partition-view ?d2 {0}) 0 [real (Suc k)] [real (Suc k)] ?B
     (bpi.range-tree {0} 0 (Fin (real (Suc k)))
       # [bpi.range-tree {0} (real (Suc k)) ?B])
     [U-c]
     (pk.bound-tree {0} (Fin 0) ∪
       ⋃(set (bpi.range-tree {0} 0 (Fin (real (Suc k))))
             # [bpi.range-tree {0} (real (Suc k)) ?B])))
     c-c [c-c]
   by (rule bpi.charged-loop-single-step-then-done-costed-dist
      [where Bmax = real (Suc k) and beta = real (Suc k) and b = real (Suc k)
       and a = 0::real and S-pull = {0}
       and D-pull = bpi.label-partition-view ?d2 {} :: nat partition-view
       and d-child = pk.dist 0 and U-child = U-c and c-child = c-c])
     (use pull beta-bd pre2 split reaches-ball a-le-b a0-bd small childrun
      tail-empty beta-bd in simp-all)
  let ?U = (pk.bound-tree {0} (Fin 0) ∪
       ⋃(set (bpi.range-tree {0} 0 (Fin (real (Suc k))))
             # [bpi.range-tree {0} (real (Suc k)) ?B])))
     ∪ {v ∈ pk.bound-tree {0} ?B. ?d2 v = pk.dist 0 v}
  have run: bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q
p p cap
     (Suc l) d {0} ?B (pk.dist 0) ?B ?U
     (bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B + 0 + c-c)
   by (rule bpi.charged-direct-insert-costed-bmssp-step-with-insert-cost
```

```
      [where D = bpi.label-partition-view ?d2 {0} and c-insert = 0])
    (use looprun P0 in ‹simp-all add: bpi.partition-initial-insert-cost-bound-def
pk.complete-on-def›)
  have cost-le: bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B + 0 + c-c
    ≤ Cb + p * bpi.edge-count
  proof −
    have fp: bpi.fp-iter-capped-scan-cost p cap d {0} {0} ?B ≤ p * bpi.edge-count
      by (rule bpi.fp-iter-capped-scan-cost-le-rounds)
    show ?thesis using fp c-c-le by simp
  qed
  show ?thesis using run cost-le by blast
qed
```

Cost-bounded level chain. Stacking the cost-bounded cover lift from the cost-bounded level-1 base, a whole-path cover of cost at most $(Suc\ l) \cdot (p \cdot edge\text{-}count) + p$ exists at every level $Suc\ l$: each level adds one FindPivots scan. Specialising at $l = p - 1$ bounds the single charged child run feeding the top-level reduction by $p \cdot (p \cdot edge\text{-}count) + p$, which is polylogarithmic in the schedule.

```
lemma path-k-cover-ge1-costed:
  fixes p q cap :: nat
  assumes pk-le: p ≤ k and cover: Suc k ≤ cap and ppos: 0 < p
  shows ⋀d. pk.sound-label d ⟹ d 0 = 0 ⟹ pk.bmssp-pre-full d {0} Infinity
⟹
    ∃U c. bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q p
p
      cap (Suc l) d {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U c
      ∧ c ≤ Suc l * (p * bpi.edge-count) + p
proof (induct l)
  case 0
  have capgt: 1 < cap using cover ppos pk-le by linarith
  show ?case
    using path-k-base1-costed[OF pk-le cover ppos capgt 0(1) 0(2) 0(3)]
    by simp
next
  case (Suc l)
  let ?d2 = bpi.find-pivots-label-capped p cap d {0} (Fin (real (Suc k)))
  have reaches0: ⋀x. x ∈ {0::nat} ⟹ pk.reachable 0 x
    using pk.reachable-refl[OF pk.source-in-V] by simp
  have sound2: pk.sound-label ?d2
    unfolding bpi.find-pivots-label-capped-def
    by (rule bpi.fp-iter-capped-label-sound[OF Suc(2) reaches0])
  have d2-0: ?d2 0 = 0
  proof −
    have pk.dist 0 0 ≤ ?d2 0 using sound2 pk.source-in-V pk.reachable-refl[OF
pk.source-in-V]
      unfolding pk.sound-label-def by blast
    moreover have pk.dist 0 0 = 0 by (rule pk.dist-refl-zero[OF pk.source-in-V])
    ultimately show ?thesis
```

```
    using bpi.find-pivots-label-capped-le[of p cap d {0} Fin (real (Suc k)) 0] Suc(3)
by simp
  qed
  have pre2: pk.bmssp-pre-full ?d2 {0} Infinity
  proof (rule pk.top-bmssp-pre-full)
    show ⋀v. v ∈ path-k-V k ⟹ pk.reachable 0 v by (rule path-k-all-reachable)
    show ?d2 0 = pk.dist 0 0 using d2-0 pk.dist-refl-zero[OF pk.source-in-V] by
simp
  qed
  have child: ∃ U c. bpi.charged-direct-insert-costed-bmssp (Suc 0)
      (bmssp-level-cap p q) q p p cap (Suc l) ?d2 {0} (Fin (real (Suc k)))
      (pk.dist 0) (Fin (real (Suc k))) U c
      ∧ c ≤ Suc l * (p * bpi.edge-count) + p
    by (rule Suc(1)[OF sound2 d2-0 pre2])
  obtain U c where run:
    bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap p q) q p p
      cap (Suc (Suc l)) d {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k)))
U c
    and cle: c ≤ Suc l * (p * bpi.edge-count) + p + p * bpi.edge-count
    using path-k-cover-lift-costed[OF pk-le cover ppos Suc(2) Suc(3) Suc(4) child]
    by blast
  have c ≤ Suc (Suc l) * (p * bpi.edge-count) + p
    using cle by (simp add: algebra-simps)
  then show ?case using run by blast
qed
```

Cost-bounded top-level reduction. The cost-tracking twin of *path-k-charged-cost-exists-if-child*: the source-pivot top-level run pulls the sole source pivot once, runs the bounded level-$p-1$ child of cost at most Cb, and terminates (the direct batch is empty on the path). Its cost is one more FindPivots scan on top of the child, hence at most $Cb + p \cdot edge\text{-}count$.

```
lemma path-k-charged-cost-le-if-child:
  fixes Cb :: nat
  assumes pk-le: sssp-log-one-third-param N ≤ k
    and cover: Suc k ≤ bmssp-level-cap (sssp-log-one-third-param N)
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    and ppos: 0 < sssp-log-one-third-param N
    and child:
      ∃ U-child c-child.
        bpi.charged-direct-insert-costed-bmssp (Suc 0)
          (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param
N))
          (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
          (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))
          (sssp-log-one-third-param N − 1)
          (bpi.find-pivots-label-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))
          pk.finite-initial-label {0} Infinity)
        {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k))) U-child c-child
      ∧ c-child ≤ Cb
  shows ∃ c. bpi.charged-direct-insert-top-level-cost 1 N c
     ∧ c ≤ Cb + sssp-log-one-third-param N * bpi.edge-count
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  let ?d-fp = bpi.find-pivots-label-capped ?p ?cap pk.finite-initial-label {0} Infinity
  let ?P = bpi.find-pivots-pivots-capped ?p ?cap pk.finite-initial-label {0} Infinity
  have P0: ?P = {0} by (rule path-k-source-is-sole-pivot[OF pk-le cover])
  have d-lt: ?d-fp 0 < real (Suc k)
  proof −
    have ?d-fp 0 ≤ 0 by (rule path-k-fp-label-source-le-zero)
    also have (0::real) < real (Suc k) by simp
    finally show ?thesis .
  qed
  have pre: pk.bmssp-pre-full ?d-fp {0} (Fin (real (Suc k)))
    by (rule path-k-fp-source-pre-full)
  have cap-pos: 0 < ?p * ?cap
    using sssp-log-one-third-param-pos[of N] by (simp add: bmssp-level-cap-def)
  have card-lt: card (pk.bound-tree {0} (Fin 0)) < ?p * ?cap
  proof −
    have pk.bound-tree {0} (Fin 0) = {} by (rule path-k-bound-tree-source-Fin0-
empty)
    then show ?thesis using cap-pos by simp
  qed
  have batch: bpi.edge-relaxation-pairs-in-bound (pk.dist 0)
      (bpi.range-tree {0} 0 (Fin (real (Suc k)))) (real (Suc k)) Infinity = []
    using P0 path-k-direct-batch-empty by simp
  obtain U-child c-child where childrun:
    bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap ?p ?q) ?q ?p ?p
?cap
      (?p − 1) ?d-fp {0} (Fin (real (Suc k))) (pk.dist 0) (Fin (real (Suc k)))
      U-child c-child
    and c-child-le: c-child ≤ Cb
    using child by blast
  have Mpos: 0 < bmssp-level-cap ?p ?q (?p − 1)
    using ppos by (simp add: bmssp-level-cap-def)
  have pull: pull-separates (bpi.label-partition-view ?d-fp {0})
      (bmssp-level-cap ?p ?q (?p − 1)) (real (Suc k)) {0} (real (Suc k))
      (bpi.label-partition-view ?d-fp {} :: nat partition-view)
    unfolding pull-separates-def bpi.label-partition-view-def
    using Mpos d-lt by simp
  have split: {0} = bpi.split-below ?d-fp {0} (real (Suc k))
    using d-lt by (auto simp: bpi.split-below-def)
```

have *reaches0*: $\forall x{\in}\{0{::}nat\}$. *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *beta-bd*: *bound-le* (*Fin* (*real* (*Suc k*))) *Infinity* **by** *simp*
have *a-le-b*: (*0*::*real*) ≤ *real* (*Suc k*) **by** *simp*
have *a0-bd*: *bound-le* (*Fin* (*0*::*real*)) *Infinity* **by** *simp*
have *tail-empty*:
keys-of (*batch-min-update* (*bpi.label-partition-view ?d-fp* {} :: *nat partition-view*)
(*bpi.edge-relaxation-pairs-in-bound* (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) *Infinity* @
bpi.edge-relaxation-pairs-between (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) (*real* (*Suc k*)) @
bpi.label-pairs-between ?d-fp {*0*} (*real* (*Suc k*)) (*real* (*Suc k*)))) = {}
proof −
have *lower*: *bpi.edge-relaxation-pairs-between* (*pk.dist 0*)
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))) (*real* (*Suc k*)) (*real* (*Suc k*)) = []
unfolding *bpi.edge-relaxation-pairs-between-def*
by (*auto simp*: *filter-empty-conv split*: *prod.splits*)
have *src*: *bpi.label-pairs-between ?d-fp* {*0*} (*real* (*Suc k*)) (*real* (*Suc k*)) = []
unfolding *bpi.label-pairs-between-def* **by** (*simp add*: *filter-empty-conv*)
show *?thesis* **using** *lower src batch* **unfolding** *batch-min-update-def* **by** *simp*
qed
have *looprun*:
bpi.charged-direct-insert-costed-partition-loop-state (*Suc 0*)
(*bmssp-level-cap ?p ?q*) *?q ?p ?p ?cap* (*?p* − *1*) *?d-fp* {*0*} *Infinity* (*pk.dist 0*)
(*bpi.label-partition-view ?d-fp* {*0*}) *0* [*real* (*Suc k*)] [*real* (*Suc k*)] *Infinity*
(*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))
[*bpi.range-tree* {*0*} (*real* (*Suc k*)) *Infinity*])
[*U-child*]
(*pk.bound-tree* {*0*} (*Fin 0*) ∪
⋃(*set* (*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))
[*bpi.range-tree* {*0*} (*real* (*Suc k*)) *Infinity*])))
c-child [*c-child*]
by (*rule bpi.charged-loop-single-step-then-done-costed-dist*
[**where** *Bmax* = *real* (*Suc k*) **and** *beta* = *real* (*Suc k*) **and** *b* = *real* (*Suc k*)
and *a* = *0*::*real* **and** *S-pull* = {*0*}
and *D-pull* = *bpi.label-partition-view ?d-fp* {} :: *nat partition-view*
and *d-child* = *pk.dist 0* **and** *U-child* = *U-child* **and** *c-child* = *c-child*])
(*use pull beta-bd pre split reaches0 a-le-b a0-bd card-lt childrun*
tail-empty beta-bd **in** *simp-all*)
let *?U* = (*pk.bound-tree* {*0*} (*Fin 0*) ∪
⋃(*set* (*bpi.range-tree* {*0*} *0* (*Fin* (*real* (*Suc k*)))
[*bpi.range-tree* {*0*} (*real* (*Suc k*)) *Infinity*])))
∪ {*v* ∈ *pk.bound-tree* {*0*} *Infinity*. *?d-fp v* = *pk.dist 0 v*}
let *?c* = *bpi.fp-iter-capped-scan-cost ?p ?cap pk.finite-initial-label* {*0*} {*0*} *Infinity*
+ *0* + *c-child*
have *run*: *bpi.charged-direct-insert-costed-bmssp* (*Suc 0*) (*bmssp-level-cap ?p ?q*)
?q ?p ?p ?cap
(*Suc* (*?p* − *1*)) *pk.finite-initial-label* {*0*} *Infinity* (*pk.dist 0*) *Infinity ?U ?c*
by (*rule bpi.charged-direct-insert-costed-bmssp-step-with-insert-cost*

```
      [where D = bpi.label-partition-view ?d-fp {0} and c-insert = 0])
    (use looprun P0 in ‹simp-all add: bpi.partition-initial-insert-cost-bound-def
pk.complete-on-def›)
  have p-eq: Suc (?p − 1) = ?p using ppos by simp
  have run': bpi.charged-direct-insert-costed-bmssp (Suc 0) (bmssp-level-cap ?p ?q)
?q ?p ?p ?cap
      ?p pk.finite-initial-label {0} Infinity (pk.dist 0) Infinity ?U ?c
    using run p-eq by simp
  have toprun: bpi.charged-direct-insert-top-level-run (Suc 0) N (pk.dist 0) ?U ?c
    unfolding bpi.charged-direct-insert-top-level-run-def
    using run' by (simp add: Let-def)
  have topcost: bpi.charged-direct-insert-top-level-cost (Suc 0) N ?c
    unfolding bpi.charged-direct-insert-top-level-cost-def using toprun by blast
  have cost-le: ?c ≤ Cb + ?p ∗ bpi.edge-count
  proof −
    have fp: bpi.fp-iter-capped-scan-cost ?p ?cap pk.finite-initial-label {0} {0}
Infinity
      ≤ ?p ∗ bpi.edge-count
     by (rule bpi.fp-iter-capped-scan-cost-le-rounds)
    show ?thesis using fp c-child-le by simp
  qed
  have main: ∃ c. bpi.charged-direct-insert-top-level-cost (Suc 0) N c
     ∧ c ≤ Cb + ?p ∗ bpi.edge-count
    using topcost cost-le by blast
  then show ?thesis by simp
qed
```

The cheap charged top-level cost is polylogarithmic. Assembling the cost-bounded level chain (to level $p-1$) and the cost-bounded top-level reduction: in the genuine $2 \le p \le k$ pivot regime where the cap covers the path, a charged top-level cost exists that is bounded by $p \cdot (p \cdot \mathit{edge\text{-}count}) + p$ — a single *square-of-the-pivot-rounds* times edge factor. No band-containment or amortized engine is needed: the cheap run pays one FindPivots scan per recursion level plus one base scan.

```
theorem path-k-charged-cost-le:
  assumes pk-le: sssp-log-one-third-param N ≤ k
    and cover: Suc k ≤ bmssp-level-cap (sssp-log-one-third-param N)
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    and p2: 2 ≤ sssp-log-one-third-param N
  shows ∃ c. bpi.charged-direct-insert-top-level-cost 1 N c
     ∧ c ≤ sssp-log-one-third-param N ∗ (sssp-log-one-third-param N ∗ bpi.edge-
count)
          + sssp-log-one-third-param N
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  let ?d-fp = bpi.find-pivots-label-capped ?p ?cap pk.finite-initial-label {0} Infinity
  have ppos: 0 < ?p using p2 by simp
```

have *allr*: $\bigwedge v.\ v \in$ *path-k-V k* $\Longrightarrow$ *pk.reachable 0 v* **by** (*rule path-k-all-reachable*)
have *reaches0*: $\bigwedge x.\ x \in \{0::nat\} \Longrightarrow$ *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *fsound*: *pk.sound-label pk.finite-initial-label*
by (*rule pk.finite-initial-label-sound*[*OF allr*])
have *sc*: *pk.finite-initial-label 0 = pk.dist 0 0*
by (*rule pk.finite-initial-label-source-complete*)
have *pre0*: *pk.bmssp-pre-full pk.finite-initial-label* {*0*} *Infinity*
by (*rule pk.top-bmssp-pre-full*) (*use allr sc* **in** *auto*)
have *dfp-sound*: *pk.sound-label ?d-fp*
unfolding *bpi.find-pivots-label-capped-def*
by (*rule bpi.fp-iter-capped-label-sound*[*OF fsound reaches0*])
have *dfp-pre*: *pk.bmssp-pre-full ?d-fp* {*0*} *Infinity*
by (*rule bpi.find-pivots-label-capped-preserves-source-pre*[*OF fsound pre0 reaches0*])
have *dfp-0*: *?d-fp 0 = 0*
proof −
have *pk.dist 0 0* ≤ *?d-fp 0*
using *dfp-sound pk.source-in-V pk.reachable-refl*[*OF pk.source-in-V*]
unfolding *pk.sound-label-def* **by** *blast*
moreover have *?d-fp 0* ≤ *pk.finite-initial-label 0*
by (*rule bpi.find-pivots-label-capped-le*)
moreover have *pk.finite-initial-label 0 = 0*
using *sc pk.dist-refl-zero*[*OF pk.source-in-V*] **by** *simp*
moreover have *pk.dist 0 0 = 0* **by** (*rule pk.dist-refl-zero*[*OF pk.source-in-V*])
ultimately show *?thesis* **by** *simp*
qed
have *chain*: ∃ *U c. bpi.charged-direct-insert-costed-bmssp* (*Suc 0*)
(*bmssp-level-cap ?p ?q*) *?q ?p ?p ?cap* (*Suc* (*?p* − *2*)) *?d-fp* {*0*} (*Fin* (*real* (*Suc k*)))
(*pk.dist 0*) (*Fin* (*real* (*Suc k*))) *U c*
∧ *c* ≤ *Suc* (*?p* − *2*) ∗ (*?p* ∗ *bpi.edge-count*) + *?p*
by (*rule path-k-cover-ge1-costed*[*OF pk-le cover ppos dfp-sound dfp-0 dfp-pre*])
have *pe*: *Suc* (*?p* − *2*) = *?p* − *1* **using** *p2* **by** *simp*
have *child*: ∃ *U-child c-child. bpi.charged-direct-insert-costed-bmssp* (*Suc 0*)
(*bmssp-level-cap ?p ?q*) *?q ?p ?p ?cap* (*?p* − *1*) *?d-fp* {*0*} (*Fin* (*real* (*Suc k*)))
(*pk.dist 0*) (*Fin* (*real* (*Suc k*))) *U-child c-child*
∧ *c-child* ≤ (*?p* − *1*) ∗ (*?p* ∗ *bpi.edge-count*) + *?p*
using *chain pe* **by** *simp*
obtain *c* **where** *tc*: *bpi.charged-direct-insert-top-level-cost 1 N c*
and *cle*: *c* ≤ ((*?p* − *1*) ∗ (*?p* ∗ *bpi.edge-count*) + *?p*) + *?p* ∗ *bpi.edge-count*
using *path-k-charged-cost-le-if-child*[*OF pk-le cover ppos child*] **by** *blast*
have *arith*: ((*?p* − *1*) ∗ (*?p* ∗ *bpi.edge-count*) + *?p*) + *?p* ∗ *bpi.edge-count*
= *?p* ∗ (*?p* ∗ *bpi.edge-count*) + *?p*
proof −
obtain *m* **where** *pm*: *?p = Suc m* **using** *gr0-implies-Suc*[*OF ppos*] **by** *blast*
show *?thesis* **by** (*simp add*: *pm*)
qed
have *c* ≤ *?p* ∗ (*?p* ∗ *bpi.edge-count*) + *?p* **using** *cle arith* **by** *simp*

with *tc* **show** *?thesis* **by** *blast*
qed

Charged-cost structural facts. The next facts record permissive-relation structure on the path: a returned bound that covers the path forces the whole vertex set into the bounded tree, a level-0 base settles at most $kk \leq k$ vertices, and (the harder fact) a source-pivot loop whose data structure already excludes the source but is still nonempty can never finish. They are reusable ingredients for bounding charged costs.

A covering returned bound forces the whole path into the bounded tree.

lemma *path-k-cover-imp-V-subset*:
assumes *cov*: *bound-le* (*Fin* (*real* (*Suc k*))) B'
shows *path-k-V k* $\subseteq$ *pk.bound-tree* $\{0\}$ B'
proof
fix *v* **assume** $v \in$ *path-k-V k*
then have *vk*: $v \leq k$ **by** (*simp add*: *path-k-V-eq*)
have *below*: *below-bound* (*pk.dist 0 v*) B'
proof (*cases* B')
case (*Fin r*)
have *real* (*Suc k*) $\leq r$ **using** *cov Fin* **by** *simp*
then have *real v* $< r$ **using** *vk* **by** *simp*
then show *?thesis* **using** *Fin path-k-dist-eq*[*OF vk*] **by** *simp*
next
case *Infinity*
then show *?thesis* **by** *simp*
qed
have *thr*: *pk.through* $\{0\}$ *v* **using** *path-k-through-iff*[*OF vk*] **by** *simp*
show $v \in$ *pk.bound-tree* $\{0\}$ B'
unfolding *pk.bound-tree-def*
using *vk thr below path-k-all-reachable* **by** (*simp add*: *path-k-V-eq*)
qed

Base of the no-coverage induction. A level-0 charged base run from the source settles at most $kk \leq k$ vertices, so its returned bound cannot cover the *Suc k* path vertices.

lemma *path-k-charged-base-noncover*:
fixes *kk* :: *nat*
assumes *kk-le*: $kk \leq k$
and *run*: *bpi.charged-direct-insert-costed-bmssp* Δ *M-of t h kk cap 0 d* $\{0\}$ *B d′ B′ U c*
and *sound*: *pk.sound-label d*
and *pre*: *pk.bmssp-pre-full d* $\{0\}$ *B*
shows $\neg$ *bound-le* (*Fin* (*real* (*Suc k*))) B'
proof
assume *cov*: *bound-le* (*Fin* (*real* (*Suc k*))) B'
have *reaches*: $\bigwedge x.\ x \in \{0::nat\} \Longrightarrow$ *pk.reachable 0 x*
using *pk.reachable-refl*[*OF pk.source-in-V*] **by** *simp*
have *post*: *pk.bmssp-post-full d* $\{0\}$ *B d′ B′ U*

```
  by (rule bpi.charged-direct-insert-costed-bmssp-correct[OF run sound pre reaches])
 have U-eq: U = pk.bound-tree {0} B'
  using post unfolding pk.bmssp-post-full-def by blast
 have sub: path-k-V k ⊆ U
  using path-k-cover-imp-V-subset[OF cov] U-eq by simp
 have finU: finite U
 proof −
  have U ⊆ path-k-V k
   using U-eq unfolding pk.bound-tree-def by auto
  then show ?thesis using path-k-finite-V by (rule finite-subset)
 qed
 have ge: Suc k ≤ card U
  using card-mono[OF finU sub] by (simp add: path-k-card-V)
 have U-base: U = pk.base-case-vertices kk 0 B
  using run by (cases rule: bpi.charged-direct-insert-costed-bmssp-zeroE) auto
 have card U ≤ kk
  using U-base pk.card-base-case-vertices-le by metis
 with ge kk-le show False by simp
qed
```

Frontier persistence (the undone band-monotonicity invariant, on the charged cost relation). On the source-pivot path a partition loop whose current data structure already excludes the source *0* but is still nonempty can never finish: the only pullable pivot is *0* (every *split-below d* $\{0\}$ lies in $\{0\}$), so each remaining step is a null pull, and the relaxation only ever pushes strictly positive frontier vertices (*path-k-relax-in-bound-keys-pos*, *path-k-relax-between-keys-pos*). Hence the nonempty, source-free frontier is preserved forever and neither *Done* (empty keys) nor *Stop* (cap threshold, excluded by *Suc k* < *kk* · *cap*) can fire.

```
lemma path-k-loop-frontier-persist:
 bpi.charged-direct-insert-costed-partition-loop-state Δ M-of t h kk cap l d P B d'
    D a betas bs B' charged-Us child-outputs U c child-costs ⟹
   P = {0} ⟹ Suc k < kk * cap ⟹ (0::nat) ∉ keys-of D ⟹ keys-of D ≠ {}
   ⟹ False
 and bpi.charged-direct-insert-costed-bmssp Δ M-of t h kk cap l d S B d' B' U c
   ⟹ True
proof (induction rule:
   bpi.charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-
bmssp.inducts)
 case (Charged-Direct-Insert-State-Done D a B d' P Δ M-of t h kk cap l d)
 then show ?case by simp
next
 case (Charged-Direct-Insert-State-Stop a B d' P kk cap Δ M-of t h l d D)
 have thr: kk * cap ≤ card (pk.bound-tree P (Fin a))
  using Charged-Direct-Insert-State-Stop by blast
 have card-le: card (pk.bound-tree P (Fin a)) ≤ Suc k
 proof −
  have pk.bound-tree P (Fin a) ⊆ path-k-V k
```

```
    unfolding pk.bound-tree-def by auto
  then show ?thesis
    using card-mono[OF path-k-finite-V] by (simp add: path-k-card-V)
 qed
 show ?case
  using thr card-le Charged-Direct-Insert-State-Stop.prems(2) by linarith
next
 case (Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d
    P a b B' d' kk cap Δ t h d-child U-child c-child charged-child
    direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
    c-direct c-lower c-sources betas bs charged-tail child-outputs-tail
    U-tail c-tail child-costs-tail c)
 have P0: P = {0} using Charged-Direct-Insert-State-Step.prems(1) .
 have regime: Suc k < kk * cap using Charged-Direct-Insert-State-Step.prems(2) .
 have notin: (0::nat) ∉ keys-of D using Charged-Direct-Insert-State-Step.prems(3)
.
 have ne: keys-of D ≠ {} using Charged-Direct-Insert-State-Step.prems(4) .
 have split: S-pull = bpi.split-below d P beta
  using Charged-Direct-Insert-State-Step by metis
 have spull-sub: S-pull ⊆ {0}
  using split P0 unfolding bpi.split-below-def by auto
 have pull: pull-separates D (M-of l) Bmax S-pull beta D-pull
  using Charged-Direct-Insert-State-Step by metis
 have spull-keys: S-pull ⊆ keys-of D
  using pull unfolding pull-separates-def by simp
 have spull-empty: S-pull = {}
  using spull-sub spull-keys notin by blast
 have dpull-keys: keys-of D-pull = keys-of D
  using pull spull-empty unfolding pull-separates-def by simp
 have batch-eq: batch = direct-edge-batch @ lower-edge-batch @ source-batch
  using Charged-Direct-Insert-State-Step by metis
 have dir-eq: direct-edge-batch
    = bpi.edge-relaxation-pairs-in-bound d-child charged-child beta B
  using Charged-Direct-Insert-State-Step by metis
 have low-eq: lower-edge-batch
    = bpi.edge-relaxation-pairs-between d-child charged-child b beta
  using Charged-Direct-Insert-State-Step by metis
 have src-eq: source-batch = bpi.label-pairs-between d S-pull b beta
  using Charged-Direct-Insert-State-Step by metis
 have src-empty: source-batch = []
 proof −
  have fin: finite (keys-of (bpi.label-partition-view d {})) by simp
  have set (partition-key-order (bpi.label-partition-view d {})) = {}
    using partition-key-order-properties(1)[OF fin] by simp
  then have partition-key-order (bpi.label-partition-view d {}) = []
    by simp
  then show ?thesis
    using src-eq spull-empty by (simp add: bpi.label-pairs-between-def)
 qed
```

```
  have batch-pos: (0::nat) ∉ fst ‘ set batch
  proof −
    have fst ‘ set batch
        = fst ‘ set direct-edge-batch ∪ fst ‘ set lower-edge-batch
      using batch-eq src-empty by (simp add: image-Un)
    then show ?thesis
      using dir-eq low-eq
        path-k-relax-in-bound-keys-pos path-k-relax-between-keys-pos by auto
  qed
  have Dnext: D-next = batch-min-update D-pull batch
    using Charged-Direct-Insert-State-Step by metis
  have keys-next: keys-of D-next = keys-of D ∪ fst ‘ set batch
    using Dnext dpull-keys by (simp add: batch-min-update-keys)
  have notin-next: (0::nat) ∉ keys-of D-next
    using keys-next notin batch-pos by simp
  have ne-next: keys-of D-next ≠ {}
    using keys-next ne by simp
  show ?case
    using Charged-Direct-Insert-State-Step notin-next ne-next by metis
next
  case (Charged-Direct-Insert-Base S x Δ M-of t h kk cap d B)
  then show ?case by simp
next
  case (Charged-Direct-Insert-Step D kk cap d S B c-insert t Δ M-of h l d′ a
      betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c)
  then show ?case by simp
qed

end
```

10.5 The bucket cap dominates the path size at the inflated schedule

```
abbreviation cap-at :: nat ⇒ nat where
  cap-at N ≡ bmssp-level-cap (sssp-log-one-third-param N)
    (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)

lemma param-one-third-ge-factor:
  sssp-log-factor-one-third N ≤ real (sssp-log-one-third-param N)
proof −
  have (0::real) ≤ sssp-log-factor-one-third N
    by (rule less-imp-le[OF sssp-log-factor-one-third-pos])
  then show ?thesis
    by (simp add: sssp-log-one-third-param-def)
qed

lemma param-two-thirds-ge-factor:
  sssp-log-factor N ≤ real (sssp-log-two-thirds-param N)
proof −
```

```
  have (0::real) ≤ sssp-log-factor N
    by (rule less-imp-le[OF sssp-log-factor-pos])
  then show ?thesis
    by (simp add: sssp-log-two-thirds-param-def)
qed

lemma ln-le-param-product:
  ln (real N + 2)
    ≤ real (sssp-log-one-third-param N) * real (sssp-log-two-thirds-param N)
proof −
  have lnnn: 0 ≤ ln (real N + 2)
    by simp
  have prod-eq:
    sssp-log-factor-one-third N * sssp-log-factor N = ln (real N + 2)
  proof −
    have sssp-log-factor-one-third N * sssp-log-factor N
        = (ln (real N + 2)) powr (1 / 3) * (ln (real N + 2)) powr (2 / 3)
      by (simp add: sssp-log-factor-one-third-def sssp-log-factor-def)
    also have ... = (ln (real N + 2)) powr (1 / 3 + 2 / 3)
      by (rule powr-add[symmetric])
    also have ... = ln (real N + 2)
      using lnnn by simp
    finally show ?thesis .
  qed
  have f23-nn: 0 ≤ sssp-log-factor N
    by (rule less-imp-le[OF sssp-log-factor-pos])
  have p-nn: 0 ≤ real (sssp-log-one-third-param N)
    by simp
  have sssp-log-factor-one-third N * sssp-log-factor N
      ≤ real (sssp-log-one-third-param N) * real (sssp-log-two-thirds-param N)
    by (rule mult-mono[OF param-one-third-ge-factor param-two-thirds-ge-factor
        p-nn f23-nn])
  then show ?thesis
    using prod-eq by simp
qed

lemma powr-ln2-le-cap:
  (real N + 2) powr ln 2 ≤ real (cap-at N)
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  have cap-eq: cap-at N = ?p * 2 ^ (?p * ?q)
    by (rule bmssp-level-cap-eq)
  have rp1: (1::real) ≤ real ?p
  proof −
    have (1::nat) ≤ ?p
      using sssp-log-one-third-param-pos[of N] by linarith
    thus ?thesis by simp
  qed
```

```
  have ln-le: ln (real N + 2) ≤ real (?p * ?q)
    using ln-le-param-product[of N] by simp
  have (real N + 2) powr ln 2 = 2 powr ln (real N + 2)
    by (simp add: powr-def mult.commute)
  also have ... ≤ 2 powr real (?p * ?q)
    by (rule powr-mono[OF ln-le]) simp
  also have ... = (2::real) ^ (?p * ?q)
    by (rule powr-realpow[OF zero-less-numeral])
  also have ... = 1 * (2::real) ^ (?p * ?q)
    by simp
  also have ... ≤ real ?p * (2::real) ^ (?p * ?q)
    by (rule mult-right-mono[OF rp1]) simp
  also have ... = real (cap-at N)
    by (simp add: cap-eq)
  finally show ?thesis .
qed
```

The decisive arithmetic fact (milestone **B0**, deliverable (G2)): at the inflated schedule $n \cdot n$ the top-level bucket cap eventually covers all *Suc n* path vertices.

```
theorem eventually-card-path-le-cap-inflated:
  eventually (λn. card (path-k-V n) ≤ cap-at (n * n)) at-top
proof −
  have growth:
    eventually (λn::nat. real (Suc n) ≤ (real n) powr (2 * ln 2)) at-top
    by (rule real-Suc-le-powr-eventually[OF one-less-two-ln-two])
  have ev-n: eventually (λn::nat. (1::nat) ≤ n) at-top
    by (rule eventually-ge-at-top)
  from growth ev-n show ?thesis
  proof eventually-elim
    case (elim n)
    have npos: 0 < real n
      using elim(2) by simp
    have sq: real (n * n) = (real n) powr 2
    proof −
      have (real n) powr 2 = (real n) ^ 2
        by (simp add: powr-numeral)
      then show ?thesis
        by (simp add: power2-eq-square)
    qed
    have real (Suc n) ≤ (real n) powr (2 * ln 2)
      by (rule elim(1))
    also have (real n) powr (2 * ln 2) = ((real n) powr 2) powr ln 2
      by (rule powr-powr[symmetric])
    also have ... = real (n * n) powr ln 2
      by (simp only: sq)
    also have real (n * n) powr ln 2 ≤ (real (n * n) + 2) powr ln 2
    proof (rule powr-mono2)
      show 0 ≤ ln (2::real)
```

```
      using ln-gt-zero[of 2] by simp
    show 0 ≤ real (n * n)
      by simp
    show real (n * n) ≤ real (n * n) + 2
      by simp
  qed
  also have ... ≤ real (cap-at (n * n))
    using powr-ln2-le-cap[of n * n] by simp
  finally have real (Suc n) ≤ real (cap-at (n * n)) .
  then have Suc n ≤ cap-at (n * n)
    by (simp only: of-nat-le-iff)
  then show ?case
    by (simp add: path-k-card-V)
 qed
qed
```

10.6 The inflated schedule index stays below the graph size

Growth upper bound (used so that the prefix $\{0..p\}$ of short-tight witnesses fits inside the path): at the inflated schedule $k \cdot k$ the FindPivots round parameter $p = \textit{sssp-log-one-third-param}\ (k \cdot k)$ is eventually at most the path length k, because $p \sim (\ln\ (k^2 + 2))^{1/3}$ grows far slower than k.

```
lemma path-param-le-self:
  eventually (λk::nat. sssp-log-one-third-param (k * k) ≤ k) at-top
proof −
  have ev2: eventually (λk::nat. 2 ≤ k) at-top
    by (rule eventually-ge-at-top)
  show ?thesis
  proof (rule eventually-mono[OF ev2])
    fix k :: nat
    assume k2: 2 ≤ k
    have kpos: 0 < k
      using k2 by simp
    have rk-pos: 0 ≤ real k
      by simp
    have nat-cube: k * k + 1 ≤ k * k * k
    proof −
      have kk1: 1 ≤ k * k
        using kpos by simp
      have k * k + 1 ≤ k * k + k * k
        using kk1 by simp
      also have k * k + k * k = 2 * (k * k)
        by simp
      also have 2 * (k * k) ≤ k * (k * k)
        using k2 by (rule mult-le-mono1)
      also have k * (k * k) = k * k * k
        by (simp add: mult.assoc)
      finally show ?thesis .
    qed
```

have *real-cube*: *real* $(k * k) + 1 \leq (real\ k) \hat{}\ 3$
proof −
have *real* $(k * k) + 1 = real\ (k * k + 1)$
by *simp*
also have $\ldots \leq real\ (k * k * k)$
using *nat-cube* **by** (*simp only*: *of-nat-le-iff*)
also have *real* $(k * k * k) = (real\ k) \hat{}\ 3$
by (*simp add*: *power3-eq-cube*)
finally show *?thesis* **.**
qed
have *ln-le*: *ln* $(real\ (k * k) + 2) \leq (real\ k) \hat{}\ 3$
proof −
have *pos*: $0 < real\ (k * k) + 2$
by (*rule add-nonneg-pos*) *simp-all*
have *ln* $(real\ (k * k) + 2) \leq (real\ (k * k) + 2) - 1$
by (*rule ln-le-minus-one*[*OF pos*])
also have $(real\ (k * k) + 2) - 1 = real\ (k * k) + 1$
by *simp*
also have *real* $(k * k) + 1 \leq (real\ k) \hat{}\ 3$
by (*rule real-cube*)
finally show *?thesis* **.**
qed
have *ln-nonneg*: $0 \leq ln\ (real\ (k * k) + 2)$
proof −
have $(0::real) \leq real\ (k * k)$
by *simp*
then have $(1::real) < real\ (k * k) + 2$
by *linarith*
then have $0 < ln\ (real\ (k * k) + 2)$
by (*rule ln-gt-zero*)
then show *?thesis* **by** *simp*
qed
have *factor-le*: *sssp-log-factor-one-third* $(k * k) \leq real\ k$
proof −
have *sssp-log-factor-one-third* $(k * k)$
$= ln\ (real\ (k * k) + 2)\ powr\ (1\ /\ 3)$
by (*simp add*: *sssp-log-factor-one-third-def*)
also have $\ldots = root\ 3\ (ln\ (real\ (k * k) + 2))$
using *root-powr-inverse*[*of 3 ln* $(real\ (k * k) + 2)$] *ln-nonneg*
by *simp*
also have $\ldots \leq root\ 3\ ((real\ k) \hat{}\ 3)$
using *real-root-le-mono*[*of 3 ln* $(real\ (k * k) + 2)\ (real\ k) \hat{}\ 3$]
ln-le **by** *simp*
also have *root 3* $((real\ k) \hat{}\ 3) = real\ k$
using *real-root-power-cancel*[*of 3 real k*] *rk-pos* **by** *simp*
finally show *?thesis* **.**
qed
have *sssp-log-one-third-param* $(k * k)$
$= nat\ \lceil sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ (k * k) \rceil$

by (*simp add*: *sssp-log-one-third-param-def*)
also have $\ldots \leq$ *nat* $\lceil$*real k*$\rceil$
using *factor-le* **by** (*intro nat-mono ceiling-mono*)
also have *nat* $\lceil$*real k*$\rceil = k$
by *simp*
finally show *sssp-log-one-third-param* $(k * k) \leq k$ **.**
qed
qed

10.7 The inflated schedule is genuine and keeps the polylog in $\Theta(ln\ n)$

lemma *nat-le-mult-self*: $(n::nat) \leq n * n$
by (*cases n*) *auto*

theorem *inflated-schedule-at-top*:
filterlim $(\lambda n::nat.\ n * n)$ *at-top at-top*
unfolding *filterlim-at-top*
proof
fix $Z :: nat$
have *eventually* $(\lambda n::nat.\ Z \leq n)$ *at-top*
by (*rule eventually-ge-at-top*)
then show *eventually* $(\lambda n::nat.\ Z \leq n * n)$ *at-top*
proof *eventually-elim*
case (*elim n*)
show *?case*
using *elim nat-le-mult-self*[*of n*] **by** *linarith*
qed
qed

theorem *ln-inflated-schedule*:
assumes $1 \leq n$
shows *ln* (*real* $(n * n)$) $= 2 *$ *ln* (*real n*)
proof −
have $0 <$ *real n*
using *assms* **by** *simp*
then show *?thesis*
by (*simp add*: *ln-mult*)
qed

Deliverable (G1): the graph size itself is unbounded.

theorem *path-card-V-at-top*:
filterlim $(\lambda n.$ *card* (*path-k-V n*)) *at-top at-top*
proof −
have *filterlim* $(\lambda n::nat.$ *Suc n*) *at-top at-top*
by (*simp add*: *filterlim-Suc*)
then show *?thesis*
by (*simp add*: *path-k-card-V*)
qed

The inflated-schedule pivot regime (B1 staging). Three facts hold together for all but finitely many n: the FindPivots round count $p =$ *sssp-log-one-third-param* $(n \cdot n)$ has grown past *1* (so the level chain $p - 1 \geq 1$ is non-degenerate), it stays below the path length n, and the inflated bucket cap covers the *Suc n* path vertices. These are exactly the hypotheses $p \leq k$, *Suc* $k \leq$ *cap*, $0 < p$ of the path cover bricks at $k = n$, packaged as a single eventual statement to feed the (G3) reduction.

theorem *eventually-inflated-pivot-regime*:
eventually (λ*n*. *2* ≤ *sssp-log-one-third-param* (*n* ∗ *n*)
∧ *sssp-log-one-third-param* (*n* ∗ *n*) ≤ *n*
∧ *Suc n* ≤ *cap-at* (*n* ∗ *n*)) *at-top*
proof −
have *p2*: *eventually* (λ*n*::*nat*. *2* ≤ *sssp-log-one-third-param* (*n* ∗ *n*)) *at-top*
proof −
have *filterlim* (λ*n*::*nat*. *sssp-log-one-third-param* (*n* ∗ *n*)) *at-top at-top*
by (*rule filterlim-compose*[*OF sssp-log-one-third-param-at-top inflated-schedule-at-top*])
then show *?thesis*
unfolding *filterlim-at-top* **by** *blast*
qed
have *ple*: *eventually* (λ*k*::*nat*. *sssp-log-one-third-param* (*k* ∗ *k*) ≤ *k*) *at-top*
by (*rule path-param-le-self*)
have *cap*: *eventually* (λ*n*. *Suc n* ≤ *cap-at* (*n* ∗ *n*)) *at-top*
using *eventually-card-path-le-cap-inflated* **by** (*simp add*: *path-k-card-V*)
from *p2 ple cap* **show** *?thesis*
by *eventually-elim simp*
qed

10.8 The decoupled charged running time of the path family

definition *path-nonvac-time* :: *nat* ⇒ *nat* **where**
path-nonvac-time n =
strict-tie-breaking-digraph.charged-direct-insert-top-level-time
(*path-k-V n*) (*path-k-E n*) (*path-k-weight n*) *0 1* (*n* ∗ *n*)

The decoupled charged reduction (milestone **B1**, deliverable (G3)): whenever the source pivot finishes on P_n at the inflated schedule $n \cdot n$, a charged top-level run — hence a charged top-level cost — exists. This is the schedule-decoupled analogue of *eventually-path-family-charged-direct-insert-top-level-cost-if-source-pivot-finishes*.

theorem *eventually-path-nonvac-charged-cost-exists-if-source-pivot-finishes*:
assumes *source-pivot-finishes*:
eventually
(λ*n*. *strict-tie-breaking-digraph.charged-direct-insert-source-pivot-finishes*
(*path-k-V n*) (*path-k-E n*) (*path-k-weight n*) *0 1* (*n* ∗ *n*)) *at-top*
shows *eventually*
(λ*n*. ∃ *c*. *strict-tie-breaking-digraph.charged-direct-insert-top-level-cost*
(*path-k-V n*) (*path-k-E n*) (*path-k-weight n*) *0 1* (*n* ∗ *n*) *c*) *at-top*

```
  using source-pivot-finishes
proof eventually-elim
  case (elim n)
  interpret bpi: bounded-reduced-positive-instance
    path-k-V n path-k-E n path-k-weight n 0 1
    by (rule path-k-bounded-instance)
  have bpi.charged-direct-insert-source-pivot-finishes 1 (n * n)
    using elim .
  then show ?case
    by (rule bpi.charged-direct-insert-top-level-cost-exists-if-source-pivot-finishes-
pred)
qed
```

Milestone B1, fully discharged. Dropping the *source-pivot-finishes* hypothesis: at the inflated schedule the genuine $2 \leq p \leq n$ pivot regime holds eventually (*eventually-inflated-pivot-regime*), so the unconditional charged-cost existence *path-k-charged-cost-exists* applies for all but finitely many n. A charged top-level run on the decoupled family therefore exists eventually, with *no* assumption.

```
theorem eventually-path-nonvac-charged-cost-exists:
  eventually
    (λn. ∃ c. strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
      (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c) at-top
  using eventually-inflated-pivot-regime
proof eventually-elim
  case (elim n)
  show ?case
  proof (rule path-k-charged-cost-exists)
    show sssp-log-one-third-param (n * n) ≤ n using elim by simp
    show Suc n ≤ bmssp-level-cap (sssp-log-one-third-param (n * n))
        (sssp-log-two-thirds-param (n * n)) (sssp-log-one-third-param (n * n))
      using elim by simp
    show 2 ≤ sssp-log-one-third-param (n * n) using elim by simp
  qed
qed
```

Status of the unconditional discharge of *source-pivot-finishes* (G3). The reduction above turns (G3) into the obligation

- *eventually* (λ*n*. *charged-direct-insert-source-pivot-finishes* (P_n) *0 1* (*n* · *n*)) *at-top*. Its first conjunct $P = \{s\}$ is supplied by the reusable brick *unique-shortest-digraph.find-pivots-pivots-capped-singleton-source-if-large-tree*, whose two premises specialise to the path as follows. The genuine distance *path-k-dist-eq* and the source-tree identity *path-k-tree-of-source* collapse the source-tree slice to the capped scan, and the no-overflow premise is *path-k-find-pivots-seen-card-le* together with the cap domination below. What remains for $P = \{s\}$ is the seen *lower* bound $p \leq$ *card* (*find-pivots-seen-capped* ...), i.e. that the p-round

bounded scan from $\{0\}$ reaches at least p path vertices; with *path-k-dist-eq* every prefix edge is tight, so the prefix $\{0..p\}$ are short-tight witnesses inside the scan. The remaining conjuncts of *source-pivot-finishes* (the band child run and the empty direct-edge batch) are then a multi-step loop-coverage construction that pulls the path band by band, each child closed by the empty-pivots route on a sub-band that fits below p, and is the substance of milestone **B1**.

10.9 The decoupled charged headline (conditional skeleton for B3)

Mirroring the diagonal conditional *bmssp-path-family-charged-direct-insert-runtime-bigo-size-if-cost-bounded*, but on the *decoupled* family P_n run at schedule $n \cdot n$: if charged top-level runs eventually exist and every charged cost is bounded by the refined graph-time budget at the inflated schedule, the least charged cost has the genuine size-parametric target $O(n \cdot (ln\ n)^{2/3})$. Unlike the diagonal version, the two hypotheses here are *not* obstructed: the cap covers the path (see *eventually-card-path-le-cap-inflated*), so the goal of milestones **B1**/**B2** is to discharge them unconditionally.

theorem *path-nonvac-runtime-bigo-if-runs-and-cost-bound*:
  assumes *charged-runs-exist*:
    eventually
      ($\lambda n.\ \exists c.$ *strict-tie-breaking-digraph.charged-direct-insert-top-level-cost*
        (*path-k-V n*) (*path-k-E n*) (*path-k-weight n*) *0 1* (*n* $*$ *n*) *c*) *at-top*
    and *charged-cost-bound*:
    eventually
      ($\lambda n.\ \forall c.$
        strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
          (*path-k-V n*) (*path-k-E n*) (*path-k-weight n*) *0 1* (*n* $*$ *n*) *c* $\longrightarrow$
        c $\leq$ *bmssp-refined-graph-time-bound*
          (λ-. *Suc 1* $*$ *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-. *n*) (*Suc n*)) *at-top*
  shows ($\lambda n.$ *real* (*path-nonvac-time n*))
    $\in$ *O*($\lambda n.$ *real n* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
proof −
  have ($\lambda n.$ *real* (*path-nonvac-time n*))
      $\in$ *O*($\lambda n.$ *sssp-time-target* ($\lambda n.$ *n*) *n*)
  proof (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg-slack*
      [**where** *D* = *1* **and** *Cn* = *2* **and** *Cm* = *1*
        and *v* = $\lambda n.$ *Suc n* **and** *m′* = $\lambda n.$ *n* **and** *m* = $\lambda n.$ *n*])
    show *0* < (*2* :: *real*) **by** *simp*

```
    show 0 < (1 :: real) by simp
    show eventually (λn. real (Suc n) ≤ 2 * real n) at-top
      by (rule eventually-at-top-linorderI[of 1]) simp
    show eventually (λn. real n ≤ 1 * real n) at-top
      by simp
    show eventually
        (λn. path-nonvac-time n ≤
          bmssp-refined-graph-time-bound
            (λ-. Suc 1 * 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-. n) (Suc n)) at-top
      using charged-runs-exist charged-cost-bound
    proof eventually-elim
      case (elim n)
      then obtain c where cost:
        strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
          (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c
        by blast
      interpret bpi: bounded-reduced-positive-instance
        path-k-V n path-k-E n path-k-weight n 0 1
        by (rule path-k-bounded-instance)
      have cost': bpi.charged-direct-insert-top-level-cost 1 (n * n) c
        using cost .
      have time-le-c: bpi.charged-direct-insert-top-level-time 1 (n * n) ≤ c
        unfolding bpi.charged-direct-insert-top-level-time-def
        by (rule Least-le) (rule cost')
      have c-le:
        c ≤ bmssp-refined-graph-time-bound
          (λ-. Suc 1 * 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-. n) (Suc n)
        using elim(2) cost by blast
      have tb:
        bpi.charged-direct-insert-top-level-time 1 (n * n) = path-nonvac-time n
        by (simp add: path-nonvac-time-def)
      show ?case
        using time-le-c c-le tb by linarith
    qed
  qed
  then show ?thesis
    unfolding sssp-time-target-def sssp-log-factor-def by simp
qed
```

The same conditional headline phrased in deliverable (G5)'s exact form,

in the *genuine vertex count card* (*path-k-V n*). This is the precise statement that milestones **B1**/**B2** must make unconditional.

```
theorem path-nonvac-runtime-bigo-card-V-if-runs-and-cost-bound:
  assumes charged-runs-exist:
    eventually
      (λn. ∃ c. strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
        (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c) at-top
    and charged-cost-bound:
    eventually
      (λn. ∀ c.
        strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
          (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c ⟶
        c ≤ bmssp-refined-graph-time-bound
          (λ-. Suc 1 * 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-. n) (Suc n)) at-top
  shows (λn. real (path-nonvac-time n))
    ∈ O(λn. real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
proof −
  have base: (λn. real (path-nonvac-time n))
    ∈ O(λn. real n * (ln (real n + 2)) powr (2 / 3))
    by (rule path-nonvac-runtime-bigo-if-runs-and-cost-bound
      [OF charged-runs-exist charged-cost-bound])
  have dom: (λn::nat. real n * (ln (real n + 2)) powr (2 / 3))
    ∈ O(λn. real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
  proof (rule landau-o.big-mono)
    show eventually
      (λn::nat. norm (real n * (ln (real n + 2)) powr (2 / 3)) ≤
        norm (real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))) at-top
    proof (intro always-eventually allI)
      fix n :: nat
      have card-eq: card (path-k-V n) = Suc n
        by (rule path-k-card-V)
      have f2: (ln (real n + 2)) powr (2 / 3)
          ≤ (ln (real (Suc n) + 2)) powr (2 / 3)
      proof (rule powr-mono2)
        show 0 ≤ (2 / 3 :: real) by simp
        show 0 ≤ ln (real n + 2) by simp
        show ln (real n + 2) ≤ ln (real (Suc n) + 2) by simp
      qed
      have prod-le: real n * (ln (real n + 2)) powr (2 / 3)
          ≤ real (Suc n) * (ln (real (Suc n) + 2)) powr (2 / 3)
      proof (rule mult-mono)
```

```
      show real n ≤ real (Suc n) by simp
      show (ln (real n + 2)) powr (2 / 3)
         ≤ (ln (real (Suc n) + 2)) powr (2 / 3) by (rule f2)
      show 0 ≤ real (Suc n) by simp
      show 0 ≤ (ln (real n + 2)) powr (2 / 3) by simp
    qed
    have f-nonneg: 0 ≤ real n * (ln (real n + 2)) powr (2 / 3)
      by simp
    have g-nonneg: 0 ≤ real (Suc n) * (ln (real (Suc n) + 2)) powr (2 / 3)
      by simp
    show norm (real n * (ln (real n + 2)) powr (2 / 3)) ≤
       norm (real (card (path-k-V n)) *
         (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
      using prod-le f-nonneg g-nonneg by (simp add: card-eq)
  qed
 qed
 show ?thesis
  using base dom by (rule landau-o.big-trans)
qed
```

Factoring the cost hypothesis through the library cost bound (B2 scaffolding). The verbatim *bmssp-refined-graph-time-bound* hypothesis of the headline above is exactly what the proven library theorem *strict-tie-breaking-digraph.charged-direct-insert-top-level-cost-refined-bound-log-params-fixed-degree* delivers from the canonical closed-bound predicate *strict-tie-breaking-digraph.charged-direct-insert-closed-refined-bound-log-params-fixed-degree* on the decoupled family (the path is fully reachable and unit out-degree, so the two side conditions are automatic). Discharging *charged-cost-bound* therefore reduces to the canonical closed-bound predicate at the inflated schedule, which is the single remaining obligation of milestone **B2**.

```
lemma eventually-path-nonvac-cost-bound-if-closed-bound:
 assumes closed-bound:
   eventually
    (λn. strict-tie-breaking-digraph.charged-direct-insert-closed-refined-bound-log-
params-fixed-degree
      (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n)) at-top
 shows eventually
   (λn. ∀ c.
     strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
      (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c ⟶
     c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc 1 * 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-. n) (Suc n)) at-top
```

```
  using closed-bound
proof eventually-elim
  case (elim n)
  interpret bpi: bounded-reduced-positive-instance
    path-k-V n path-k-E n path-k-weight n 0 1
    by (rule path-k-bounded-instance)
  show ?case
  proof (intro allI impI)
    fix c
    assume cost: bpi.charged-direct-insert-top-level-cost 1 (n * n) c
    have c ≤ bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. bpi.edge-count) bpi.vertex-count
      by (rule bpi.charged-direct-insert-top-level-cost-refined-bound-log-params-fixed-degree
        [OF bpi.all-vertices-reachable bpi.bounded-edge-outdegree elim cost])
    then show c ≤ bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. n) (Suc n)
      by (simp add: bpi.edge-count-def bpi.vertex-count-def path-k-card-E path-k-card-V)
  qed
qed
```

The headline phrased against the canonical obligations: top-level runs exist and the closed refined bound holds, both eventually. This trims **B3** to the two library-shaped premises that **B1**/**B2** must make unconditional.

```
theorem path-nonvac-runtime-bigo-card-V-if-runs-and-closed-bound:
  assumes charged-runs-exist:
    eventually
      (λn. ∃c. strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
        (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c) at-top
    and closed-bound:
    eventually
      (λn. strict-tie-breaking-digraph.charged-direct-insert-closed-refined-bound-log-params-fixed-degree
        (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n)) at-top
  shows (λn. real (path-nonvac-time n))
    ∈ O(λn. real (card (path-k-V n)) *
        (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
  by (rule path-nonvac-runtime-bigo-card-V-if-runs-and-cost-bound
```

[*OF charged-runs-exist eventually-path-nonvac-cost-bound-if-closed-bound*[*OF closed-bound*]])

Milestone B1 applied to the headline. The run-existence premise of the headline is now genuinely *discharged* (not merely bypassed by the vacuous least-cost constant): *eventually-path-nonvac-charged-cost-exists* proves a charged top-level run exists for all but finitely many n, so the headline reduces to the single canonical cost obligation *closed-bound* (**B2**). This makes the runtime claim genuinely non-vacuous: the bounded quantity is the cost of a run that provably exists.

theorem *path-nonvac-runtime-bigo-card-V-if-closed-bound-runs-discharged*:
assumes *closed-bound*:
eventually
(λ*n*. *strict-tie-breaking-digraph.charged-direct-insert-closed-refined-bound-log-params-fixed-degree*
(*path-k-V n*) (*path-k-E n*) (*path-k-weight n*) *0 1* (*n* ∗ *n*)) *at-top*
shows (λ*n*. *real* (*path-nonvac-time n*))
∈ *O*(λ*n*. *real* (*card* (*path-k-V n*)) ∗
(*ln* (*real* (*card* (*path-k-V n*)) + *2*)) *powr* (*2* / *3*))
by (*rule path-nonvac-runtime-bigo-card-V-if-runs-and-closed-bound*
[*OF eventually-path-nonvac-charged-cost-exists closed-bound*])

10.10 Dropping the run-existence premise (B1, unconditional in runs)

lemma *Least-le-or-const*:
fixes *P* :: *nat* ⇒ *bool* **and** *B* :: *nat*
assumes ⋀*c*. *P c* ⟹ *c* ≤ *B*
shows (*LEAST c*. *P c*) ≤ *B* + (*LEAST c*::*nat*. ¬ *True*)
proof (*cases* ∃ *c*. *P c*)
case *True*
then obtain *c* **where** *P c* **by** *blast*
then have (*LEAST c*. *P c*) ≤ *c* **by** (*rule Least-le*)
also have *c* ≤ *B* **using** *assms*[*OF* ‹*P c*›] **.**
finally show *?thesis* **by** *simp*
next
case *False*
then have (λ*c*. *P c*) = (λ*c*. ¬ *True*) **by** *auto*
then show *?thesis* **by** *simp*
qed

lemma *path-refined-bound-bigo*:
(λ*n*. *real* (*bmssp-refined-graph-time-bound*
(λ-. *Suc 1* ∗ *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-. *n*) (*Suc n*)))

```
    ∈ O(λn. sssp-time-target (λn. n) n)
  by (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-
square-arg-slack
      [where D = 1 and Cn = 2 and Cm = 1
        and v = λn. Suc n and m' = λn. n and m = λn. n])
     (simp-all add: eventually-at-top-linorderI[of 1])

lemma sssp-time-target-self-ge:
  eventually (λn::nat. real K ≤ sssp-time-target (λn. n) n) at-top
proof (rule eventually-at-top-linorderI[of max 1 K])
  fix n :: nat assume n: max 1 K ≤ n
  have n1: 1 ≤ n and nK: K ≤ n using n by auto
  have lnge: 1 ≤ ln (real n + 2)
  proof −
    have exp (1::real) ≤ 3 by (rule exp-le)
    also have (3::real) ≤ real n + 2 using n1 by simp
    finally have exp (1::real) ≤ real n + 2 .
    then show ?thesis by (simp add: ln-ge-iff)
  qed
  have real K ≤ real n using nK by simp
  also have real n = real n * 1 by simp
  also have ... ≤ real n * (ln (real n + 2)) powr (2/3)
  proof (rule mult-left-mono)
    have (1::real) = 1 powr (2/3) by simp
    also have ... ≤ (ln (real n + 2)) powr (2/3)
      using lnge by (intro powr-mono2) auto
    finally show (1::real) ≤ (ln (real n + 2)) powr (2/3) .
  qed simp
  also have ... = sssp-time-target (λn. n) n
    unfolding sssp-time-target-def sssp-log-factor-def by simp
  finally show real K ≤ sssp-time-target (λn. n) n .
qed

lemma const-bigo-target:
  (λn::nat. real K) ∈ O(λn. sssp-time-target (λn. n) n)
proof (rule landau-o.big-mono, rule eventually-mono[OF sssp-time-target-self-ge[of
K]])
  fix n :: nat assume real K ≤ sssp-time-target (λn. n) n
  then show norm (real K) ≤ norm (sssp-time-target (λn. n) n) by simp
qed
```

Milestone B1, unconditional in run-existence. The charged running time of the decoupled path family is size-parametric O of the SSSP time target as soon as every charged top-level cost is bounded by the refined graph-time budget — with *no* assumption that a charged run exists. When a run exists the least cost is bounded by the budget; when none exists the least-cost is a fixed constant, which the divergent target dominates.

```
theorem path-nonvac-runtime-bigo-if-cost-bound:
  assumes charged-cost-bound:
```

```
    eventually
     (λn. ∀ c.
       strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
        (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c ⟶
       c ≤ bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. n) (Suc n)) at-top
  shows (λn. real (path-nonvac-time n))
    ∈ O(λn. real n * (ln (real n + 2)) powr (2 / 3))
proof −
  have key: (λn. real (path-nonvac-time n))
    ∈ O(λn. sssp-time-target (λn. n) n)
  proof −
    have le: eventually (λn. real (path-nonvac-time n) ≤
      real (bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. n) (Suc n))
      + real (LEAST c::nat. ¬ True)) at-top
      using charged-cost-bound
    proof eventually-elim
      case (elim n)
      interpret bpi: bounded-reduced-positive-instance
        path-k-V n path-k-E n path-k-weight n 0 1
        by (rule path-k-bounded-instance)
      have time-eq: path-nonvac-time n
        = (LEAST c. bpi.charged-direct-insert-top-level-cost 1 (n * n) c)
        by (simp add: path-nonvac-time-def bpi.charged-direct-insert-top-level-time-
def)
      have bound: ⋀c. bpi.charged-direct-insert-top-level-cost 1 (n * n) c ⟹
        c ≤ bmssp-refined-graph-time-bound
          (λ-. Suc 1 * 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-. n) (Suc n)
        using elim by blast
      have (LEAST c. bpi.charged-direct-insert-top-level-cost 1 (n * n) c)
        ≤ bmssp-refined-graph-time-bound
          (λ-. Suc 1 * 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-. n) (Suc n)
      + (LEAST c::nat. ¬ True)
    by (rule Least-le-or-const[OF bound])
  then show ?case using time-eq by simp
 qed
 have sum-bigo: (λn. real (bmssp-refined-graph-time-bound
      (λ-. Suc 1 * 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-. n) (Suc n))
    + real (LEAST c::nat. ¬ True))
    ∈ O(λn. sssp-time-target (λn. n) n)
  using path-refined-bound-bigo const-bigo-target by (rule sum-in-bigo)
 have mono: (λn. real (path-nonvac-time n))
    ∈ O(λn. real (bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. n) (Suc n))
      + real (LEAST c::nat. ¬ True))
 proof (rule landau-o.big-mono)
  show eventually (λn. norm (real (path-nonvac-time n)) ≤
    norm (real (bmssp-refined-graph-time-bound
      (λ-. Suc 1 * 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-. n) (Suc n))
    + real (LEAST c::nat. ¬ True))) at-top
   using le by (auto elim!: eventually-mono)
 qed
 show ?thesis using mono sum-bigo by (rule landau-o.big-trans)
qed
then show ?thesis
 unfolding sssp-time-target-def sssp-log-factor-def by simp
qed
```

The same statement in the genuine vertex count, and reduced to the canonical closed refined bound: both now *without* the run-existence premise.

```
theorem path-nonvac-runtime-bigo-card-V-if-cost-bound:
 assumes charged-cost-bound:
```

```
    eventually
     (λn. ∀ c.
       strict-tie-breaking-digraph.charged-direct-insert-top-level-cost
        (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n) c ⟶
       c ≤ bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. n) (Suc n)) at-top
  shows (λn. real (path-nonvac-time n))
    ∈ O(λn. real (card (path-k-V n)) *
         (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
proof −
  have base: (λn. real (path-nonvac-time n))
    ∈ O(λn. real n * (ln (real n + 2)) powr (2 / 3))
   by (rule path-nonvac-runtime-bigo-if-cost-bound[OF charged-cost-bound])
  have dom: (λn::nat. real n * (ln (real n + 2)) powr (2 / 3))
    ∈ O(λn. real (card (path-k-V n)) *
         (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
  proof (rule landau-o.big-mono)
   show eventually
     (λn::nat. norm (real n * (ln (real n + 2)) powr (2 / 3)) ≤
       norm (real (card (path-k-V n)) *
         (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))) at-top
   proof (intro always-eventually allI)
    fix n :: nat
    have card-eq: card (path-k-V n) = Suc n
     by (rule path-k-card-V)
    have f2: (ln (real n + 2)) powr (2 / 3)
       ≤ (ln (real (Suc n) + 2)) powr (2 / 3)
    proof (rule powr-mono2)
     show 0 ≤ (2 / 3 :: real) by simp
     show 0 ≤ ln (real n + 2) by simp
     show ln (real n + 2) ≤ ln (real (Suc n) + 2) by simp
    qed
    have prod-le: real n * (ln (real n + 2)) powr (2 / 3)
       ≤ real (Suc n) * (ln (real (Suc n) + 2)) powr (2 / 3)
    proof (rule mult-mono)
     show real n ≤ real (Suc n) by simp
     show (ln (real n + 2)) powr (2 / 3)
        ≤ (ln (real (Suc n) + 2)) powr (2 / 3) by (rule f2)
     show 0 ≤ real (Suc n) by simp
     show 0 ≤ (ln (real n + 2)) powr (2 / 3) by simp
    qed
    have f-nonneg: 0 ≤ real n * (ln (real n + 2)) powr (2 / 3)
     by simp
    have g-nonneg: 0 ≤ real (Suc n) * (ln (real (Suc n) + 2)) powr (2 / 3)
```

```
      by simp
    show norm (real n * (ln (real n + 2)) powr (2 / 3)) ≤
       norm (real (card (path-k-V n)) *
         (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
      using prod-le f-nonneg g-nonneg by (simp add: card-eq)
  qed
 qed
 show ?thesis
  using base dom by (rule landau-o.big-trans)
qed

theorem path-nonvac-runtime-bigo-card-V-if-closed-bound:
 assumes closed-bound:
  eventually
   (λn. strict-tie-breaking-digraph.charged-direct-insert-closed-refined-bound-log-
params-fixed-degree
     (path-k-V n) (path-k-E n) (path-k-weight n) 0 1 (n * n)) at-top
 shows (λn. real (path-nonvac-time n))
  ∈ O(λn. real (card (path-k-V n)) *
       (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
 by (rule path-nonvac-runtime-bigo-card-V-if-cost-bound
   [OF eventually-path-nonvac-cost-bound-if-closed-bound[OF closed-bound]])
```

10.11 Charged-cost upper bound via the uncharged refinement (B2, uncharged route)

The decoupled charged running time is bounded by the refined graph-time budget as soon as a single *uncharged* top-level run exists. The charged closed-bound predicate bounds *every* charged run, which on the path is genuinely too strong: the permissive charged loop admits runs at non-zero source bands whose slice accounting breaks band-containment. The headline only needs the *least* charged cost, so it suffices to exhibit one bounded run. Every uncharged run refines to a charged run of equal cost (*unique-shortest-digraph.direct-insert-costed-bmssp-refines-charged-direct-insert*), and the uncharged top-level cost is bounded unconditionally by the proven library bound *strict-tie-breaking-digraph.finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-amortized* (no band-containment side condition). Hence the least charged cost — i.e. the decoupled running time — is bounded by the refined budget whenever an uncharged run exists, reducing **B2** to the genuinely *constructible* obligation of uncharged top-level run existence on the path.

```
context fixes k :: nat
begin
interpretation pk: finite-weighted-digraph
 path-k-V k path-k-E k path-k-weight k 0
 by (rule path-k-finite-weighted-digraph)
```

interpretation *bpi*: *bounded-reduced-positive-instance*
path-k-V k path-k-E k path-k-weight k 0 1
by (*rule path-k-bounded-instance*)

lemma *path-k-uncharged-top-run-imp-charged-cost*:
assumes *run*:
bpi.direct-insert-costed-bmssp (*1*::*nat*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*) (*sssp-log-two-thirds-param N*))
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*)
(*sssp-log-one-third-param N*) (*cap-at N*) (*sssp-log-one-third-param N*)
pk.finite-initial-label {*0*} *Infinity d' Infinity U c*
shows *bpi.charged-direct-insert-top-level-cost 1 N c*
proof −
have *charged*:
bpi.charged-direct-insert-costed-bmssp (*1*::*nat*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*) (*sssp-log-two-thirds-param N*))
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*)
(*sssp-log-one-third-param N*) (*cap-at N*) (*sssp-log-one-third-param N*)
pk.finite-initial-label {*0*} *Infinity d' Infinity U c*
by (*rule bpi.direct-insert-costed-bmssp-refines-charged-direct-insert*[*OF run*])
have *bpi.charged-direct-insert-top-level-run 1 N d' U c*
unfolding *bpi.charged-direct-insert-top-level-run-def*
using *charged* **by** (*simp add*: *Let-def*)
then show *?thesis*
unfolding *bpi.charged-direct-insert-top-level-cost-def* **by** *blast*
qed

lemma *path-k-charged-cost-imp-nonvac-time-le*:
assumes *cost*: *bpi.charged-direct-insert-top-level-cost 1* (*k* ∗ *k*) *c*
shows *path-nonvac-time k* ≤ *c*
proof −
have *bpi.charged-direct-insert-top-level-time 1* (*k* ∗ *k*) ≤ *c*
by (*rule bpi.charged-direct-insert-top-level-time-le*[*OF cost*])
then show *?thesis* **by** (*simp add*: *path-nonvac-time-def*)
qed

lemma *path-k-uncharged-run-cost-bound*:
fixes *N* :: *nat*
assumes *p2*: *2* ≤ *sssp-log-one-third-param N*
and *run*:
bpi.direct-insert-costed-bmssp (*1*::*nat*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*) (*sssp-log-two-thirds-param N*))
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*)
(*sssp-log-one-third-param N*) (*cap-at N*) (*sssp-log-one-third-param N*)
pk.finite-initial-label {*0*} *Infinity d' Infinity U c*
shows *c* ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc 1* ∗ *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-. *k*) (*Suc k*)

proof −
let *?p* = *sssp-log-one-third-param N* **and** *?q* = *sssp-log-two-thirds-param N*
have *ppos*: *0* < *?p* **using** *p2* **by** *simp*
have *qp2*: *?q* ≤ *?p* ∗ *?p* **by** (*rule sssp-log-two-thirds-param-le-one-third-square*)
have *pp2*: *?p* ∗ *?p* ≤ *Suc 1* ∗ *?p* ∗ *?p* **by** *simp*
have *df*: (*1*::*nat*) ≤ *Suc 1* ∗ *?p* **using** *ppos* **by** *presburger*
have *rp*: *0* < *?q* **by** *simp*
have *insf*: *?q* ≤ *Suc 1* ∗ *?p* ∗ *?p* **using** *qp2 pp2* **by** *linarith*
have *seensf*: *?p* ∗ *1* + *Suc 1* ∗ *?p* ≤ *2* ∗ (*Suc 1* ∗ *?p*) **by** *presburger*
have *srcf*: *Suc ?p* ≤ *2* ∗ (*Suc 1* ∗ *?p*) **using** *ppos* **by** *presburger*
note *base* = *bpi.finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-amortized*
[**where** *A* = *Suc 1* ∗ *?p* **and** *R* = *?q* **and** *A-insert* = *Suc 1* ∗ *?p*
and Δ = *1*::*nat* **and** *t* = *?q* **and** *h* = *?p* **and** *k* = *?p* **and** *l* = *?p*,
OF bpi.all-vertices-reachable bpi.bounded-edge-outdegree df rp insf insf seensf srcf ppos run]
show *?thesis*
using *base*
by (*simp add*: *bpi.edge-count-def bpi.vertex-count-def path-k-card-E path-k-card-V*)
qed

lemma *path-k-uncharged-run-imp-nonvac-time-le-bound*:
assumes *p2*: *2* ≤ *sssp-log-one-third-param* (*k* ∗ *k*)
and *run*:
bpi.direct-insert-costed-bmssp (*1*::*nat*)
(*bmssp-level-cap* (*sssp-log-one-third-param* (*k* ∗ *k*)) (*sssp-log-two-thirds-param* (*k* ∗ *k*)))
(*sssp-log-two-thirds-param* (*k* ∗ *k*)) (*sssp-log-one-third-param* (*k* ∗ *k*))
(*sssp-log-one-third-param* (*k* ∗ *k*)) (*cap-at* (*k* ∗ *k*)) (*sssp-log-one-third-param* (*k* ∗ *k*))
pk.finite-initial-label {*0*} *Infinity d′ Infinity U c*
shows *path-nonvac-time k* ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc 1* ∗ *sssp-log-one-third-param* (*k* ∗ *k*)) (λ-. *sssp-log-two-thirds-param* (*k* ∗ *k*))
(λ-. *sssp-log-one-third-param* (*k* ∗ *k*)) (λ-. *sssp-log-one-third-param* (*k* ∗ *k*))
(λ-. *sssp-log-two-thirds-param* (*k* ∗ *k*)) (λ-. *k*) (*Suc k*)
proof −
have *cost*: *bpi.charged-direct-insert-top-level-cost 1* (*k* ∗ *k*) *c*
by (*rule path-k-uncharged-top-run-imp-charged-cost*[*OF run*])
have *tle*: *path-nonvac-time k* ≤ *c*
by (*rule path-k-charged-cost-imp-nonvac-time-le*[*OF cost*])
have *cle*: *c* ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc 1* ∗ *sssp-log-one-third-param* (*k* ∗ *k*)) (λ-. *sssp-log-two-thirds-param* (*k* ∗ *k*))
(λ-. *sssp-log-one-third-param* (*k* ∗ *k*)) (λ-. *sssp-log-one-third-param* (*k* ∗ *k*))
(λ-. *sssp-log-two-thirds-param* (*k* ∗ *k*)) (λ-. *k*) (*Suc k*)
by (*rule path-k-uncharged-run-cost-bound*[*OF p2 run*])
show *?thesis* **using** *tle cle* **by** *linarith*

qed

The cheap charged running time is below the refined graph-time budget. The genuinely *constructible* analogue of *path-k-uncharged-run-imp-nonvac-time-le-bound*: rather than assuming an (in the genuine regime non-existent) uncharged top-level run, we use the cheap charged top-level cost *path-k-charged-cost-le*, whose value $p \cdot (p \cdot \mathit{edge\text{-}count}) + p$ is dominated termwise by the refined budget — the edge factor absorbs $p^2 \cdot \mathit{edge\text{-}count}$ and the vertex factor absorbs the additive p. Hence the least charged cost, i.e. the decoupled running time, is below the refined budget, with *no* run-existence or cost-bound assumption.

lemma *path-k-charged-cheap-imp-nonvac-time-le-bound*:
assumes *pk-le*: *sssp-log-one-third-param* $(k * k) \leq k$
and *cover*: *Suc* $k \leq$ *bmssp-level-cap* (*sssp-log-one-third-param* $(k * k)$)
(*sssp-log-two-thirds-param* $(k * k)$) (*sssp-log-one-third-param* $(k * k)$)
and *p2*: $2 \leq$ *sssp-log-one-third-param* $(k * k)$
shows *path-nonvac-time* $k \leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc 1* $*$ *sssp-log-one-third-param* $(k * k)$) (λ-. *sssp-log-two-thirds-param* $(k * k)$)
(λ-. *sssp-log-one-third-param* $(k * k)$) (λ-. *sssp-log-one-third-param* $(k * k)$)
(λ-. *sssp-log-two-thirds-param* $(k * k)$) (λ-. *k*) (*Suc k*)
proof −
let *?p* = *sssp-log-one-third-param* $(k * k)$
let *?q* = *sssp-log-two-thirds-param* $(k * k)$
obtain *c* **where** *cost*: *bpi.charged-direct-insert-top-level-cost 1* $(k * k)$ *c*
and *cle*: $c \leq$ *?p* $*$ (*?p* $*$ *bpi.edge-count*) + *?p*
using *path-k-charged-cost-le*[*OF pk-le cover p2*] **by** *blast*
have *tle*: *path-nonvac-time* $k \leq c$
by (*rule path-k-charged-cost-imp-nonvac-time-le*[*OF cost*])
have *edge*: *bpi.edge-count* $= k$
by (*simp add*: *bpi.edge-count-def path-k-card-E*)
have *c-le2*: $c \leq$ *?p* $*$ (*?p* $* k$) + *?p* **using** *cle edge* **by** *simp*
have *edge-term*: *?p* $*$ (*?p* $* k$) $\leq$ (*?q* + *?q* + *?p* $*$ *?p*) $* k$
proof −
have *?p* $*$ (*?p* $* k$) = (*?p* $*$ *?p*) $* k$ **by** (*simp add*: *mult.assoc*)
also have ... $\leq$ (*?q* + *?q* + *?p* $*$ *?p*) $* k$ **by** (*rule mult-le-mono1*) *simp*
finally show *?thesis* **.**
qed
have *ge1*: (*1*::*nat*) $\leq$ (*2* $*$ *?p* + *1*) $*$ (*Suc k*)
using *mult-le-mono*[*of 1 2* $*$ *?p* + *1 1 Suc k*] **by** *simp*
have *vert-term*: *?p* $\leq$ *2* $*$ (*Suc 1* $*$ *?p*) $*$ (*2* $*$ *?p* + *1*) $*$ (*Suc k*)
proof −
have *a*: *?p* $\leq$ *2* $*$ (*Suc 1* $*$ *?p*) **by** *simp*
have *b*: *2* $*$ (*Suc 1* $*$ *?p*) $\leq$ *2* $*$ (*Suc 1* $*$ *?p*) $*$ (*2* $*$ *?p* + *1*) $*$ (*Suc k*)
proof −
have *2* $*$ (*Suc 1* $*$ *?p*) $*$ *1*
$\leq$ *2* $*$ (*Suc 1* $*$ *?p*) $*$ ((*2* $*$ *?p* + *1*) $*$ (*Suc k*))
by (*rule mult-le-mono2*[*OF ge1*])

```
    thus ?thesis by (simp add: mult.assoc)
   qed
   from a b show ?thesis by linarith
  qed
  have budget-ge: ?p * (?p * k) + ?p ≤ bmssp-refined-graph-time-bound
     (λ-. Suc 1 * ?p) (λ-. ?q) (λ-. ?p) (λ-. ?p) (λ-. ?q) (λ-. k) (Suc k)
  proof −
   have ?p * (?p * k) + ?p
      ≤ 2 * (Suc 1 * ?p) * (2 * ?p + 1) * (Suc k) + (?q + ?q + ?p * ?p) * k
     using edge-term vert-term by linarith
   also have ... = bmssp-refined-graph-time-bound
      (λ-. Suc 1 * ?p) (λ-. ?q) (λ-. ?p) (λ-. ?p) (λ-. ?q) (λ-. k) (Suc k)
     by (simp add: bmssp-refined-graph-time-bound-def)
   finally show ?thesis .
  qed
  show ?thesis using tle c-le2 budget-ge by linarith
qed

end

theorem path-nonvac-runtime-bigo-card-V-if-time-le-bound:
  assumes tle:
   eventually (λn. path-nonvac-time n ≤ bmssp-refined-graph-time-bound
      (λ-. Suc 1 * 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-. n) (Suc n)) at-top
  shows (λn. real (path-nonvac-time n))
   ∈ O(λn. real (card (path-k-V n)) *
       (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
proof −
  let ?B = λn. bmssp-refined-graph-time-bound
      (λ-. Suc 1 * 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-. n) (Suc n)
  have dom-b: (λn. real (path-nonvac-time n)) ∈ O(λn. real (?B n))
  proof (rule landau-o.big-mono)
   show eventually (λn. norm (real (path-nonvac-time n)) ≤ norm (real (?B n)))
at-top
     using tle by eventually-elim simp
  qed
  have b-target: (λn. real (?B n)) ∈ O(λn. sssp-time-target (λn. n) n)
   by (rule path-refined-bound-bigo)
  have base: (λn. real (path-nonvac-time n)) ∈ O(λn. sssp-time-target (λn. n) n)
   using dom-b b-target by (rule landau-o.big-trans)
  have base': (λn. real (path-nonvac-time n))
   ∈ O(λn. real n * (ln (real n + 2)) powr (2 / 3))
   using base unfolding sssp-time-target-def sssp-log-factor-def by simp
```

```
  have dom: (λn::nat. real n * (ln (real n + 2)) powr (2 / 3))
    ∈ O(λn. real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
  proof (rule landau-o.big-mono)
    show eventually
       (λn::nat. norm (real n * (ln (real n + 2)) powr (2 / 3)) ≤
         norm (real (card (path-k-V n)) *
           (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))) at-top
    proof (intro always-eventually allI)
      fix n :: nat
      have card-eq: card (path-k-V n) = Suc n by (rule path-k-card-V)
      have f2: (ln (real n + 2)) powr (2 / 3) ≤ (ln (real (Suc n) + 2)) powr (2 /
3)
        by (rule powr-mono2) simp-all
      have prod-le: real n * (ln (real n + 2)) powr (2 / 3)
        ≤ real (Suc n) * (ln (real (Suc n) + 2)) powr (2 / 3)
        by (rule mult-mono[OF - f2]) simp-all
      show norm (real n * (ln (real n + 2)) powr (2 / 3)) ≤
        norm (real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
        using prod-le by (simp add: card-eq)
    qed
  qed
  show ?thesis using base' dom by (rule landau-o.big-trans)
qed
```

10.12 The unconditional non-vacuous size-parametric running-time bound

Milestones B1/B2/B3, fully discharged. The decoupled charged running time of the path family is *unconditionally* in the genuine size-parametric class $O(card\ V_n \cdot (ln\ (card\ V_n))^{2/3})$, with *no* run-existence, cost-bound, or closed-bound hypothesis. Combining *eventually-inflated-pivot-regime* (the genuine $2 \le p \le n$ pivot regime where the cap covers the path holds eventually) with *path-k-charged-cheap-imp-nonvac-time-le-bound* (the cheap charged top-level cost lies below the refined graph-time budget) discharges the time hypothesis of *path-nonvac-runtime-bigo-card-V-if-time-le-bound*.

This is the unconditional form of the conditional headline *path-nonvac-runtime-bigo-card-V-if-runs-and-closed-bound*: both of its premises (eventual charged-run existence and the canonical closed refined bound) are now dropped. The running time bounded here is genuinely non-vacuous: it is the *least* charged cost of the decoupled family, a cost of a charged top-level run that provably exists and is provably small.

```
theorem path-nonvac-runtime-bigo-card-V-unconditional:
  (λn. real (path-nonvac-time n))
    ∈ O(λn. real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
```

```
proof (rule path-nonvac-runtime-bigo-card-V-if-time-le-bound)
  show eventually
      (λn. path-nonvac-time n ≤ bmssp-refined-graph-time-bound
        (λ-. Suc 1 * 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-. n) (Suc n)) at-top
    using eventually-inflated-pivot-regime
  proof eventually-elim
    case (elim n)
    show ?case
      by (rule path-k-charged-cheap-imp-nonvac-time-le-bound) (use elim in auto)
  qed
qed
```

The same unconditional bound restated against the two library-shaped premises of the conditional headline, witnessing that both have been discharged: the charged runtime is in the size-parametric class with neither the run-existence premise nor the closed refined bound assumed.

```
theorem path-nonvac-runtime-bigo-card-V-runs-and-closed-bound-discharged:
  (λn. real (path-nonvac-time n))
    ∈ O(λn. real (card (path-k-V n)) *
          (ln (real (card (path-k-V n)) + 2)) powr (2 / 3))
  by (rule path-nonvac-runtime-bigo-card-V-unconditional)
```

```
definition path-uncharged-top-run :: nat ⇒ (nat ⇒ real) ⇒ nat set ⇒ nat ⇒ bool
where
  path-uncharged-top-run n d' U c ⟷
      unique-shortest-digraph.direct-insert-costed-bmssp (path-k-V n) (path-k-E n)
(path-k-weight n) 0
       (1::nat) (bmssp-level-cap (sssp-log-one-third-param (n * n)) (sssp-log-two-
thirds-param (n * n)))
       (sssp-log-two-thirds-param (n * n)) (sssp-log-one-third-param (n * n))
       (sssp-log-one-third-param (n * n)) (cap-at (n * n)) (sssp-log-one-third-param
(n * n))
       (finite-weighted-digraph.finite-initial-label (path-k-V n) (path-k-E n) (path-k-
weight n) 0)
       {0} Infinity d' Infinity U c
```

```
lemma eventually-path-nonvac-time-le-bound-if-uncharged-runs:
  assumes ex: eventually (λn. ∃ d' U c. path-uncharged-top-run n d' U c) at-top
  shows eventually (λn. path-nonvac-time n ≤ bmssp-refined-graph-time-bound
       (λ-. Suc 1 * 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-. n) (Suc n)) at-top
```

using *ex eventually-inflated-pivot-regime*
proof *eventually-elim*
case (*elim n*)
interpret *pk*: *finite-weighted-digraph*
path-k-V n path-k-E n path-k-weight n 0
by (*rule path-k-finite-weighted-digraph*)
interpret *bpi*: *bounded-reduced-positive-instance*
path-k-V n path-k-E n path-k-weight n 0 1
by (*rule path-k-bounded-instance*)
from *elim* **obtain** *d′ U c* **where**
run: *bpi.direct-insert-costed-bmssp* (*1*::*nat*)
(*bmssp-level-cap* (*sssp-log-one-third-param* (*n* ∗ *n*)) (*sssp-log-two-thirds-param* (*n* ∗ *n*)))
(*sssp-log-two-thirds-param* (*n* ∗ *n*)) (*sssp-log-one-third-param* (*n* ∗ *n*))
(*sssp-log-one-third-param* (*n* ∗ *n*)) (*cap-at* (*n* ∗ *n*)) (*sssp-log-one-third-param* (*n* ∗ *n*))
pk.finite-initial-label {*0*} *Infinity d′ Infinity U c*
unfolding *path-uncharged-top-run-def* **by** *blast*
have *p2*: *2* ≤ *sssp-log-one-third-param* (*n* ∗ *n*) **using** *elim* **by** *simp*
show *?case*
by (*rule path-k-uncharged-run-imp-nonvac-time-le-bound*[*OF p2 run*])
qed

theorem *path-nonvac-runtime-bigo-card-V-if-uncharged-runs*:
assumes *ex*: *eventually* (λ*n*. ∃ *d′ U c*. *path-uncharged-top-run n d′ U c*) *at-top*
shows (λ*n*. *real* (*path-nonvac-time n*))
∈ *O*(λ*n*. *real* (*card* (*path-k-V n*)) ∗
(*ln* (*real* (*card* (*path-k-V n*)) + *2*)) *powr* (*2* / *3*))
by (*rule path-nonvac-runtime-bigo-card-V-if-time-le-bound*
[*OF eventually-path-nonvac-time-le-bound-if-uncharged-runs*[*OF ex*]])

10.13 Charged closed-bound engine (B2): the charged level recurrence solver

The charged top-level closed-bound predicate *strict-tie-breaking-digraph.charged-direct-insert-closed-refined-bound-log-params-fixed-degree* bounds *every* charged top-level run. The uncharged analogue is already solved by *strict-tie-breaking-digraph.direct-insert-costed-bmssp-level-bound-from-local-budgets-with-invariants-and-edge-budget*; here we build the charged analogue, threading the charged per-step closer *strict-tie-breaking-digraph.charged-direct-insert-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants-and-edge-budget* through the recursion. The two semantic obligations the charged closer carries — the loop edge budget and the source-progress (no-stranding) invariant — are discharged generically here (edge budget) and on the path below (source progress).

context *strict-tie-breaking-digraph*
begin

Charged analogue of *direct-insert-costed-partition-loop-state-trivial-edge-budget*: the separated direct/lower edge-range sums of a charged loop step are dominated by the outgoing-edge count of the produced loop output, with combined factor $t + h$.

lemma *charged-direct-insert-costed-partition-loop-state-trivial-edge-budget*:
assumes *run*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a betas bs B' charged-Us child-outputs 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 −
have *mono*: *nondecreasing-from a bs*
by (*rule charged-direct-insert-costed-partition-loop-state-mono*[*OF run*])
have *trace*: *concrete-partition-loop-trace P B a bs d' B' charged-Us U*
by (*rule charged-direct-insert-costed-partition-loop-state-trace*
[*OF run sound pre P-reaches*])
have *children*:
list-all2 ($\lambda U\ X.\ U = X \wedge$ *complete-on d' U*) *charged-Us*
(*range-tree-chain-list P a bs B'*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *U-def*: *U* = *bound-tree P* (*Fin a*) $\cup \bigcup$(*set charged-Us*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *Union-eq*:
$\bigcup$(*set charged-Us*) = $\bigcup$(*set* (*range-tree-chain-list P a bs B'*))
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
U = *bound-tree P* (*Fin a*) $\cup$ *range-tree-chain P a bs B'*
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B'*] **by** *simp*
have *chain-subset*: *range-tree-chain P a bs B'* $\subseteq$ *U*
using *U-eq-chain* **by** *blast*
have *outgoing-subset*:
outgoing-edges (*range-tree-chain P a bs B'*) $\subseteq$ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *card-out-le*:
card (*outgoing-edges* (*range-tree-chain P a bs B'*)) $\leq$
card (*outgoing-edges U*)
by (*rule card-mono*) (*simp-all add*: *outgoing-subset*)
have *sum-le-chain*:
$h *$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
$t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*))
$\leq (h + t) *$ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*))
by (*rule weighted-sum-child-lower-direct-edge-ranges-le-outgoing-edges-chain*
[*OF mono*])

```
  have chain-to-U:
    (h + t) * card (outgoing-edges (range-tree-chain P a bs B')) ≤
      (h + t) * card (outgoing-edges U)
    using card-out-le by simp
  show ?thesis
    using sum-le-chain chain-to-U by (simp add: algebra-simps)
qed
```

Per-child amortized output → slice converter (B2 engine). The charged amortized loop closer *charged-direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs* consumes per-child amortized bounds stated against the *disjoint range-tree slices*. A recursion-level induction instead produces per-child bounds against the child *outputs*. Whenever each output sits inside its slice (the band-containment maintained by the descending recursion), the amortized budget's monotonicity *bmssp-amortized-cost-bound-mono* lifts the output bounds to the slice bounds the closer needs.

```
lemma list-all2-bmssp-amortized-cost-bound-lift-subset:
  assumes outb:
    list-all2 (λc-child UB.
        c-child ≤ bmssp-amortized-cost-bound A R h t q L (snd UB) (fst UB))
      child-costs output-pairs
    and contain:
    list-all2 (λOB SB.
        fst OB ⊆ fst SB ∧ snd OB = snd SB ∧ finite (fst SB))
      output-pairs slice-pairs
  shows list-all2 (λc-child UB.
      c-child ≤ bmssp-amortized-cost-bound A R h t q L (snd UB) (fst UB))
    child-costs slice-pairs
proof (rule list-all2-all-nthI)
  show length child-costs = length slice-pairs
    using outb contain by (simp add: list-all2-lengthD)
next
  fix i assume i: i < length child-costs
  have i2: i < length output-pairs
    using outb i by (simp add: list-all2-lengthD)
  have c-le:
    child-costs ! i ≤
      bmssp-amortized-cost-bound A R h t q L
        (snd (output-pairs ! i)) (fst (output-pairs ! i))
    using list-all2-nthD[OF outb i] by simp
  have cont:
    fst (output-pairs ! i) ⊆ fst (slice-pairs ! i) ∧
      snd (output-pairs ! i) = snd (slice-pairs ! i) ∧
      finite (fst (slice-pairs ! i))
    using list-all2-nthD[OF contain i2] by simp
  have bmssp-amortized-cost-bound A R h t q L
        (snd (output-pairs ! i)) (fst (output-pairs ! i))
      ≤ bmssp-amortized-cost-bound A R h t q L
        (snd (slice-pairs ! i)) (fst (slice-pairs ! i))
```

using *cont bmssp-amortized-cost-bound-mono*[*of fst* (*output-pairs* ! *i*)
fst (*slice-pairs* ! *i*) *A R h t q L snd* (*slice-pairs* ! *i*)]
by *auto*
with *c-le* **show**
child-costs ! *i* ≤
bmssp-amortized-cost-bound A R h t q L
(*snd* (*slice-pairs* ! *i*)) (*fst* (*slice-pairs* ! *i*))
by *linarith*
qed

Band containment (B2 engine): a settled child output sits inside its slice. The descending recursion maintains, at a loop step with lower band a, two modular invariants of the working label d over the pivot set S: a *label floor* $a \le d\ x$ and *prefix exactness* below a. Under those a pulled child's whole produced tree *bound-tree S B* lies at or above the band, hence inside the band-range slice *range-tree S a B* — using only *tree-path-dist-le* and the floor/exactness reductions of the note preceding *split-below-inter-bound-tree-empty-if-label-ge-exact*.

lemma *bound-tree-subset-range-tree-if-floor-exact*:
assumes *floor*: ⋀*x*. *x* ∈ *S* ⟹ *a* ≤ *d x*
and *exact*: ⋀*u*. *u* ∈ *S* ⟹ *dist s u* < *a* ⟹ *d u* = *dist s u*
shows *bound-tree S B* ⊆ *range-tree S a B*
proof
fix *v* **assume** *v* ∈ *bound-tree S B*
then have *vtree*: *v* ∈ *tree-set S* **and** *vbelow*: *below-bound* (*dist s v*) *B*
unfolding *bound-tree-eq-tree-set* **by** *auto*
from *vtree* **obtain** *u* **where** *uS*: *u* ∈ *S* **and** *tp*: *tree-path u v*
unfolding *tree-set-def* **by** *blast*
have *du-le*: *dist s u* ≤ *dist s v* **by** (*rule tree-path-dist-le*[*OF tp*])
have *a-le-u*: *a* ≤ *dist s u*
proof (*rule ccontr*)
assume ¬ *a* ≤ *dist s u*
then have *lt*: *dist s u* < *a* **by** *simp*
then have *d u* = *dist s u* **using** *exact*[*OF uS*] **by** *simp*
with *lt* **have** *d u* < *a* **by** *simp*
with *floor*[*OF uS*] **show** *False* **by** *simp*
qed
from *a-le-u du-le* **have** *a* ≤ *dist s v* **by** *simp*
with *vtree vbelow* **show** *v* ∈ *range-tree S a B*
unfolding *range-tree-def* **by** *simp*
qed

lemma *bound-tree-split-below-subset-range-tree-if-floor-exact*:
assumes *floor*: ⋀*x*. *x* ∈ *P* ⟹ *a* ≤ *d x*
and *exact*: ⋀*u*. *u* ∈ *P* ⟹ *dist s u* < *a* ⟹ *d u* = *dist s u*
shows *bound-tree* (*split-below d P beta*) *B* ⊆ *range-tree P a B*
proof (*rule subset-trans*[*OF - range-tree-split-below-subset*])
show *bound-tree* (*split-below d P beta*) *B*
⊆ *range-tree* (*split-below d P beta*) *a B*

proof (*rule bound-tree-subset-range-tree-if-floor-exact*)
fix x **assume** x: $x \in$ *split-below d P beta*
have $x \in P$ **by** (*rule rev-subsetD*[*OF x split-below-subset*])
then show $a \leq d\ x$ **by** (*rule floor*)
next
fix u **assume** u: $u \in$ *split-below d P beta* **and** *ult*: *dist s u* $<$ a
have $u \in P$ **by** (*rule rev-subsetD*[*OF u split-below-subset*])
then show $d\ u =$ *dist s u* **using** *ult* **by** (*rule exact*)
qed
qed

Charged per-child amortized slice-bound propagator (B2 engine). The charged amortized loop closer *charged-direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs* consumes per-child amortized bounds stated against the disjoint range-tree slices *range-tree-child-bound-pair-list P a betas bs*. A recursion-level induction supplies per-child bounds against the child *outputs*; whenever each output sits inside its slice — a *list-all2* band-containment that the descending recursion threads as data (and which is trivial on the source band $a = 0$, where the slice is the full bounded tree) — the amortized budget's monotonicity lifts the output bounds to the slice bounds. Unlike the unusable library closers, the per-child budget here is stated against *outputs*, so it is dischargeable by an ordinary inductive hypothesis.

lemma *charged-child-amortized-slice-bounds-from-output-and-containment*:
charged-direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ charged-Us child-outputs 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′*.
⟦*charged-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* $\leq$ *bmssp-amortized-cost-bound A R h t q L B-child U-child*) $\Longrightarrow$
list-all2 (λ*U-out SB. U-out* $\subseteq$ *fst SB*) *child-outputs*
(*range-tree-child-bound-pair-list P a betas bs*) $\Longrightarrow$
list-all2 (λ*c-child SB*.
case SB 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 *charged-direct-insert-costed-bmssp-child-amortized-slice-bounds-trivial*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
proof (*induction rule*:

charged-direct-insert-costed-partition-loop-state-charged-direct-insert-costed-bmssp.inducts)
case (*Charged-Direct-Insert-State-Done D a B d′ P Δ M-of t h k cap l d*)
then show *?case* **by** *simp*
next
case (*Charged-Direct-Insert-State-Stop a B d′ P k cap Δ M-of t h l d D*)
then show *?case* **by** *simp*
next
case (*Charged-Direct-Insert-State-Step D M-of l Bmax S-pull beta D-pull B d P a b B′ d′ k cap Δ t h d-child U-child c-child charged-child direct-edge-batch lower-edge-batch source-batch batch D-next c-pull c-direct c-lower c-sources betas bs charged-tail child-outputs-tail U-tail c-tail child-costs-tail c*)
have *contain-unf*:
list-all2 (λ*U-out SB*. *U-out* ⊆ *fst SB*) (*U-child* # *child-outputs-tail*) ((*range-tree P a* (*Fin b*), *Fin beta*) # *range-tree-child-bound-pair-list P b betas bs*)
using *Charged-Direct-Insert-State-Step.prems* **by** *simp*
have *contain-head*: *U-child* ⊆ *range-tree P a* (*Fin b*)
using *contain-unf* **by** *simp*
have *contain-tail*:
list-all2 (λ*U-out SB*. *U-out* ⊆ *fst SB*) *child-outputs-tail* (*range-tree-child-bound-pair-list P b betas bs*)
using *contain-unf* **by** *simp*
have *tail*:
list-all2 (λ*c-child SB*. *case SB of* (*U-child*, *B-child*) ⇒ *c-child* ≤ *bmssp-amortized-cost-bound A R h t q L B-child U-child*) *child-costs-tail* (*range-tree-child-bound-pair-list P b betas bs*)
using *Charged-Direct-Insert-State-Step.IH* *Charged-Direct-Insert-State-Step.prems contain-tail* **by** *blast*
have *card-pull*: *card S-pull* ≤ *M-of l*
using *Charged-Direct-Insert-State-Step* **unfolding** *pull-separates-def* **by** *blast*
have *below-pull*: ⋀*x*. *x* ∈ *S-pull* ⟹ *below-bound* (*d x*) (*Fin beta*)
proof −
fix *x* **assume** *xS*: *x* ∈ *S-pull*
have *S-pull* = *split-below d P beta*
using *Charged-Direct-Insert-State-Step* **by** *blast*
then show *below-bound* (*d x*) (*Fin beta*)
using *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *pull-k-cap*: *k* ∗ *card S-pull* ≤ *cap*
proof −
have *S-pull* = *split-below d P beta*
using *Charged-Direct-Insert-State-Step* **by** *blast*
then show *?thesis*
using *split-below-scaled-card-le* [*OF Charged-Direct-Insert-State-Step.prems*(*1*)

```
      Charged-Direct-Insert-State-Step.prems(2), of d beta] by simp
  qed
  have pull-anti: tree-antichain S-pull
  proof −
    have S-pull = split-below d P beta
      using Charged-Direct-Insert-State-Step by blast
    then show ?thesis
      using split-below-tree-antichain
        [OF Charged-Direct-Insert-State-Step.prems(3), of d beta] by simp
  qed
  have child-run:
    charged-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
    using Charged-Direct-Insert-State-Step by blast
  have pre-pull: bmssp-pre-full d S-pull (Fin beta)
    using Charged-Direct-Insert-State-Step by blast
  have reaches-pull: ⋀x. x ∈ S-pull ⟹ reachable s x
    using Charged-Direct-Insert-State-Step by blast
  have head-out:
    c-child ≤ bmssp-amortized-cost-bound A R h t q L (Fin beta) U-child
    by (rule Charged-Direct-Insert-State-Step.prems(4)
      [OF child-run pre-pull reaches-pull card-pull below-pull
        pull-k-cap pull-anti])
  have head:
    c-child ≤
      bmssp-amortized-cost-bound A R h t q L (Fin beta) (range-tree P a (Fin b))
    using head-out
      bmssp-amortized-cost-bound-mono[OF contain-head finite-range-tree,
        of A R h t q L Fin beta]
    by simp
  show ?case
    using head tail by simp
next
  case (Charged-Direct-Insert-Base S x Δ M-of t h k cap d B)
  then show ?case by simp
next
  case (Charged-Direct-Insert-Step D k cap d S B c-insert t Δ M-of h l
      d′ a betas bs B′ charged-Us child-outputs U-loop c-loop child-costs-loop U c)
  then show ?case by simp
qed
```

Charged amortized level recurrence (B2 engine). The charged analogue of *direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-invariants*: a strong-induction on the recursion level closes the charged run cost against the amortized graph budget *bmssp-amortized-cost-bound* of the produced output U. The per-loop scan/insert budget, the source-progress (settledness) oracle, and the band-containment of child outputs are taken as the modular threaded side-conditions *step-budget*, *source-progress* and *child-containment*; each is discharged on the decoupled path below, where

the source band $a = 0$ makes them transparent.

lemma *charged-direct-insert-costed-bmssp-amortized-bound-from-budgets-progress-containment*:
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*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ charged\text{-}Us\ child\text{-}outputs\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs\ U.$
$D = label\text{-}partition\text{-}view\ (find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$partition\text{-}initial\text{-}insert\text{-}cost\text{-}bound\ c\text{-}insert\ t\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \Longrightarrow$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ (find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B)\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ B\ d'\ D\ a\ betas\ bs\ B'\ charged\text{-}Us\ child\text{-}outputs\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
$complete\text{-}on\ d'\ \{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup \{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq cap \Longrightarrow$
$tree\text{-}antichain\ S \Longrightarrow$
$fp\text{-}iter\text{-}capped\text{-}scan\text{-}cost\ k\ cap\ d\ S\ S\ B + c\text{-}insert \leq A * card\ U$
and *source-progress*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ charged\text{-}Us\ child\text{-}outputs\ U\ c\ child\text{-}costs\ child\text{-}sources.$
$charged\text{-}direct\text{-}insert\text{-}costed\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ charged\text{-}Us\ child\text{-}outputs\ U\ c\ child\text{-}costs \Longrightarrow$
$sound\text{-}label\ d \Longrightarrow$
$bmssp\text{-}pre\text{-}full\ d\ P\ B \Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$list\text{-}all2\ (\lambda S\text{-}child\ charged\text{-}child.\ \exists\ c\text{-}child\ U\text{-}child\ B\text{-}child\ d\text{-}child\ B\text{-}child'.\ charged\text{-}direct\text{-}insert\text{-}costed\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l'\ d\ S\text{-}child\ B\text{-}child\ d\text{-}child\ B\text{-}child'\ U\text{-}child\ c\text{-}child \wedge bmssp\text{-}pre\text{-}full\ d\ S\text{-}child\ B\text{-}child \wedge (\forall\, x{\in}S\text{-}child.\ reachable\ s\ x) \wedge card\ S\text{-}child \leq M\text{-}of\ l' \wedge (\forall\, x{\in}S\text{-}child.\ below\text{-}bound\ (d\ x)\ B\text{-}child))$

```
      child-sources (range-tree-child-list P a bs) ⟹
      sum-list (map card child-sources) ≤ card (range-tree-chain P a bs B')
  and child-containment:
    ⋀l' d P B d' D a betas bs B' charged-Us child-outputs U c child-costs.
      charged-direct-insert-costed-partition-loop-state Δ M-of t h k cap l' d P B
        d' D a betas bs B' charged-Us child-outputs U c child-costs ⟹
      sound-label d ⟹
      bmssp-pre-full d P B ⟹
      (⋀x. x ∈ P ⟹ reachable s x) ⟹
      list-all2 (λU-out SB. U-out ⊆ fst SB) child-outputs
        (range-tree-child-bound-pair-list P a betas bs)
  and run:
    charged-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: ⋀x. x ∈ S ⟹ reachable s x
  and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
  and S-k-cap: k * card S ≤ cap
  and S-anti: tree-antichain S
 shows c ≤ bmssp-amortized-cost-bound A R h t l (2 * l + 1) B U
 using run sound pre S-reaches below S-k-cap S-anti M-cap
proof (induction l arbitrary: d S B d' B' U c rule: less-induct)
 case (less l)
 show ?case
 proof (cases l)
  case 0
  have run0:
    charged-direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    using less.prems(1) 0 by simp
  show ?thesis
    using run0
  proof cases
    case (Charged-Direct-Insert-Base x)
    have scan:
      base-case-scan-cost Δ k x B ≤
        R * card (outgoing-edges (base-case-vertices k x B))
      using R-pos unfolding base-case-scan-cost-def by simp
    show ?thesis
      using Charged-Direct-Insert-Base 0 scan
      by (simp add: bmssp-amortized-cost-bound-def; linarith)
  qed
 next
  case (Suc l0)
  have run-suc:
    charged-direct-insert-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d' B' U c
    using less.prems(1) Suc by simp
  show ?thesis
  proof (cases rule: charged-direct-insert-costed-bmssp-SucE[OF run-suc])
    case (1 c-insert a betas bs charged-Us child-outputs U-loop c-loop
```

child-costs-loop)
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?W* = {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF less.prems*(*2*) *less.prems*(*4*)])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF less.prems*(*2*) *less.prems*(*3*) *less.prems*(*4*)] **.**
have *S-subset*: *S* ⊆ *V*
using *less.prems*(*3*) **unfolding** *bmssp-pre-full-def* **by** *blast*
have *P-subset-S*: *?P* ⊆ *S*
unfolding *find-pivots-pivots-capped-def* **by** (*auto split*: *if-splits*)
have *P-subset-V*: *?P* ⊆ *V*
using *P-subset-S S-subset* **by** *blast*
have *P-reaches*: ⋀*x*. *x* ∈ *?P* ⟹ *reachable s x*
using *P-subset-S less.prems*(*4*) **by** *blast*
have *P-k-cap*: *k* ∗ *card ?P* ≤ *cap*
by (*rule find-pivots-pivots-capped-scaled-card-le*
[*OF S-subset less.prems*(*6*)])
have *P-anti*: *tree-antichain ?P*
by (*rule find-pivots-pivots-capped-tree-antichain*[*OF less.prems*(*7*)])
have *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*charged-direct-insert-costed-bmssp* Δ *M-of t h k cap l0 ?d-fp S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full ?d-fp S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l0*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*?d-fp x*) *B-child*;
k ∗ *card S-child* ≤ *cap*;
tree-antichain S-child⟧
⟹ *c-child* ≤
bmssp-amortized-cost-bound A R h t l0 (*2* ∗ *l0* + *1*) *B-child U-child*
proof −
fix *c-child U-child S-child B-child d-child B-child′*
assume *child*:
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l0 ?d-fp S-child*
B-child d-child B-child′ U-child c-child
and *child-pre*: *bmssp-pre-full ?d-fp S-child B-child*
and *child-reaches*: ⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*
and *child-below*:
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*?d-fp x*) *B-child*
and *child-k-cap*: *k* ∗ *card S-child* ≤ *cap*
and *child-anti*: *tree-antichain S-child*
have *M-cap-child*: ⋀*i*. *i* ≤ *l0* ⟹ *M-of i* ≤ *cap*
using *less.prems*(*8*) *Suc* **by** *simp*
show *c-child* ≤

*bmssp-amortized-cost-bound A R h t l0 (2 * l0 + 1) B-child U-child*
by (*rule less.IH*[*of l0 ?d-fp S-child B-child d-child B-child′*
U-child c-child, *OF - child sound-fp child-pre child-reaches*
child-below child-k-cap child-anti M-cap-child])
(*simp add*: *Suc*)
qed
have *contain*:
list-all2 (λ*U-out SB*. *U-out* ⊆ *fst SB*) *child-outputs*
(*range-tree-child-bound-pair-list ?P a betas bs*)
by (*rule child-containment*[*OF 1*(*4*) *sound-fp pivot-pre P-reaches*])
have *child-cost-bounds*:
list-all2 (λ*c-child SB*.
case SB of (*U-child*, *B-child*) ⇒
c-child ≤ *bmssp-amortized-cost-bound A R h t l0* (*2 * l0 + 1*)
B-child U-child)
child-costs-loop (*range-tree-child-bound-pair-list ?P a betas bs*)
by (*rule charged-child-amortized-slice-bounds-from-output-and-containment*
[*OF 1*(*4*) *P-subset-V P-k-cap P-anti child-bound contain*])
have *src-prog*:
⋀*child-sources*.
list-all2
(λ*S-child charged-child*. ∃ *c-child U-child B-child d-child B-child′*.
charged-direct-insert-costed-bmssp Δ *M-of t h k cap l0 ?d-fp S-child*
B-child d-child B-child′ U-child c-child ∧
bmssp-pre-full ?d-fp S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l0* ∧
(∀ *x*∈*S-child*. *below-bound* (*?d-fp x*) *B-child*))
child-sources (*range-tree-child-list ?P a bs*) ⟹
sum-list (*map card child-sources*) ≤ *card* (*range-tree-chain ?P a bs B′*)
using *source-progress*[*OF 1*(*4*) *sound-fp pivot-pre P-reaches*] **.**
have *scan-insert*:
fp-iter-capped-scan-cost k cap d S S B + *c-insert* ≤ *A* ∗ *card U*
by (*rule step-budget*[*OF refl 1*(*3*) *1*(*4*) *1*(*5*) *1*(*1*)
less.prems(*2*) *less.prems*(*3*) *less.prems*(*4*) *less.prems*(*5*)
less.prems(*6*) *less.prems*(*7*)])
have *trace*:
concrete-partition-loop-trace ?P B a bs d′ B′ charged-Us U-loop
by (*rule charged-direct-insert-costed-partition-loop-state-trace*
[*OF 1*(*4*) *sound-fp pivot-pre P-reaches*])
have *loop-post*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF trace*])
have *finite-U-loop*: *finite U-loop*
using *loop-post* **unfolding** *bmssp-post-full-def* **by** *simp*
have *finite-U*: *finite U*
using *1*(*1*) *finite-U-loop* **by** *simp*
have *step*:
c ≤ *bmssp-amortized-cost-bound A R h t*
(*Suc l0*) (*Suc* (*Suc* (*2 * l0 + 1*))) *B U*

```
      by (rule charged-direct-insert-costed-nonbase-step-closes-amortized-bound-
from-child-costs
          [OF 1(4) sound-fp pivot-pre P-reaches child-cost-bounds source-factor
            src-prog 1(1) finite-U scan-insert 1(2)])
    show ?thesis
      using step Suc by simp
  qed
 qed
qed
```

Charged top-level refined-graph-time bound (B2), conditional on the two threaded settledness invariants. The charged analogue of *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-amortized*: every charged top-level run is bounded by the refined graph-time budget, with the per-loop scan/insert budget discharged by the bucketed step-cost lemma and the two recursion-threaded oracles — source progress and child-output band-containment — left as the only side-conditions. On the decoupled path family below both reduce to the source-band settledness already isolated for the family, so this delivers the canonical closed refined bound unconditionally there.

```
theorem finite-initial-label-charged-top-level-refined-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 source-progress:
     ⋀l' d P B d' D a betas bs B' charged-Us child-outputs U c child-costs
       child-sources.
      charged-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' charged-Us child-outputs U c child-costs ⟹
      sound-label d ⟹
      bmssp-pre-full d P B ⟹
      (⋀x. x ∈ P ⟹ reachable s x) ⟹
      list-all2
       (λS-child charged-child. ∃ c-child U-child B-child d-child B-child'.
         charged-direct-insert-costed-bmssp Δ (bmssp-level-cap k t) t h k
          (bmssp-level-cap k t l) 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 ≤ bmssp-level-cap k t l' ∧
         (∀ x∈S-child. below-bound (d x) B-child))
```

```
      child-sources (range-tree-child-list P a bs) ⟹
     sum-list (map card child-sources) ≤ card (range-tree-chain P a bs B')
  and child-containment:
   ⋀l' d P B d' D a betas bs B' charged-Us child-outputs U c child-costs.
     charged-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' charged-Us child-outputs U c child-costs ⟹
     sound-label d ⟹
     bmssp-pre-full d P B ⟹
     (⋀x. x ∈ P ⟹ reachable s x) ⟹
     list-all2 (λU-out SB. U-out ⊆ fst SB) child-outputs
      (range-tree-child-bound-pair-list P a betas bs)
  and run:
   charged-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 c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
  (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof −
 have pre: bmssp-pre-full finite-initial-label {s} Infinity
  using all-reachable finite-initial-label-source-complete
  by (rule top-bmssp-pre-full)
 have sound: sound-label finite-initial-label
  using finite-initial-label-sound[OF all-reachable] .
 have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
  using all-reachable source-in-V by blast
 have below: ⋀x. x ∈ {s} ⟹
    below-bound (finite-initial-label x) Infinity
  by simp
 have top-cap: k * card {s} ≤ bmssp-level-cap k t l
 proof −
  have one-le: 1 ≤ bmssp-level-width t l
   unfolding bmssp-level-width-def by simp
  have k * 1 ≤ k * bmssp-level-width t l
   by (rule mult-left-mono[OF one-le]) simp
  then show ?thesis
   unfolding bmssp-level-cap-def by simp
 qed
 have top-anti: tree-antichain {s}
  by simp
 have amortized:
  c ≤ bmssp-amortized-cost-bound (2 * A) R h t l (2 * l + 1) Infinity U
  proof (rule charged-direct-insert-costed-bmssp-amortized-bound-from-budgets-
progress-containment
    [where A = 2 * A and R = R,
     OF - R-pos source-factor k-pos - - source-progress child-containment
      run sound pre S-reaches below top-cap top-anti])
  show Δ ≤ 2 * A
   using degree-factor by simp
```

```
  next
    fix i
    assume i ≤ l
    then show bmssp-level-cap k t i ≤ bmssp-level-cap k t l
      by (rule bmssp-level-cap-mono)
  next
    fix l' d S B D c-insert d' a betas bs B' charged-Us child-outputs
      U-loop c-loop child-costs U
    assume D-def:
        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)
      and insert:
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B)
      and loop:
        charged-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' charged-Us child-outputs U-loop c-loop child-costs
      and complete:
        complete-on d'
          {v ∈ bound-tree S B'.
            find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
              dist s v}
      and U-def:
        U = U-loop ∪
          {v ∈ bound-tree S B'.
            find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
              dist s v}
      and sound-s: sound-label d
      and pre-s: bmssp-pre-full d S B
      and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
      and below-s: ⋀x. x ∈ S ⟹ below-bound (d x) B
      and S-k-cap: k * card S ≤ bmssp-level-cap k t l
      and anti: tree-antichain S
    have seen-success:
      B' = B ⟹
        card (find-pivots-seen-capped k (bmssp-level-cap k t l) d S B)
        ≤ card U
      by (rule charged-direct-insert-costed-step-seen-success
        [OF loop U-def sound-s pre-s reaches-s below-s])
    show fp-iter-capped-scan-cost k (bmssp-level-cap k t l) d S S B +
        c-insert ≤ 2 * A * card U
      by (rule charged-direct-insert-costed-capped-step-scan-insert-budget-from-scaled-
seen-or-threshold
        [OF degree degree-factor insert-factor insert-scaled-factor
          seen-scaled-factor insert loop U-def sound-s pre-s reaches-s
```

```
      S-k-cap anti k-pos seen-success])
  qed
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule charged-direct-insert-costed-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  have U-V: U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  have graph-bound:
    bmssp-amortized-cost-bound (2 * A) R h t l (2 * l + 1) Infinity U
      ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
        (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
  proof −
    have U-card: card U = vertex-count
      unfolding U-V vertex-count-def by simp
    have out-le: card (outgoing-edges U) ≤ edge-count
      by (rule edge-count-outgoing-bound)
    have range-le: card (outgoing-edges-range U 0 Infinity) ≤ edge-count
      by (rule card-outgoing-edges-range-le-edge-count)
    have out-term:
      (R + l * h) * card (outgoing-edges U) ≤ (R + l * h) * edge-count
      using out-le by simp
    have range-term:
      t * card (outgoing-edges-range U 0 Infinity) ≤ t * edge-count
      using range-le by simp
    show ?thesis
      unfolding bmssp-amortized-cost-bound-def bmssp-refined-graph-time-bound-def
      using U-card out-term range-term by (simp add: algebra-simps; linarith)
  qed
  show ?thesis
    using amortized graph-bound by linarith
qed

end

end
```

References

[1] 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*, pages 36–44, 2025. DOI: `https://doi.org/10.1145/3717823.3718179`.

[2] A. F. Ramos, D. B. Hulak, and R. J. G. B. de Queiroz. Correctness and complexity of BMSSP in Isabelle/HOL. *Archive of Formal Proofs*,

June 2026. https://isa-afp.org/entries/BMSSP_Correctness.html, Formal proof development.