Formalization of Lexicographic Breadth-First Search

Thomas Holme Surlykke and Dmitriy Traytel

November 26, 2026

Abstract

Despite widespread use of the lexicographic breadth-first search graph ordering algorithm, the literature lacks a complete proof characterizing its output. We verify that the standard characterization provides a sound and complete specification. Our approach introduces a binary reformulation of the label function, exposing intermediate states and enabling proof through monadic refinement.

Contents

1 Introduction **2**

2 Preliminaries **3**
2.1 Triples . . . 3
2.2 Lists . . . 3

3 Implementation and Characterization **5**

4 Soundness **6**
4.1 The binary algorithm and its soundness . . . 6
4.1.1 Preservation of invariants . . . 8
4.2 Soundness of *lexbfs-bin* . . . 10
4.3 *lexbfs* refines *lexbfs-bin* . . . 11
4.4 Soundness of *lexbfs* . . . 13

5 Completeness **14**
5.1 Completeness of *lexbfs-bin* . . . 14
5.1.1 Reconstructing the state of *lexbfs-bin* . . . 14
5.1.2 Preservation of the constructed state . . . 16
5.2 *lexbfs-bin* refines *lexbfs* . . . 18
5.3 Completeness of *lexbfs* . . . 19

6 Correctness **19**

1 Introduction

Lexicographic breadth-first search (LexBFS) is a graph vertex ordering algorithm implementable in linear time [5]. Rose, Tarjan, and Lueker [6] developed it to decompose graphs into slices according to vertex distance from an initial vertex. They use it to find *minimal orderings* and *perfect orderings* of undirected graphs. The algorithm has since found applications in *Chordal Graphs* and *Graph Coloring* [5].

We call any complete list of vertices in a graph a *LexBFS ordering* if it is a possible output of the LexBFS algorithm. As the algorithm selects an arbitrary maximal vertex when multiple exist, different implementations may produce different orderings. Therefore, it is a non-trivial task to characterize the structure and behavior of LexBFS orderings. The Wikipedia page *Lexicographic breadth-first search* [5] provides a mathematical characterization of all LexBFS orderings:

> **Theorem** (LexBFS Property [5])**.** *Let $G = (V, E)$ be a graph with n vertices. Recall that $N(v)$ is the set of neighbors of v. Let $\sigma = (v_1, \ldots, v_n)$ be an enumeration of the vertices of V. The enumeration σ is a LexBFS ordering (with source v_1) if, for all $1 \leq i < j < k \leq n$ with $v_i \in N(v_k) - N(v_j)$, there exists $m < i$ such that $v_m \in N(v_j) - N(v_k)$.*

Wikipedia does not reference a proof, but references Corneil [2], which also describes the LexBFS algorithm and states a variant of the property:

> **Theorem** (LexBFS Property [2])**.** *An ordering $\prec$ of the vertices of an arbitrary graph $G(V, E)$ is an LexBFS ordering if and only if for all vertices a, b, c of G such that $ac \in E$ and $bc \notin E$, $c \prec b \prec a$ implies the existence of a vertex d in G, adjacent to b but not to a and such that $d \prec c$.*

Corneil [2] also provides no proof, but instead references [1], which defines the property as:

(P1) If $a < b < c$ and $ac \in E$ and $ab \notin E$ then there exists a vertex d such that $c < d$, $db \in E$ and $da \notin E$.

and states:

> **Theorem** (LexBFS Property [1])**.**
>
> *(1) Any LexBFS-ordering has property (P1).*
>
> *(2) Any ordering fulfilling (P1) can be generated by LexBFS.*

This paper does prove (2), but for (1) they merely state:

> *Proof.* (1) We refer to the well known proof in [21]. □

Their reference [21] is Golumbic [4], who lists yet another variant of the property as property (L3):

> **Theorem** (LexBFS Property (L3) [4])**.** *if* $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ *and* $c \in \mathrm{Adj}(a) - \mathrm{Adj}(b)$*, then there exists a vertex* $d \in \mathrm{Adj}(b) - \mathrm{Adj}(a)$ *with* $\sigma^{-1}(c) < \sigma^{-1}(d)$*.*

Again, despite what was written in [1], no proof is listed. The "well-known" proof is merely an exercise [4]:

> Exercises
> ...
> 3. Prove properties (L1)–(L3) of lexicographic breadth-first search (Section 4.3).

This formalization is motivated by the absence of a published proof, combined with the inconsistent notation across sources. We verify that the LexBFS Property is both sound and complete as a specification of the LexBFS algorithm: Any LexBFS ordering satisfies the property (soundness), and any ordering satisfying the property can be produced by LexBFS (completeness).

2 Preliminaries

2.1 Triples

Three indices meaning c picked before b, picked before a.

abbreviation *triple* σ *ci bi ai* $\equiv$ *ci* $<$ *bi* $\wedge$ *bi* $<$ *ai* $\wedge$ *ai* $<$ *length* σ

2.2 Lists

lemma *nth-not-mem-take-if-ge*: $[\![i \leq j; j < \mathit{length}\ \mathit{xs}; \mathit{distinct}\ \mathit{xs}]\!] \Longrightarrow \mathit{xs}\ !\ j \notin \mathit{set}\ (\mathit{take}\ i\ \mathit{xs})$
by (*metis distinct-take id-take-nth-drop not-distinct-conv-prefix set-take-subset-set-take subset-code*(*1*))

We compare lists lexicographically.

lemma *list-lt-iff-ind-lt*:
fixes *xs*::('*c*::*linorder list*)
assumes *len-eq*: *length xs* = *length ys*

```
  shows xs < ys ⟷ (∃ i < length xs. xs ! i < ys ! i ∧ (∀j<i . xs ! j = ys ! j))
  using len-eq
proof (induct xs ys rule: list-induct2)
  case Nil thus ?case by simp
next
  — Shorthand for conclusion
  let ?exists-first-less = λ xs ys. ∃i<length xs. xs ! i < ys ! i ∧ (∀j<i. xs ! j = ys
! j)
  case (Cons x xs y ys)
  from Cons show ?case
  proof (intro iffI)
  next
    assume ?exists-first-less (x#xs) (y#ys)
    then obtain i where i-hyp:
      i<length (x#xs)
      (x#xs) ! i < (y#ys)!i
      ∀j<i. (x # xs) ! j = (y # ys) ! j
      by blast
    then show x # xs < y # ys
    proof (cases i)
      case (Suc k)
      with i-hyp len-eq have k < length xs
        and xs ! k < ys ! k
        and ∀ j < k . xs ! j = ys ! j
        by auto
      with Cons.hyps have xs < ys
        by blast

      moreover from i-hyp Suc len-eq have x = y
        by auto

      ultimately show ?thesis
        by simp
    qed (simp add: i-hyp)
  next
    assume x # xs < y # ys
    with Cons.prems Cons-less-Cons consider x < y | (non-triv) x = y xs < ys
      by blast
    then show ?exists-first-less (x#xs) (y#ys)
    proof cases
      case non-triv
      have ∗∗: ∀ j < length xs . (x # xs) ! Suc j = xs ! j
        by simp
      with Cons.hyps non-triv obtain i
        where i < length xs xs!i < ys!i
          and ∗: ∀ j<i . xs ! j = ys ! j
        by blast
      hence (x # xs) ! 0 = (y # ys) ! 0
        using ‹x =y›
```

```
      by simp
    moreover have ∀ z zs . ∀ i < length zs . zs ! i = (z # zs) ! Suc i
      by simp
    hence ∀ j < i . (x # xs) ! Suc j = (y # ys) ! Suc j
      using ∗ ‹i < length xs› ‹length xs = length ys›
      by auto
    ultimately have ∀ j < Suc i . (x#xs) ! j = (y # ys) ! j
      by (metis Suc-less-eq old.nat.exhaust)

    with ‹i < length xs› ‹xs!i < ys ! i› show ?thesis
      by (intro exI[where x=Suc i]) simp
  qed (auto)
 qed
qed

lemma prev-lt-if-ind-gt:
 fixes xs :: 'c::linorder list
 assumes length xs = length ys
   and i < length xs
   and xs ≤ ys
   and xs!i > ys!i
 shows ∃ j < i . xs!j < ys!j
 using assms(1,3,4) by (metis list-lt-iff-ind-lt not-less-iff-gr-or-eq order-le-imp-less-or-eq)

context Graph
begin

abbreviation is-arc :: 'a ⇒ 'a ⇒ bool (infixl ∼ 60) where v∼w ≡ (v,w) ∈ E
abbreviation is-not-arc :: 'a ⇒ 'a ⇒ bool (infixl ∼∗ 60) where v ∼∗ w ≡ ¬
v∼w
```

3 Implementation and Characterization

The LexBFS algorithm, adapted as closely as possible from Algorithm 1 of Corneil et al. [3]:

```
definition lexbfs where lexbfs = do {
 (i, U, ℒ, σ) ← WHILE_T (λ(i, U, ℒ, σ). U ≠ {}) (λ(i, U, ℒ, σ). do {
   x <− RES {x ∈ U. ℒ x = Max (ℒ ‘ U)};
   let U = U − {x};
   let ℒ = (λy. if y ∈ U ∧ x ∼ y then ℒ y @ [i] else ℒ y);
   RETURN (i − 1, U, ℒ, σ @ [x])
}) (card V, V, λ - . [], []);
 RETURN σ
}
```

The LexBFS property.

```
definition lexbfs-prop σ ≡
 ∀ ic ib ia. triple σ ic ib ia ∧ σ!ic ∼ σ!ia ∧ σ!ic ∼∗ σ!ib
```

$\longrightarrow (\exists\, id{<}ic.\ \ \sigma!id \sim\!\!* \sigma!ia\ \ \wedge\ \ \sigma!id \sim \sigma!ib)$

For *lexbfs-prop* to be an actual characterization, we need the additional assumption that the elements of σ correspond precisely to the vertices of V.

definition *is-lexbfs* $\sigma \longleftrightarrow$ *distinct* $\sigma \wedge$ *set* $\sigma = V \wedge$ *lexbfs-prop* σ

4 Soundness

In this section we prove soundness of the LexBFS algorithm with respect to the specification *is-lexbfs*. In other words, we show *lexbfs* $\leq$ *SPEC is-lexbfs*.

4.1 The binary algorithm and its soundness

The algorithm with $\mathcal{L}$ replaced by the binary label function $\mathcal{B}$. We will, however, define the algorithm's invariants before defining the algorithm.

Convention: We use $\mathcal{B}$ for the binary label function, and $\mathcal{L}$ for the ordinary label function.

definition *partition-V U* $\sigma \equiv$ *distinct* $\sigma \wedge U \cup$ *set* $\sigma = V \wedge U \cap$ *set* $\sigma = \{\}$
lemma *partition-V-empty*[*simp*]: *partition-V V* []
by (*simp add*: *partition-V-def*)

lemma *U-finite-if-partition-V*: *partition-V U* $\sigma \Longrightarrow$ *finite U*
by (*metis partition-V-def Un-upper1 finite-subset finite-vertex-set*)

lemma *len-sigma-if-partition-V*: *partition-V U* $\sigma \Longrightarrow$ *length* σ = *card* (*set* σ)
by (*metis partition-V-def distinct-card*)

lemma *card-V-split-if-partition-V*: *partition-V U* $\sigma \Longrightarrow$ *card V* = *card U* + *length* σ
using *card-Un-disjoint*[**where** *A*=*U* **and** *B*=*set* σ]
by (*auto simp add*: *partition-V-def U-finite-if-partition-V len-sigma-if-partition-V*)

This states that, if we look at any element $y \in \sigma$ and x later than y (either to the right of y in σ, or not in σ at all), then x must have a less-or-equal $\mathcal{B}$-label value, if we truncate $\mathcal{B}$ to the point where y was added.

definition *label-le-earlier* $\mathcal{B}$ $\sigma \equiv$
$\forall\, iy{<}$*length* σ. $\forall\, x{\in}V$ − *set* (*take iy* σ). *take iy* ($\mathcal{B}$ (σ ! *iy*)) $\geq$ *take iy* ($\mathcal{B}$ *x*)

lemma *label-le-earlierD*[*simp*]:
⟦*label-le-earlier* $\mathcal{B}$ σ; *iy* < *length* σ; *x* ∈ *V* − *set* (*take iy* σ)⟧
$\Longrightarrow$ *take iy* ($\mathcal{B}$ (σ!*iy*)) $\geq$ *take iy* ($\mathcal{B}$ *x*)
by (*simp add*: *label-le-earlier-def*)

States that each entry of $\mathcal{B}$ x determines whether a particular vertex in σ is connected to x or not.

definition *label-iff-connected* $\mathcal{B}\ \sigma \equiv \forall\ ix{<}length\ \sigma.\ \forall\, y.\ (\mathcal{B}\ y)!ix \longleftrightarrow (\sigma!ix){\sim}y$
lemma *label-iff-connectedD*[*simp*]:
$[\![label\text{-}iff\text{-}connected\ \mathcal{B}\ \sigma;\ iy < length\ \sigma]\!] \Longrightarrow \mathcal{B}\ x!iy \longleftrightarrow (\sigma!iy){\sim}x$
by (*simp add*: *label-iff-connected-def*)

definition *label-len-eq* $\mathcal{B}\ \sigma \equiv \forall\ x\ .\ length\ (\mathcal{B}\ x) = length\ \sigma$

This simp rule chooses *length* σ as simplest representation, and helps a lot in automating the proofs.

lemma *len-eq-sigma-if-label-len-eq*[*simp*]: *label-len-eq* $\mathcal{B}\ \sigma \Longrightarrow length\ (\mathcal{B}\ x) = length\ \sigma$
by (*simp add*: *label-len-eq-def*)

lemma *len-take-eq-if-label-len-eq*:
label-len-eq $\mathcal{B}\ \sigma \Longrightarrow length\ (take\ i\ (\mathcal{B}\ x)) = length\ (take\ i\ (\mathcal{B}\ y))$
by *simp*

Combining the invariants.

definition *lexbfs-bin-INV* $\equiv \lambda(U,\ \mathcal{B},\ \sigma)\ .$
partition-V $U\ \sigma$
$\wedge$ *label-le-earlier* $\mathcal{B}\ \sigma$
$\wedge$ *label-iff-connected* $\mathcal{B}\ \sigma$
$\wedge$ *label-len-eq* $\mathcal{B}\ \sigma$

The refine WHILE procedure requires writing *lexbfs-bin-INV* as a function on a tuple. This unfortunately turns unfolding *lexbfs-bin-INV* into a case statement with a single case. This lemma automatically unfolds *lexbfs-bin-INV* into the auxiliary invariants.

lemma *INV-iff*[*iff*]:
lexbfs-bin-INV $(U,\ \mathcal{B},\ \sigma)$
$\longleftrightarrow$ *partition-V* $U\ \sigma \wedge$ *label-le-earlier* $\mathcal{B}\ \sigma \wedge$ *label-iff-connected* $\mathcal{B}\ \sigma \wedge$ *label-len-eq* $\mathcal{B}\ \sigma$
by (*simp add*: *lexbfs-bin-INV-def*)

Finally, the binary version of the LexBFS algorithm.

definition *lexbfs-bin* **where** *lexbfs-bin* = *do* {
    $(U,\ \mathcal{B},\ \sigma) \leftarrow WHILE_T{}^{lexbfs\text{-}bin\text{-}INV}\ (\lambda(U,\ \mathcal{B},\ \sigma).\ U \neq \{\})\ (\lambda(U,\ \mathcal{B},\ \sigma).$ *do* {
    x <− *RES* $\{x \in U.\ \mathcal{B}\ x = Max\ (\mathcal{B}\ `\ U)\}$;
    let $U = U - \{x\}$;
    let $\mathcal{B} = (\lambda y.\ \mathcal{B}\ y\ @\ [x{\sim}y])$;
    RETURN $(U,\ \mathcal{B},\ \sigma\ @\ [x])$
}) $(V,\ \lambda\ \text{-}\ .\ [],\ [])$;
    RETURN σ
}

4.1.1 Preservation of invariants

We start with the main point of the invariants – if the invariants hold for the terminal case (where $U = \{\}$), then the output satisfies *is-lexbfs*. The remaining work will then be showing that the invariants are preserved.

lemma *lexbfs-prop-if-INV*:
  assumes *eq-V*: *partition-V* {} σ
    and *aux1*: *label-le-earlier* $\mathcal{B}$ σ
    and *aux2*: *label-iff-connected* $\mathcal{B}$ σ
    and *aux3*: *label-len-eq* $\mathcal{B}$ σ
  shows *lexbfs-prop* σ
  unfolding *lexbfs-prop-def*
proof (*intro allI impI*)
  fix *ic ib ia*
  let *?a* = σ!*ia* **and** *?b* = σ!*ib* **and** *?c* = σ!*ic*
    and *?*$\mathcal{B}$*-b* = λx . *take ib* ($\mathcal{B}$ *x*) — $\mathcal{B}$ function truncated at moment before *?b* added
  assume *triple* σ *ic ib ia* $\wedge$ *?c* $\sim$ *?a* $\wedge$ *?c* $\sim*$ *?b*
  hence *triple*: *triple* σ *ic ib ia* **and** *?c* $\sim$ *?a ?c* $\sim*$ *?b*
    by *simp-all*
  from *triple eq-V* **have** *?a* $\notin$ *set* (*take ib* σ)
    using *partition-V-def*
    by (*simp add*: *nth-not-mem-take-if-ge*)
  with *triple eq-V* **have** *?a* $\in$ *V* $-$ *set* (*take ib* σ)
    using *partition-V-def*
    by *auto*
  with *aux1 triple* **have** *?*$\mathcal{B}$*-b ?b* $\geq$ *?*$\mathcal{B}$*-b ?a*
    by *simp*

  moreover from *aux2 triple* ‹*?c*$\sim$*?a*› ‹*?c*$\sim*$*?b*› **have** $\mathcal{B}$ *?b* !*ic* $<$ $\mathcal{B}$ *?a* !*ic*
    by *simp*
  with *triple* **have** *?*$\mathcal{B}$*-b ?b* !*ic* $<$ *?*$\mathcal{B}$*-b ?a* !*ic*
    by *simp*

  moreover from *aux3 triple* **have** *ic* $<$ *length* (*?*$\mathcal{B}$*-b ?b*)
    by *simp*

  ultimately have $\exists$ *id* $<$ *ic* . *?*$\mathcal{B}$*-b ?b* !*id* $>$ *?*$\mathcal{B}$*-b ?a* !*id*
    by (*metis aux3 len-take-eq-if-label-len-eq prev-lt-if-ind-gt*)
  then obtain *id* **where** *id* $<$ *ic ?*$\mathcal{B}$*-b ?a* !*id* $<$ *?*$\mathcal{B}$*-b ?b* !*id*
    by *metis*
  from ‹*id* $<$ *ic*› *triple* **have** ‹*triple* σ *id ib ia*›
    by *simp*
  let *?d* = σ!*id*
  from ‹*?*$\mathcal{B}$*-b ?a* !*id* $<$ *?*$\mathcal{B}$*-b ?b* !*id*› ‹*triple* σ *id ib ia*› **have** $\mathcal{B}$ *?a* !*id* $<$ $\mathcal{B}$ *?b* !*id*
    by *simp*
  with ‹*triple* σ *id ib ia*› *aux2* **have** *?d*$\sim$*?b* $\wedge$ *?d*$\sim*$*?a*
    by *simp*
  with ‹*id* $<$ *ic*› **show** $\exists$ *id*$<$*ic*. σ!*id* $\sim*$ σ!*ia* $\wedge$ σ!*id* $\sim$ σ!*ib*

by *metis*
qed

lemma *is-lexbfs-if-INV*:
lexbfs-bin-INV ({}, $\mathcal{B}$, σ) $\Longrightarrow$ *is-lexbfs* σ
unfolding *is-lexbfs-def* **using** *lexbfs-prop-if-INV partition-V-def*
by *blast*

The following lemmas show that if an auxiliary invariant holds, then it also holds during the next step of the while loop.

lemma *partition-V-next*:
⟦*partition-V U* σ; $x \in U$⟧ $\Longrightarrow$ *partition-V* ($U - \{x\}$) (σ@[x])
by (*auto simp add*: *partition-V-def*)

lemma *label-le-earlier-next*:
assumes *partition*: *partition-V U* σ
and *aux1*: *label-le-earlier* $\mathcal{B}$ σ
and *aux3*: *label-len-eq* $\mathcal{B}$ σ
and $\mathcal{B}$ a = *Max* ($\mathcal{B}$ ‘ *U*)
shows *label-le-earlier* (λy. $\mathcal{B}$ y @[$a \sim y$]) (σ@[a])
unfolding *label-le-earlier-def*
proof *safe*
fix *iy x*
let *?y* = (σ @ [a]) !*iy* — Vertex obtained from index *iy*
let *?*$\mathcal{B}$*-next* = λz . $\mathcal{B}$ z @[$a \sim z$] — $\mathcal{B}$ updated by appending whether vertex is connected to a
assume *iy* < *length* (σ @ [a])
and $x \notin$ *set* (*take iy* (σ @ [a]))
and $x \in V$
show *take iy* (*?*$\mathcal{B}$*-next x*) $\leq$ *take iy* (*?*$\mathcal{B}$*-next ?y*)
proof *cases*
assume *length* σ = *iy*
with *partition* ‹$x \in V$› ‹$x \notin$ *set* (*take iy* (σ@[a]))› **have** $x \in U$
by (*auto simp add*: *partition-V-def*)
with ‹$\mathcal{B}$ a = *Max* ($\mathcal{B}$ ‘ *U*)› **have** $\mathcal{B}$ x $\leq$ $\mathcal{B}$ a
using *partition*
by (*simp add*: *U-finite-if-partition-V*)

moreover from ‹*length* σ = *iy*› *aux3* **have** $\forall$ z . *take iy* (*?*$\mathcal{B}$*-next z*) = $\mathcal{B}$ z
by *simp*

moreover from ‹*length* σ = *iy*› **have** *?y* = a
by *auto*

ultimately show *?thesis*
by *simp*
next
assume *length* $\sigma \neq$ *iy*

with ‹*iy* < *length* (*σ* @ [*a*])› **have** *iy* < *length σ*
by *simp*
with *aux1* ‹*x* ∈ *V*› ‹*x* ∉ *set* (*take iy* (*σ* @ [*a*]))› **have** *take iy* (*B ?y*) ≥ *take iy* (*B x*)
by *simp*
with ‹*iy* < *length σ*› *aux3* **show** *?thesis*
by *simp*
qed
qed

lemma *label-iff-connected-next*:
assumes *aux2*: *label-iff-connected B σ* **and** *aux3*: *label-len-eq B σ*
shows *label-iff-connected* (*λy. B y* @[*a*∼*y*]) (*σ*@[*a*])
unfolding *label-iff-connected-def*
proof (*intro allI impI*)
fix *iy x* **assume** *iy* < *length* (*σ*@[*a*])
then consider (*lt*) *iy* < *length σ* | (*eq*) *iy* = *length σ*
by *fastforce*
thus (*B x* @[*a*∼*x*])!*iy* = (*σ*@[*a*])!*iy* ∼*x*
proof *cases*
case *lt*
with *aux2 aux3* **show** *?thesis*
by *simp*
next
case *eq*
with *aux3* **have** *length* (*B x*) = *iy*
by *simp*
with *eq* **show** *?thesis*
by (*metis nth-append-length*)
qed
qed

lemma *label-len-eq-next*:
label-len-eq B σ ⟹ *label-len-eq* (*λy. B y* @ [*x*∼*y*]) (*σ*@[*x*])
unfolding *label-len-eq-def*
by *simp*

4.2 Soundness of *lexbfs-bin*

proposition *lexbfs-bin-sound*: *lexbfs-bin* ≤ *SPEC is-lexbfs*
unfolding *lexbfs-bin-def*
apply (*refine-vcg WHILEIT-rule*[**where** *I*=*lexbfs-bin-INV* **and** *R*=*measure* (*λ*(*U*, *B*, *σ*) . *card U*)])
subgoal — Wellfoundedness of measure
by *simp*
subgoal — Initialization: *lexbfs-bin-INV* holds for initial state
by (*simp add*: *partition-V-def label-le-earlier-def*

label-iff-connected-def label-len-eq-def)
subgoal — Preservation: *lexbfs-bin-INV* preserved at each WHILE-step
by (*auto simp add*: *partition-V-next label-le-earlier-next*
label-iff-connected-next label-len-eq-next)
subgoal — Termination: measure strictly decreases
by *clarify* (*simp add*: *U-finite-if-partition-V card-gt-0-iff*)
subgoal — Soundness: *lexbfs-bin-INV* on terminal state implies *is-lexbfs*
by (*auto simp add*: *is-lexbfs-if-INV*)
done

4.3 *lexbfs* refines *lexbfs-bin*

For ease, we use an intermediate algorithm, where each label is also updated outside of U.

definition *lexbfs-helper* **where** *lexbfs-helper* = *do* {
(i, U, $\mathcal{L}$, σ) ← $WHILE_T$ ($\lambda(i,\ U,\ \mathcal{L},\ \sigma).\ U \neq \{\}$) ($\lambda(i,\ U,\ \mathcal{L},\ \sigma).$ *do* {
x <− *RES* $\{x \in U.\ \mathcal{L}\ x = Max\ (\mathcal{L}\ `\ U)\}$;
let $U = U - \{x\}$;
— Also updates vertices already checked
let $\mathcal{L} = (\lambda y.$ *if* $x \sim y$ *then* $\mathcal{L}\ y$ @ $[i]$ *else* $\mathcal{L}\ y)$;
RETURN ($i - 1$, U, $\mathcal{L}$, σ @ $[x]$)
}) (*card* V, V, λ - . [], []);
RETURN σ
}

Transforms the list of Booleans by placing V-index on *True*, and removing *False*.

fun *transform-bin* :: *nat* ⇒ *bool list* ⇒ *nat list* **where**
transform-bin - [] = [] |
transform-bin n (*False*#*xs*) = *transform-bin* (*n*−*1*) *xs*|
transform-bin n (*True*#*xs*) = *n* # *transform-bin* (*n*−*1*) *xs*

lemma *transform-bin-split*[*simp*]: *transform-bin n* (*xs* @ *ys*) = *transform-bin n xs* @ *transform-bin* (*n* − *length xs*) *ys*
by (*induct n xs rule*: *transform-bin.induct*) *simp-all*

lemma *hd-transform-bin-le*[*dest*]: *transform-bin n xs* = *a* # *as* ⟹ *a* ≤ *n*
by (*induction n xs rule*: *transform-bin.induct*) *simp-all*

A rephrasing of *transform-bin* (*Suc n*) (*x* # *xs*) and *transform-bin* (*Suc n*) (*y* # *ys*), when $x < y$, i.e. $x =$ *False* and $y =$ *True*.

lemma *transform-bin-decreasing*[*simp*]: *transform-bin n xs* < *Suc n* # *transform-bin n ys*
by (*cases transform-bin n xs*) *auto*

proposition *transform-bin-le-iff*:

⟦*length xs* = *length ys*; *length ys* ≤ *n*⟧ ⟹ *xs* ≤ *ys* ⟷ *transform-bin n xs* ≤ *transform-bin n ys*
proof (*induct xs ys arbitrary*: *n rule*: *list-induct2*)
case (*Cons x xs y ys*)
thus *?case* **proof** (*cases n*)
case (*Suc k*) — *n* = *Suc k*
with *Cons* **have** *hyp*: *transform-bin k xs* ≤ *transform-bin k ys* ⟷ *xs* ≤ *ys*
by *simp*
from *Cons-le-Cons* **have**
x#*xs* ≤ *y*#*ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* ≤ *ys*
¬ *x*#*xs* ≤ *y*#*ys* ⟷ *x* > *y* ∨ *x* = *y* ∧ *xs* > *ys*
by *auto*
moreover {
assume *x*<*y*
hence *transform-bin n* (*x*#*xs*) < *transform-bin n* (*y*#*ys*)
by (*simp add*: *Suc*)
hence *transform-bin n* (*x*#*xs*) ≤ *transform-bin n* (*y*#*ys*)
by *simp*
}
moreover {
assume *x*=*y xs*≤*ys*
with *hyp* **have** *transform-bin n* (*x*#*xs*) ≤ *transform-bin n* (*y*#*ys*)
by (*auto simp add*: *Suc*)
}
moreover {
assume *x*=*y xs* > *ys*
with *hyp* **have** *transform-bin n* (*x*#*xs*) > *transform-bin n* (*y*#*ys*)
by (*auto simp add*: *Suc*)
}
moreover {
assume *x*>*y*
hence *transform-bin n* (*x*#*xs*) > *transform-bin n* (*y*#*ys*)
by (*simp add*: *Suc*)
}
ultimately show *?thesis* **using** *not-le* **by** *blast*
qed *simp*
qed *simp*

Maps the binary label function $\mathcal{B}$ to the ordinary label function $\mathcal{L}$, except this $\mathcal{L}$ also updates value of vertices already processed.

abbreviation *transform-B* $\mathcal{B}$ *x* ≡ *transform-bin* (*card V*) ($\mathcal{B}$ *x*)

lemma *transform-B-refl-Max*: ⟦*partition-V U* σ; *label-len-eq* $\mathcal{B}$ σ; *x* ∈ *U*⟧
⟹ $\mathcal{B}$ *x* = *Max* ($\mathcal{B}$ ‘ *U*) ⟷ *transform-B* $\mathcal{B}$ *x* = *Max* (*transform-B* $\mathcal{B}$ ‘ *U*)
using *transform-bin-le-iff*[**where** *xs* = $\mathcal{B}$ - **and** *ys* =$\mathcal{B}$ - **and** *n*=*card V*]
by (*simp add*: *card-V-split-if-partition-V U-finite-if-partition-V len-sigma-if-partition-V*

eq-Max-iff memb-imp-not-empty)

Note: The *Id* allows us to specify what relation these algorithms are equal through. So the outer monad (that returns σ after while loop exits) is completely equal, while the WHILE loops are refined through the relation below.

abbreviation *R-helper-bin* $\equiv \{(b,\ (U,\ \mathcal{B},\ \sigma)).\ b = (card\ U,\ U,\ transform\text{-}B\ \mathcal{B},\ \sigma)\}$

lemma *lexbfs-helper-refines-lexbfs-bin*: *lexbfs-helper* $\leq \Downarrow$ *Id lexbfs-bin*
  unfolding *lexbfs-helper-def lexbfs-bin-def*
  apply (*refine-rcg WHILET-refine'*[**where** R = *R-helper-bin*])
  subgoal — Initialization: Initial states related
    by *simp*
  subgoal — Termination: Related states exit WHILE loop at same time
    by *simp*
  subgoal — Step: Related states refine
    by (*auto simp add*: *transform-B-refl-Max*)
  subgoal — Preservation: WHILE loop preserves relation
    by (*auto simp add*: *card-V-split-if-partition-V*)
  subgoal — Return: Related states return same list at end
    by *simp*
  done

The relation that identifies *lexbfs-helper* and *lexbfs*. Note that this relation is symmetric, so we can reuse it later when we show *lexbfs-helper* $\leq$ *lexbfs*.

abbreviation *R-lexbfs-helper* $\equiv \{((i,\ U,\ \mathcal{L},\ \sigma),\ (i',\ U',\ \mathcal{L}',\ \sigma'))\ .$
  $i = i' \wedge U = U' \wedge \sigma = \sigma' \wedge (\forall\ x \in U'\ .\ \mathcal{L}\ x = \mathcal{L}'\ x)\}$

lemma *lexbfs-refines-lexbfs-helper*: *lexbfs* $\leq \Downarrow$ *Id lexbfs-helper*
  unfolding *lexbfs-def lexbfs-helper-def*
  by (*refine-rcg WHILET-refine*[**where** R = *R-lexbfs-helper*]) *auto*

4.4 Soundness of *lexbfs*

Combining the previous results, and using transitivity of refinement, we can now show that LexBFS is sound with regard to the specification *is-lexbfs*. In other words: Any list that can be produced by the *lexbfs* algorithm satisfies the *lexbfs-prop*.

theorem *lexbfs-sound*: *lexbfs* $\leq$ *SPEC is-lexbfs*
  using
    lexbfs-bin-sound
    lexbfs-helper-refines-lexbfs-bin
    lexbfs-refines-lexbfs-helper
  by *simp*

5 Completeness

5.1 Completeness of *lexbfs-bin*

5.1.1 Reconstructing the state of *lexbfs-bin*

By definition of the binary label-function $\mathcal{B}$, and as indicated by the invariant *label-iff-connected*, it is possible and easy to obtain $\mathcal{B}$ from the currently picked vertices σ.

definition *construct-B* σ $y \equiv$ *map* $(\lambda x.\ x{\sim}y)$ σ

lemma *construct-B-empty*[*simp*]: *construct-B* [] $= (\lambda$ - . [])
unfolding *construct-B-def*
by *simp-all*

lemma *construct-B-tail*[*simp*]: *construct-B* $(\sigma @ [x])$ $y =$ *construct-B* σ y @ $[x{\sim}y]$
by (*simp add*: *construct-B-def*)

The constructed $\mathcal{B}$ automatically satisfies two of our auxiliary invariants: *label-len-eq* and *label-iff-connected*.

lemma *construct-B-iff-connected*:
$i <$ *length* $\sigma \Longrightarrow$ *construct-B* σ a ! $i <$ *construct-B* σ b ! $i \longleftrightarrow \sigma!i \sim* a \wedge \sigma!i \sim b$
by (*simp add*: *construct-B-def*)

lemma *construct-B-label-iff-connected*: *label-iff-connected* (*construct-B* σ) σ
by (*simp add*: *construct-B-def label-iff-connected-def*)

By making this a simp rule, we also easily obtain that *length* (*construct-B* σ x) = *length* (*construct-B* σ y), for any x, y.

lemma *construct-B-len-eq-sigma*[*simp*]: *length* (*construct-B* σ x) = *length* σ
by (*simp add*: *construct-B-def*)

lemma *construct-B-len-eq*[*simp*]: *label-len-eq* (*construct-B* σ) σ
by (*simp add*: *label-len-eq-def*)

Reconstruction of U given the currently picked vertices σ. We hide this behind a definition, as being able to control what is expanded through the simp rules below is very useful in the *refine-rcg* proofs, as those have a lot of assumptions and statements in them.

definition *U-at* $\sigma \equiv V -$ *set* σ

lemma *U-at-finite*[*simp*]: *finite* (*U-at* σ)
by (*simp add*: *finite-vertex-set U-at-def*)

lemma *U-at-eq-rest*: *is-lexbfs* $(\sigma$ @ σ-*rest*$) \Longrightarrow$ *U-at* $\sigma =$ *set* σ-*rest*
by (*auto simp add*: *U-at-def is-lexbfs-def*)

lemma *U-at-not-empty*: *is-lexbfs* (σ @ a # σ-*rest*) $\Longrightarrow$ *U-at* $\sigma \neq \{\}$
by (*auto simp add*: *U-at-def is-lexbfs-def*)

lemma *U-at-empty*[*simp*]: *is-lexbfs* $\sigma \Longrightarrow$ *U-at* $\sigma = \{\}$
by (*simp add*: *is-lexbfs-def U-at-def*)

lemma *U-at-empty-eq-V*[*simp*]: *U-at* [] = *V*
by (*simp add*: *U-at-def*)

lemma *U-at-next*: $x \in$ *U-at* $\sigma \Longrightarrow$ *U-at* (σ @ [x]) = *U-at* $\sigma - \{x\}$
by (*auto simp add*: *U-at-def*)

We also add an abbreviation for the reconstructed state.

abbreviation *bin-state-at* $\sigma \equiv$ (*U-at* σ, *construct-B* σ, σ)

By *candidate* we mean that it can be picked in a given step of the algorithm.

abbreviation *candidates* $\sigma \equiv \{x \in$ *U-at* σ . *construct-B* σ x = *Max* (*construct-B* σ ‘ *U-at* σ)$\}$

This shows that it is always possible to pick a next element to add to σ, i.e. that the algorithm can't fail.

lemma *obtains-candidate*: **assumes** *is-lexbfs* (σ @ a # σ-*rest*)
obtains x **where** $x \in$ *candidates* σ
proof *atomize-elim*
from *assms* **have** *finite* (*U-at* σ) *U-at* $\sigma \neq \{\}$
by(*simp-all add*: *U-at-eq-rest*)
thus $\exists x.\ x \in$ *candidates* σ
by (*metis* (*mono-tags*, *lifting*) *obtains-MAX mem-Collect-eq*)
qed

The main lemma of this section – if a full list satisfies the LexBFS property, then any element inside this list is a possible pick by the algorithm, if we reconstruct the state from the elements preceding it. This is the major step in showing completeness of the Binary LexBFS, i.e. that any list satisfying *is-lexbfs* can be produced by *lexbfs-bin*. The proof follows the mathematical proof from Brandstädt et al. [1].

lemma *candidate-if-is-lexbfs*:
assumes *is-lexbfs*: *is-lexbfs* (σ @ a # σ-*rest*) (**is** *is-lexbfs* ?σ-*all*)
shows $a \in$ *candidates* σ
proof (*rule ccontr*)
let ?*ia* = *length* σ **and** ?$\mathcal{B}$ = *construct-B* σ
let ?*a* = ?σ-*all*!?*ia*
assume $a \notin$ *candidates* σ — Contradiction-assumption

moreover from *is-lexbfs* **have** *U-at* σ = *set* (a # σ-*rest*)
by (*simp add*: *U-at-eq-rest*)

moreover from *is-lexbfs* **obtain** x **where** $x \in$ *candidates* σ

using *obtains-candidate*
by *blast*

ultimately have $x \in$ *set* σ*-rest* **by** *auto*
then obtain i **where** $i <$ *length* σ*-rest* σ*-rest*$!i = x$
by (*meson in-set-conv-nth*)
hence *?*σ*-all*$!$ (*length* σ + *Suc* i) $= x$ (**is** *?*$x = x$)
by (*metis nth-Cons-Suc nth-append-length-plus*)
let *?ix* = *length* σ + *Suc* i
from ‹*U-at* σ = *set* (a # σ*-rest*)› **have** $a \in$ *U-at* σ
by *simp*
with ‹$x \in$ *candidates* σ› ‹$a \notin$ *candidates* σ› **have** *?*$\mathcal{B}$ $a <$ *?*$\mathcal{B}$ x
by (*fastforce simp add*: *order-less-le*)
hence *?*$\mathcal{B}$ *?a* $<$ *?*$\mathcal{B}$ *?x*
using ‹*?x* = x›
by *simp*
then obtain *ib*
where *ib* $<$ *length* σ
and *?*$\mathcal{B}$ *?a* ! *ib* $<$ *?*$\mathcal{B}$ *?x* ! *ib*
and $\forall$ $j <$ *ib* . *?*$\mathcal{B}$ *?a* ! j = *?*$\mathcal{B}$ *?x* ! j
by (*metis list-lt-iff-ind-lt construct-B-len-eq-sigma*)
let *?b* = *?*σ*-all*$!$*ib*
from ‹*ib* $<$ *length* σ› ‹*?*$\mathcal{B}$ *?a* ! *ib* $<$ *?*$\mathcal{B}$ *?x* ! *ib*› **have** *?b* $\sim$ *?x* **and** *?b* $\sim*$ *?a*
using *construct-B-iff-connected*
by *fastforce+*

moreover from ‹$i <$ *length* σ*-rest*› ‹*ib* $<$ *length* σ› **have** *triple ?*σ*-all ib ?ia ?ix*
by *simp*

ultimately obtain *iz* **where** *iz* $<$ *ib* *?*σ*-all*$!$*iz* $\sim*$ *?x* *?*σ*-all*$!$*iz* $\sim$ *?a*
using *is-lexbfs lexbfs-prop-def* **unfolding** *is-lexbfs-def*
by *blast*
hence *?*$\mathcal{B}$ *?a* ! *iz* $>$ *?*$\mathcal{B}$ *?x* ! *iz* **using** ‹*ib* $<$ *length* σ›
using *construct-B-iff-connected*
by *fastforce*
thus *False* **using** ‹$\forall j$<*ib*. *?*$\mathcal{B}$ *?a* ! j = *?*$\mathcal{B}$ *?x* ! j› ‹*iz* $<$ *ib*›
by *simp*
qed

5.1.2 Preservation of the constructed state

For this step, we want to talk about specific steps of the *binary* algorithm. In the following, we need to show that the reconstructed state is preserved at each step, i.e. that what we reconstruct corresponds to what the algorithm outputs. For this, it will be useful to be able to keep parts of the loop from unfolding, and enabling starting the algorithm at a separate step, hence the three sub-definitions in the following.

definition *bin-step* $\equiv \lambda$ (U, $\mathcal{B}$, σ) . *do* {

```
  x <- RES {x ∈ U . B x = Max (B ‘ U)};
  let U = U - {x};
  let B = (λy . B y @ [x ~ y]);
  RETURN (U, B, σ @ [x])
}
```

definition *bin-loop* ≡ $WHILE_T$ (λ(*U*, B, σ) . *U* ≠ {}) *bin-step*

Note that, disregarding the added invariant, *lexbfs-bin-in-steps* = *lexbfs-bin*. We therefore name any theorem relating to *lexbfs-bin-in-steps* with the name *lexbfs-bin*. At the end, when we have shown LexBFS to be correct (i.e. sound and complete), we will use this to prove that *lexbfs-bin-in-steps* = *lexbfs-bin*.

```
definition lexbfs-bin-in-steps ≡ do {
  (U, B, σ) <- bin-loop (bin-state-at []);
  RETURN σ
}
```

We also need that *bin-loop* cannot fail. This amounts to just carefully unfolding the definition the right amount, as it follows from *U* being finite and not empty.

```
lemma nofail-bin-loop: finite U ⟹ nofail (bin-loop (U, B, σ))
  apply (induct card U arbitrary: U B σ)
  subgoal by (simp add: bin-loop-def WHILET-unfold)
  subgoal
    by (subst bin-loop-def,
        subst WHILET-unfold,
        subst bin-loop-def[symmetric],
        simp add: pw-bind-nofail bin-step-def)
  done
```

The following are many lemmas that show that the possible results starting at the constructed state at σ @ [*a*] are contained in the results starting at σ.

Abbreviation for *inres*, that flips order and adds infix notation, to make results more readable. Note that the ∈ is bold, unlike normal set-membership.

abbreviation *inres-rev* (**infix** ∈ *50*) **where** *inres-rev x y* ≡ *inres y x*

Next element in list can be picked by *bin-step*.

```
lemma inres-bin-step-if-is-lexbfs:
  is-lexbfs (σ @ a#σ') ⟹ bin-state-at (σ @ [a]) ∈ bin-step (bin-state-at σ)
  using candidate-if-is-lexbfs
  by(auto simp add: U-at-next bin-step-def pw-bind-inres)
```

Full list can be output, starting at any split.

```
lemma inres-bin-loop-if-is-lexbfs:
  is-lexbfs (σ @ σ') ⟹ bin-state-at (σ @ σ') ∈ bin-loop (bin-state-at σ)
```

proof (*induct* σ' *arbitrary*: σ)
case (*Cons a* σ' σ) — Full list is σ @ *a* # σ', induction hypothesis applies to σ @ [*a*]
then have *bin-step* (*bin-state-at* σ) $\gg=$ *bin-loop* $\leq$ *bin-loop* (*bin-state-at* σ)
by(*simp add*: *WHILET-unfold bin-loop-def U-at-not-empty*)

moreover from *Cons.prems* **have** *bin-state-at* (σ @ [*a*]) $\in$ *bin-step* (*bin-state-at* σ)
using *inres-bin-step-if-is-lexbfs*
by *simp*

ultimately have *bin-loop* (*bin-state-at* (σ @ [*a*])) $\leq$ *bin-loop* (*bin-state-at* σ)
by (*auto simp add*: *pw-bind-le-iff nofail-bin-loop finite-vertex-set*)

moreover from *Cons* **have** *bin-state-at* (σ @ *a*#σ') $\in$ *bin-loop* (*bin-state-at* (σ @ [*a*]))
by (*metis append.assoc append-Cons empty-append-eq-id*)

ultimately show *bin-state-at* (σ @ *a* # σ') $\in$ *bin-loop* (*bin-state-at* σ)
by (*simp add*: *pwD2*)
qed (*simp add*: *WHILET-unfold is-lexbfs-def bin-loop-def*)

Full list is valid output.

lemma *inres-lexbfs-bin-if-is-lexbfs*: *is-lexbfs* $\sigma \Longrightarrow \sigma \in$ *lexbfs-bin-in-steps*
using *inres-bin-loop-if-is-lexbfs*[*of* [] σ]
by (*auto simp add*: *pw-bind-inres lexbfs-bin-in-steps-def*)

lemma *lexbfs-bin-complete*: *SPEC is-lexbfs* $\leq$ *lexbfs-bin-in-steps*
by (*simp add*: *lhs-step-SPEC*[*folded inres-def*] *inres-lexbfs-bin-if-is-lexbfs*)

5.2 *lexbfs-bin* **refines** *lexbfs*

The inverse sequence of refinements. We construct the ordinary label function $\mathcal{L}$ by first constructing $\mathcal{B}$, and then transforming it.

abbreviation *R-bin-helper* $\equiv$ {((U, $\mathcal{B}$, σ), b).
b = (*card* U, U, *transform-B* (*construct-B* σ), σ) $\wedge$
$\mathcal{B}$ = *construct-B* σ $\wedge$
U = *U-at* σ $\wedge$
partition-V U σ}

lemma *lexbfs-bin-refines-lexbfs-helper*: *lexbfs-bin-in-steps* $\leq \Downarrow$ *Id lexbfs-helper*
unfolding *lexbfs-helper-def lexbfs-bin-in-steps-def bin-loop-def bin-step-def*
apply (*refine-rcg WHILET-refine*[**where** R = *R-bin-helper*])
subgoal by (*simp*)
subgoal by *simp*
subgoal by (*auto simp add*: *transform-B-refl-Max*)
subgoal by (*auto simp add*: *partition-V-next U-at-next card-V-split-if-partition-V*)
subgoal by *simp*

done

lemma *lexbfs-helper-refines-lexbfs*: *lexbfs-helper* $\leq \Downarrow$ *Id lexbfs*
unfolding *lexbfs-def lexbfs-helper-def*
by (*refine-rcg WHILET-refine*[**where** R = *R-lexbfs-helper*]) *auto*

5.3 Completeness of *lexbfs*

theorem *lexbfs-complete*: *SPEC is-lexbfs* $\leq$ *lexbfs*
using
lexbfs-helper-refines-lexbfs
lexbfs-bin-refines-lexbfs-helper
lexbfs-bin-complete
by *simp*

6 Correctness

corollary *lexbfs-correct*: *lexbfs* = *SPEC is-lexbfs*
using *lexbfs-complete lexbfs-sound*
by *simp*

By squeezing the different algorithms in between *lexbfs* and *SPEC is-lexbfs*, we can show they are all equivalent. In particular, none of the algorithms can fail.

corollary *lexbfs-equiv*:
lexbfs = *lexbfs-bin*
lexbfs = *lexbfs-helper*
lexbfs = *lexbfs-bin-in-steps*
using
lexbfs-bin-complete
lexbfs-bin-refines-lexbfs-helper
lexbfs-helper-refines-lexbfs-bin
lexbfs-bin-sound
lexbfs-correct
by *simp-all*

corollary *lexbfs-nofail*:
nofail lexbfs
nofail lexbfs-helper
nofail lexbfs-bin
nofail lexbfs-bin-in-steps
using *lexbfs-correct lexbfs-equiv*
by *simp-all*

end — Graph locale end

References

[1] A. Brandstädt, F. F. Dragan, and F. Nicolai. LexBFS-orderings and powers of chordal graphs. *Discret. Math.*, 171(1–3):27–42, 1997.

[2] D. G. Corneil. Lexicographic breadth first search—a survey. In J. Hromkovič, M. Nagl, and B. Westfechtel, editors, *WG 2004*, volume 3353 of *LNCS*, pages 1–19. Springer, 2004.

[3] D. G. Corneil, M. Habib, C. Paul, and M. Tedder. A recursive linear time modular decomposition algorithm via LexBFS. *CoRR*, abs/0710.3901, 2024.

[4] M. C. Golumbic. Triangulated graphs. In *Algorithmic Graph Theory and Perfect Graphs*, chapter 4, pages 81–104. Academic Press, 1980.

[5] Lexicographic breadth-first search. Wikipedia, https://en.wikipedia.org/w/index.php?title=Lexicographic_breadth-first_search&oldid=1314187570, Sept. 2025. Accessed: Oct. 25, 2025.

[6] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.