

# Project Report

## Binary Heap and Dijkstra's Algorithm

Josefine Lindmar Sonja Joost

January 8, 2026

### 1 Introduction

Explain high level idea, name all challenges addressed in the upcoming sections

### 2 Binary Heap

Array vs Tree

### 3 Dijkstra's algorithm

**Goal.** Formalize Dijkstra's algorithm over finite graphs and prove correctness, using a priority queue abstraction. Our Lean development lives in `Projects/Lindmar_Joost/Dijkstra.lean`.

We represent the graph as a finite simple graph so vertices have decidable equality (exactly how we have seen it in the BinarySort examples in class). Distances use extended natural numbers (so an unknown distance can be “infinity”), called `ENat` in Lean. The algorithm state is just a pair: a distance map from vertices to these extended naturals (`dist : V → N∞`) and a binary-heap priority queue (`queue : BinaryHeap`). The heap exposes abstract operations (empty check, extract-min, decrease-priority, size) and the proofs only rely on their contracts.

The algorithm is split up into three parts:

- Relaxation (Lean: `relaxNeighbors`) is a simple loop over the neighbors of a vertex  $u$  that updates each neighbor  $v$  by setting  $dist(v)$  to the minimum of its current value and  $dist(u)+1$ , and it conceptually decreases  $v$ 's priority in the heap when that happens.
- The recursive core of Dijkstra (Lean: `dijkstra_rec`) repeatedly extracts the current minimum  $u$ , relaxes  $u$ 's neighbors (updating the distance map and heap), and then recurses on the new state.
- The top-level function (Lean: `dijkstra`) initializes distances with 0 at the source and infinity elsewhere, inserts the source into the heap, and starts the recursion.

### Termination

We split the implementation into three parts: `dijkstra` (a non-recursive wrapper), `dijkstra_rec` (the recursive core), and `relaxNeighbors` (implemented as a fold over the finite list of neighbors). Hence termination needs to be shown only for `dijkstra_rec`: `dijkstra` is trivially terminating because it does not recurse, and `relaxNeighbors` terminates by structural recursion on a finite list (Lean accepts it as a terminating fold), so no additional termination proof is required for them.

We prove termination by the measure `queue.sizeOf`: if `queue.isEmpty` the recursion stops, otherwise extracting the minimum yields  $(u, \text{queue}')$  and `BinaryHeap.sizeOf_extract_min_lt_of_isEmpty_eq_false` gives  $\text{queue}'.\text{sizeOf} < \text{queue}.\text{sizeOf}$ ; since `relaxNeighbors` does not increase the heap size the post-relax queue has strictly smaller size than the original, so the `queue.sizeOf` measure decreases on every recursive call. In Lean this is written as `termination_by queue.sizeOf` and

discharged by applying `BinaryHeap.sizeOf_extract_min_lt_of_isEmpty_eq_false` plus the trivial bookkeeping to turn the hypothesis into the expected `isEmpty = false` form.

## Correctness proof (`dijkstra_correctness`)

*Statement.* For any vertex  $v$ , the algorithm's output equals the graph distance  $\delta$ :  $(\text{dijkstra } g s v) = \delta(g, s, v)$ .

The proof proceeds by fixing the algorithm's output and deriving a contradiction from any alleged disagreement with the true graph distance. Concretely, set

$$\text{dist} := \text{dijkstra } g s v$$

(in Lean: `set dist := (dijkstra g s v) with hdist`). First establish the global lower bound

$$\forall u, \delta(s, u) \leq \text{dist } u,$$

provided by the lemma `neverUnderestimates`. Note also that  $\delta(s, s) = 0$  and by `dijkstra_source_zero` the algorithm assigns 0 to the source, so any counterexample cannot be the source.

Assume, for contradiction, that there exists a vertex  $u$  with  $\text{dist } u \neq \delta(s, u)$ . Choose such a  $u$  that minimizes  $\delta(s, u)$  (Lean: `minimalCounterexample`). By the previous remark we have  $\delta(s, u) > 0$ , so take a shortest path from  $s$  to  $u$ ; this yields a predecessor  $y$  with  $y \sim u$  and

$$\delta(s, u) = \delta(s, y) + 1$$

(Lean: `existsPredOnShortestPath`). By minimality of  $u$  we have  $\delta(s, y) < \delta(s, u)$ , hence  $\text{dist } y = \delta(s, y)$ .

Now apply the algorithmic per-edge final bound `relaxAdj_final_bound` to the edge  $y \sim u$ . That lemma (which internally uses `exists_extract_or_top`, `relaxNeighbors_adj_upper`, and the  $y$ -stability lemmas such as `extracted_value_is_final_lemma`) yields

$$\text{dist } u \leq \text{dist } y + 1.$$

Substituting  $\text{dist } y = \delta(s, y)$  and  $\delta(s, y) + 1 = \delta(s, u)$  gives  $\text{dist } u \leq \delta(s, u)$ . Together with the global lower bound  $\delta(s, u) \leq \text{dist } u$  this forces equality  $\text{dist } u = \delta(s, u)$ , contradicting the choice of  $u$ .

Hence no counterexample exists, and we conclude `dijkstra_correctness`: for every vertex  $v$  the algorithm's output equals the graph distance  $\delta(g, s, v)$ .

**Preconditions and placeholders.** The development assumes the standard `BinaryHeap` contracts and builds the correctness proof from a number of named lemmas, many of which in turn rely on smaller auxiliary facts. Figure 1 illustrates the dependency relation between the main lemmas used in this file; For space and clarity some minor lemmas are not shown in the diagram. All of the lemmas have been fully formalized except for the search lemma `exists_extract_or_top`, which is currently left as a `sorry`, and the lower bound leamm `neverUnderestimates`. A discussion of why those particular proof remain unfinished appears in the "Reflection on Efforts" section at the end of the report.

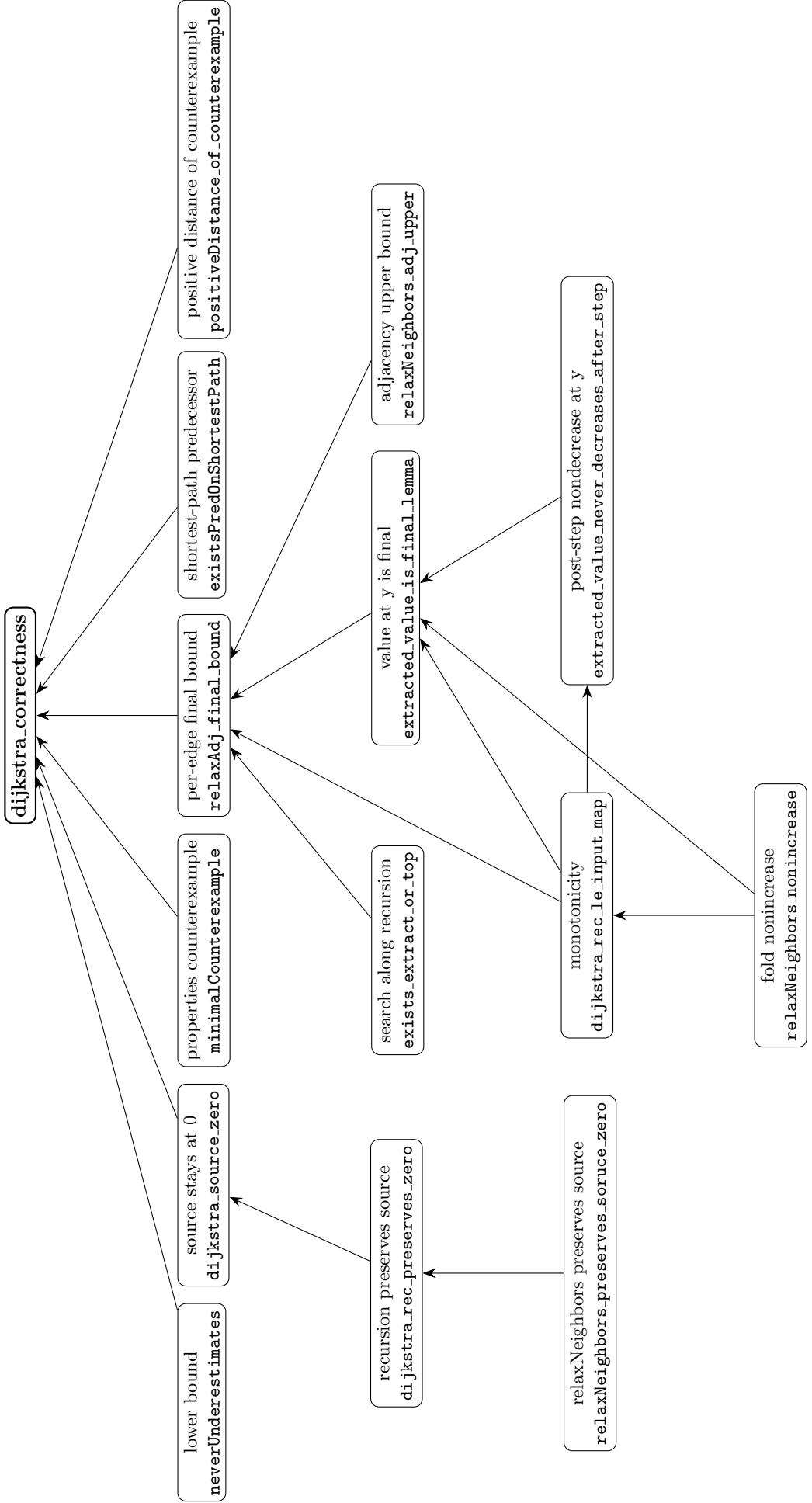


Figure 1: Lemma dependency graph shaping the proof of correctness.

## Monotonicity of the recursion

*Statement.* Distances never increase along the recursion:

$$(\text{dijkstra\_rec } g s t \text{ dist queue}) x \leq \text{dist } x.$$

*English proof.* By strong induction on `sizeOf(queue)`. If empty, the result is `dist`. Otherwise, after one step we recurse on a strictly smaller heap to obtain  $\text{dist}' x \leq \text{dist } x$  using the IH and the fact that `relaxNeighbors` is pointwise nonincreasing.

*Lean mapping.* - Lemma `dijkstra_rec_le_input_map`: generalize the size to  $n$ , revert state, apply `Nat.strong_induction_on`, split on `isEmpty`, perform one step with named equations (`step`, `u`, `queue'`, `next`), use size lemmas to apply the IH, then compose with `relaxNeighbors_nonincrease`.

## Properties of relaxation

*Pointwise nonincrease.* Folding does not increase any coordinate, since each step either keeps a value or lowers it. *Lean:* `relaxNeighbors_nonincrease`, proved by induction on the neighbor list. *Edge-wise upper bound.* If  $y \sim u$ , then after relaxing neighbors of  $y$  once, the tentative value at  $u$  is at most  $\text{dist } y + 1$ . *Lean:* `relaxNeighbors_adj_upper`, proved by list induction plus membership of  $u$  in the neighbor list.

## Extraction stability at a vertex

*Idea.* Once  $y$  is extracted, its value equals the value immediately after relaxing its neighbors and never decreases later. This captures Dijkstra's "finalization" of a node.

*Lean mapping.* - `MinGeYInvariant y (dist,queue)`: next extracted key  $\geq \text{dist } y$ . - `extracted_value_never_decreases`: strong induction on the post-step heap size; the motive carries the invariant from `hInvPreserve`. Concludes  $\text{next}.1 y \leq (\text{dijkstra\_rec } g s t \text{ next}.1 \text{ next}.2) y$ . - `extracted_value_is_final_lemma`: combine monotonicity  $((\cdot) y \leq \text{dist } y)$ , preservation of  $y$  during relaxation ( $\text{next}.1 y = \text{dist } y$ ), and the previous lemma to get equality at  $y$ .

## Search along the recursion: extract-or-top

*Statement.* Either the final value at  $y$  is  $\top$ , or there is a recursion state where the next extraction is  $y$  and one step of `dijkstra_rec` is definitionally equal to continuing from `relaxNeighbors g y ..`

*English proof.* Run strong induction on the heap size while keeping the concrete state in scope. If the queue is empty then the final value at  $y$  is whatever sits in `dist` (in particular possibly  $\top$ ). Otherwise, perform one step: if the extracted vertex is  $y$  we found the desired predecessor state; if not, recurse on the strictly smaller post-step heap until either  $y$  is extracted or the final is  $\top$ .

*Lean mapping.* Lemma `exists_extract_or_top` (in progress): the induction predicate quantifies over all `dist, q` with a size bound, and the step uses named equalities for `extract_min` and `relaxNeighbors` so the one-step equality follows by definitional `simp`.

## Final edge bound in the output map

*Statement.* For any edge  $y \sim u$ , the output satisfies

$$(\text{dijkstra } g s t) u \leq (\text{dijkstra } g s t) y + 1.$$

*English proof.* Use the extract-or-top dichotomy for  $y$ . If the final value at  $y$  is  $\top$ , the inequality holds trivially. Otherwise, identify the first state extracting  $y$ ; at that step, the relaxation at  $y$  ensures the post-step tentative value at  $u$  is  $\leq (\text{dist } y + 1)$ . Stability of  $y$  then transports this bound to the final output.

*Lean mapping.* Lemma `relaxAdj_final_bound`: case-split on the search lemma, apply `relaxNeighbors_adj_upper` at the predecessor state, use `extracted_value_is_final_lemma` and monotonicity to lift it to the final map.

## Global correctness (plan)

*Never underestimates.* Show  $\delta(s, u) \leq (\text{dijkstra } g\ s\ t)\ u$  by induction on shortest paths (lemma `neverUnderestimates`).

*Shortest-path predecessor.* For  $u$  with  $\delta(s, u) > 0$ , there exists  $y$  with  $y \sim u$  and  $\delta(s, u) = \delta(s, y) + 1$  (lemma `existsPredOnShortestPath`), using Mathlib walks and minimality.

*Minimal counterexample.* Assume a counterexample  $v$ , pick minimal  $u$ . Use the predecessor  $y$ , its correctness by minimality, and the edge bound to get an upper bound  $(\text{dijkstra } g\ s\ v)\ u \leq \delta(s, u)$ . Combined with the lower bound, obtain equality, contradiction (theorem `dijkstra_correctness`).

## English–Lean correspondence (by lemma)

- **Algorithm/core:** `dijkstra_rec`, `dijkstra` - **Relaxation primitives:** `relaxNeighbors`, `relaxNeighbors_nonincreasing` - **Heap size laws:** `sizeOf_extract_min_lt_of_isEmpty_eq_false` - **Local invariant + stability:** `MinGeYInvariant`, `extracted_value_never_decreases_after_step`, `extracted_value_is_final_lemma` - **Search lemma:** `exists_extract_or_top` (with strong induction on heap size) - **Per-edge bound and finish:** `relaxNeighbors_adj_upper`, `relaxAdj_final_bound`, `neverUnderestimates`, `existsPredOnShortestPath`, `dijkstra_correctness`

## Open obligations still to discharge

- Implement and verify the concrete heap operations and their size laws. - Complete list-based inductive sublemmas inside `relaxNeighbors`. - Finish the remaining induction engineering in the search lemma and stabilize the few marked `sorry/admit`.

## Proof walkthrough and dependencies

*How the theorems interact.* Below we list the main lemmas, the direction they prove, and what they need.

- **Termination of recursion** (in `dijkstra_rec`). Needs: heap facts  $\text{sizeOf}(\text{extract\_min}(q).2) < \text{sizeOf}(q)$ ,  $\text{sizeOf}(\text{relaxNeighbors}(\cdot).2) \leq \text{sizeOf}(\cdot)$ . Used by every strong-induction proof on the queue size.

- **Monotonicity** (`dijkstra_rec_le_input_map`). Needs: `relaxNeighbors_nonincrease`. Used by stability at  $y$  (to get one side of equality) and in edge-bound lifting.

- **Relaxation properties.** Two pieces: (i) `relaxNeighbors_nonincrease` (fold lemma), (ii) `relaxNeighbors_adj_upper` (single-step bound at an adjacent  $u$ ). Used in the per-edge bound.

- **Local invariant and stability at  $y$ .** The invariant `MinGeYInvariant` is preserved between steps (assumed via `hInvPreserve`). Then `exttextracted_value_never_decreases_after_step` (strong induction on post-step heap) + `exttextracted_value_is_final_lemma` yield that  $y$ 's value is fixed immediately after relaxing its neighbors. Used to propagate local bounds to the final map.

- **Search lemma** (`exists_extract_or_top`). Needs: strong induction over queue size with a predicate that keeps the concrete state visible; uses definitional equalities of a single recursion step to connect states. Used by `relaxAdj_final_bound` to jump to the step where  $y$  is handled.

- **Per-edge final bound** (`relaxAdj_final_bound`). Needs: `exists_extract_or_top`, `relaxNeighbors_adj_upper`, and the  $y$ -stability lemmas. Used directly in correctness.

- **Graph-theoretic path lemmas.** `existsPredOnShortestPath` (predecessor  $y$  on a shortest path) from Mathlib walk reasoning; independent of the heap proofs. Used in correctness to relate  $\delta(s, u)$  and  $\delta(s, y)$ .

- **Lower bound** (`neverUnderestimates`). Needs only graph reasoning and the algorithm's relaxation monotonicity; used in correctness to bound the output from below by true distance.

## Dependency summary.

- Termination facts → strong induction on heap size.
- Fold lemmas → monotonicity and single-step adjacency bound.
- Invariant + strong IH on post-step heap → stability of  $y$ .
- Search lemma + stability + adjacency bound → per-edge final bound.
- Predecessor on shortest path + per-edge bound + lower bound → correctness.

## 4 Reflection on Efforts

Task: "Describe the main challenges, one design decision you would reconsider, and what you learned about formalization."

Leave out: Challenges for Binary Heap

Challenges for Dijkstra: - automation failed really often with maximum recursion depth reached  
- induction setup was hard, finding out the syntax to get the correct induction hypothesis, one time i could not figure it out to the end (theorem exists\_extract\_or\_top) - i had to figure out the theorems needed with their proofs itself. we only found a super high level proof. all details i had to find out on my own

## 5 Reflection on LLM Usage

### References

- [1] Wikipedia, *Dijkstra's algorithm.* [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).
- [2] Lean4, Mathlib4, *SimpleGraph documentation.* [https://leanprover-community.github.io/mathlib4\\_docs/Mathlib/Combinatorics/SimpleGraph/Basic.html#SimpleGraph](https://leanprover-community.github.io/mathlib4_docs/Mathlib/Combinatorics/SimpleGraph/Basic.html#SimpleGraph).
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd edition, MIT Press, Cambridge, MA, 2009.