Project Report
Binary Heap and Dijkstra's Algorithm

Josefine Lindmar Sonja Joost

January 8, 2026

1 Introduction

TODO Brief overview and motivation

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` : $V \to \mathbb{N}_\infty$) 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, queue')` and `BinaryHeap.sizeOf_extract_min_lt_of_isEmpty_eq_false` gives `queue'.sizeOf` $<$ `queue.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.

Correcntess proof (`dijkstra_correctness`)

Statement. For any vertex v, the algorithm's output equals the graph distance δ:

$$(\texttt{dijkstra}\ g\ s\ v)\ v = \delta(g, s, v)$$

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

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

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

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

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 $\texttt{dist}\ u \neq \delta(s, u)$. 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 $\texttt{dist}\ y = \delta(s, y)$.
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

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

Substituting $\texttt{dist}\ y = \delta(s, y)$ and $\delta(s, y) + 1 = \delta(s, u)$ gives $\texttt{dist}\ u \leq \delta(s, u)$. Together with the global lower bound $\delta(s, u) \leq \texttt{dist}\ u$ this forces equality $\texttt{dist}\ u = \delta(s, u)$, 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 "Main Challenges" section at the end of the report.

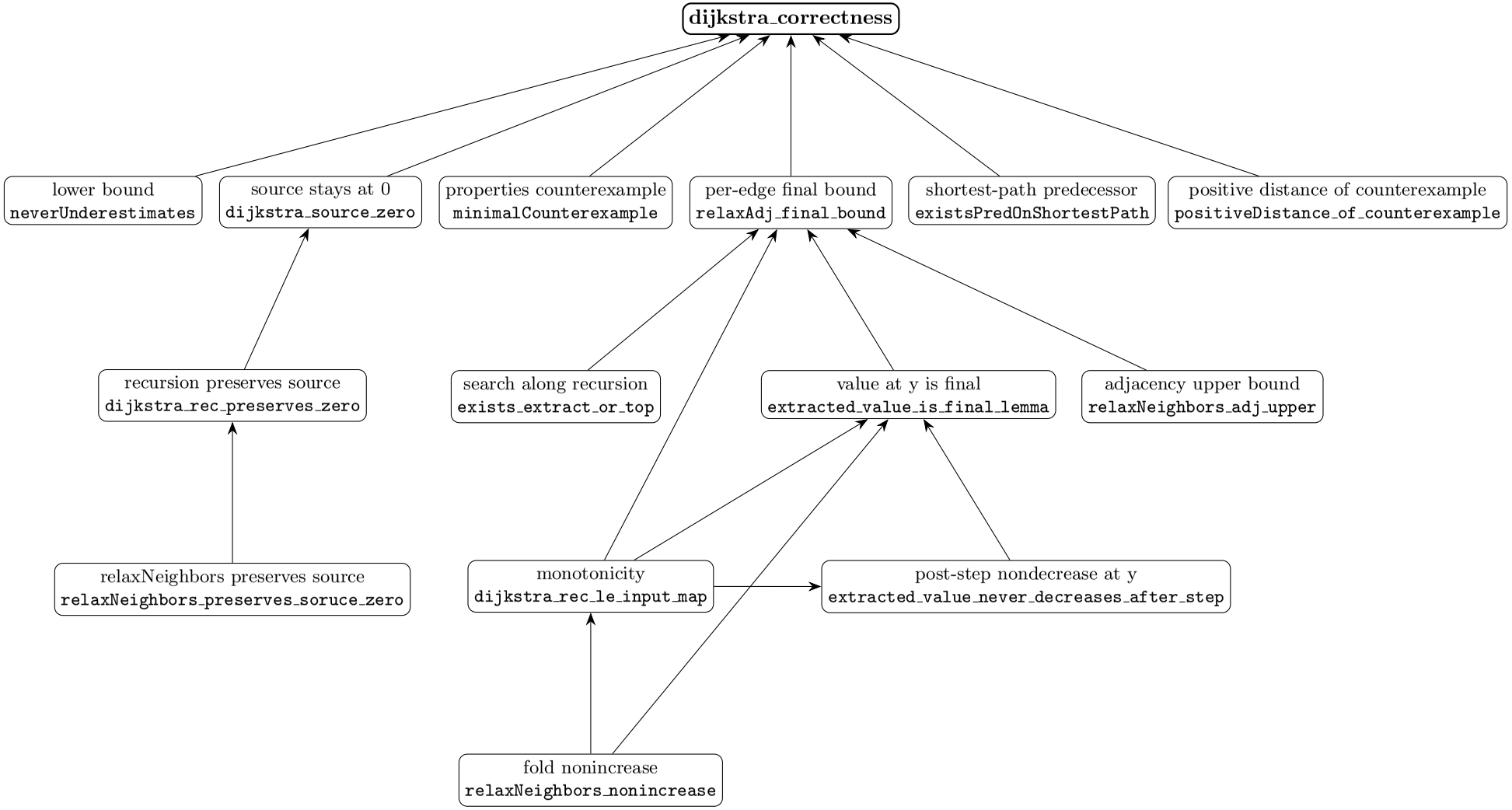

Figure 1: Lemma dependency graph shaping the proof of correctness.

Lower bound (`neverUnderestimates`)

Statement. For any vertex u, Dijkstra's output never underestimates the true graph distance:

$$(\texttt{dijkstra}\ g\ s\ t)\ u \geq \delta(g, s, u)$$

This lemma establishes a global lower bound on the algorithm's output. In the Lean file, `neverUnderestimates` is currently left as `admit`. All online references we found assume this property as a given, and formalizing it requires reasoning about graph walks and path induction with Mathlib's `SimpleGraph.Walk` machinery, which was beyond the scope of this project. We therefore treat this lemma as an axiomatic starting point for our correctness proof.

Source preservation (`dijkstra_source_zero`, `dijkstra_rec_preserves_zero`, `relaxNeighbors_preserves_source_zero`)

Statement. The source vertex s always maintains distance 0 throughout the algorithm:

$$(\texttt{dijkstra}\ g\ s\ t)\ s = 0$$

English explanation. Dijkstra initializes $\texttt{dist}(s) = 0$ and $\texttt{dist}(v) = \top$ for all $v \neq s$. We must show that this invariant is preserved through all recursive steps. The key observation is that `relaxNeighbors` only updates neighbors of the extracted vertex u, and each update sets a neighbor's distance to $\texttt{dist}(u) + 1$. Since the graph is simple (irreflexive), s is never a neighbor of itself, so it can only be updated if it appears as a neighbor of some other vertex u. But even if s were a neighbor of u, the update condition $\texttt{dist}(u) + 1 < \texttt{dist}(s)$ would require $\texttt{dist}(u) + 1 < 0$, which is impossible in extended naturals. Therefore $\texttt{dist}(s)$ remains unchanged at 0 after every relaxation step, and by induction on the recursion depth (using strong induction on the heap size), it stays 0 in the final output.
Lean correspondence. The proof is split into three lemmas in a bottom-up fashion. `relaxNeighbors_preserves_source_zero` proves preservation for a single relaxation step by unfolding the fold over neighbors and showing that the update branch never triggers for the source (the condition $\texttt{dist}(u) + 1 < 0$ is false, which we encode as `enat_add_one_not_lt_zero`). We proceed by induction on the neighbor list, proving that each step preserves $\texttt{dist}(s) = 0$. Next, `dijkstra_rec_preserves_zero` lifts this to the full recursion via strong induction on `sizeOf(queue)`: if the queue is empty the result is immediate; otherwise we extract u, apply the previous lemma to get $\texttt{dist}'(s) = 0$ after relaxation, then apply the induction hypothesis to the smaller post-step heap. Finally, `dijkstra_source_zero` invokes the recursive lemma on the initial state.

Minimal counterexample (`minimalCounterexample`, `positiveDistance_of_counterexample`)

Statement. If there exists a vertex where Dijkstra's output disagrees with the true distance, then among all such counterexamples there exists one u with minimal $\delta(s, u)$, and moreover for every w with $\delta(s, w) < \delta(s, u)$, we have $\texttt{dist}(w) = \delta(s, w)$. Furthermore, any such minimal counterexample must satisfy $\delta(s, u) > 0$, i.e., $u \neq s$.
English explanation. This is a standard proof-by-contradiction setup. We assume for contradiction that the set S of counterexamples (vertices where $\texttt{dist}(u) \neq \delta(s, u)$) is nonempty. Since the vertex set is finite, we can pick an element $u \in S$ that minimizes $\delta(s, u)$ over all counterexamples. By definition of minimality, any vertex w with strictly smaller distance must not be a counterexample, hence $\texttt{dist}(w) = \delta(s, w)$. The second part shows that $u \neq s$ because the source is correctly initialized to distance 0, which matches $\delta(s, s) = 0$, so the source cannot be a counterexample. Therefore $\delta(s, u) > 0$.
Lean correspondence. `minimalCounterexample` formalizes the minimality argument. We define S as `Finset.univ.filter` on the predicate $\texttt{dist}\ u \neq \delta\ g\ s\ u$, prove S is nonempty using the hypothesis, and apply `Finset.exists_min_image` to obtain a vertex $u \in S$ minimizing $\delta(s, \cdot)$. The minimality property follows by showing that if any w satisfies $\delta(s, w) < \delta(s, u)$ and were also a counterexample, then $\delta(s, u) \leq \delta(s, w)$ by the minimality of u, contradicting the strict

inequality. The auxiliary lemma `positiveDistance_of_counterexample` uses the hypothesis $u \neq s$ and connectivity of the graph to invoke Mathlib's `SimpleGraph.Reachable.dist_eq_zero_iff`, which states that $\delta(s,u) = 0$ if and only if $u = s$. Since $u \neq s$, we have $\delta(s,u) \neq 0$, hence $\delta(s,u) > 0$.

Shortest-path predecessor (`existsPredOnShortestPath`)

Statement. If $\delta(s,u) > 0$, there exists a vertex y adjacent to u such that $\delta(s,u) = \delta(s,y) + 1$.
English explanation. This lemma captures the simple fact that if u is not the source, any shortest path from s to u must traverse at least one edge, and the last edge of such a path provides the desired predecessor. Concretely, take a shortest walk p from s to u with $p.\texttt{length} = \delta(s,u)$. Reverse it to obtain a walk from u to s. Since $\delta(s,u) > 0$, this reversed walk is nonempty and begins with an edge from u to some vertex y. The tail of this reversed walk is a walk from y to s of length $\delta(s,u) - 1$, hence $\delta(s,y) \leq \delta(s,u) - 1$. Conversely, the triangle inequality and the edge $y \sim u$ give $\delta(s,u) \leq \delta(s,y) + 1$. Combining these yields equality.
Lean correspondence. The proof uses Mathlib's `SimpleGraph.Walk` API. We first invoke `SimpleGraph.Reachable.exists_walk_length_eq_dist` to obtain a shortest walk p : `Walk` s u. We then case-analyze `p.reverse`: if it's `nil`, the length is zero, contradicting $\delta(s,u) > 0$; otherwise it's a `cons` with head adjacency $u \sim y$ and a tail walk from y to s. We compute that the reversed walk's length equals $\texttt{tail.length}+1 = \delta(s,u)$, so $\texttt{tail.length} = \delta(s,u)-1$. Since distance is the infimum of walk lengths, $\delta(s,y) \leq \texttt{tail.length}$, giving $\delta(s,y) + 1 \leq \delta(s,u)$. The reverse inequality $\delta(s,u) \leq \delta(s,y) + 1$ comes from the triangle inequality (formalized as `delta_adj_step_ENat`). Antisymmetry yields equality.

Monotonicity (`dijkstra_rec_le_input_map`)

Statement. For any vertex x, the recursive Dijkstra function never increases distances:

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

English explanation. This lemma formalizes the intuitive property that relaxation steps can only improve (decrease) distance estimates, never worsen them. We prove it by strong induction on `sizeOf(queue)`. If the queue is empty, `dijkstra_rec` returns `dist` unchanged, so the inequality is trivial. Otherwise, we unfold one recursion step: extract the minimum vertex u, relax its neighbors to obtain a new distance map $\texttt{dist}'$ and queue $\texttt{queue}'$, then recurse. The heap size strictly decreases after extraction and does not increase during relaxation, so $\texttt{sizeOf(queue)'} < \texttt{sizeOf(queue)}$. By the induction hypothesis applied to the smaller heap, the final result satisfies $(\texttt{dijkstra\_rec}\ \cdots\ \texttt{dist}'\ \texttt{queue}')\ x \leq \texttt{dist}'\ x$. The pointwise nonincrease property of `relaxNeighbors` gives $\texttt{dist}'\ x \leq \texttt{dist}\ x$. Composing these two inequalities via transitivity yields the desired bound.
Lean correspondence. The proof structure mirrors the English argument. We generalize the heap size to a parameter n, revert the state variables, and apply `Nat.strong_induction_on`. The base case handles `queue.isEmpty = true` trivially. In the inductive step, we bind the extraction result with `let` statements (`step`, `u`, `queue'`, `next`) and apply two size lemmas: `BinaryHeap.sizeOf_extract_min_lt_of_` for the strict decrease and `sizeOf_relaxNeighbors_le` for the nonincrease. The induction hypothesis then applies to the strictly smaller post-step heap size. Finally, `relaxNeighbors_nonincrease` provides the one-step inequality $\texttt{dist}'\ x \leq \texttt{dist}\ x$, and we compose with `le_trans`.

Relaxation properties (`relaxNeighbors_nonincrease`, `relaxNeighbors_adj_upper`)

Statement (nonincrease). For all vertices x,

$$(\texttt{relaxNeighbors}\ g\ u\ \texttt{dist}\ q).1\ x \leq \texttt{dist}\ x$$

English explanation. Relaxation is a fold over the neighbor list of u. At each step, for a neighbor v, we compute $\texttt{alt} = \texttt{dist}(u) + 1$ and conditionally update $\texttt{dist}(v)$ to $\min(\texttt{dist}(v), \texttt{alt})$. If the update happens, $\texttt{dist}(v)$ decreases; otherwise it stays the same. Vertices other than the current

neighbor are untouched in that step. By induction on the neighbor list, every coordinate of the distance map either decreases or stays the same, hence the final map is pointwise $\leq$ the input map.
Lean correspondence. We prove `relaxNeighbors_nonincrease` by induction on the neighbor list. The fold function f is unfolded explicitly: if $\texttt{alt} < \texttt{dist}(v)$, we set $\texttt{dist}'(x) = \texttt{alt}$ for $x = v$ and $\texttt{dist}(x)$ otherwise; else the map is unchanged. The base case (empty list) is trivial. In the inductive step, we show that one application of f preserves the nonincrease property: for any x, $f(\texttt{dist}, q)\ v$ yields a map with $\texttt{dist}'(x) \leq \texttt{dist}(x)$. We then apply the induction hypothesis to the tail of the list and compose the inequalities.
Statement (adjacency upper bound). If $y \sim u$, then

$$(\texttt{relaxNeighbors}\ g\ y\ \texttt{dist}\ q).1\ u \leq \texttt{dist}(y) + 1$$

English explanation. When relaxing neighbors of y, the vertex u appears in the neighbor list (since $y \sim u$). At the step processing u, the alternative distance $\texttt{alt} = \texttt{dist}(y) + 1$ is compared with $\texttt{dist}(u)$. If $\texttt{alt} < \texttt{dist}(u)$, we update $\texttt{dist}(u) := \texttt{alt}$, immediately achieving the bound. If not, we already have $\texttt{dist}(u) \leq \texttt{alt}$ by linear order. Subsequent steps process other neighbors and do not change $\texttt{dist}(u)$ (since only the current neighbor's entry is updated). Therefore the final map satisfies the bound.
Lean correspondence. The proof of `relaxNeighbors_adj_upper` proceeds by showing that $u \in \texttt{neighborFinset}(y)$ (using `adj_mem_neighborFinset`), hence u appears in the fold's neighbor list. We define helper lemmas: `step_u_bound` shows that processing u itself yields the bound $\leq \texttt{dist}(y) + 1$, and `step_other_preserve` shows that processing any neighbor $v \neq u$ preserves the bound at u. We then use a master induction lemma `bound_if_mem_nodup` which, given a nodup list containing u, proves the bound by case-analyzing whether the list head equals u (apply `step_u_bound` and preserve over the tail) or not (recurse on the tail). This induction is applied to the neighbor list of y, yielding the final bound.

Extraction stability (`extracted_value_never_decreases_after_step`, `extracted_value_is_final_lemma`)

Statement. Once y is extracted from the queue, its distance value equals the value immediately after relaxing its neighbors and remains fixed in all subsequent recursion: $\texttt{dist}(y) = (\texttt{dijkstra\_rec}\ \cdots)\ y$ after extracting y.
English explanation. This pair of lemmas captures the key Dijkstra invariant that once a vertex is extracted (i.e., it has the minimum tentative distance among unprocessed vertices), its distance is final and never changes. The argument has two parts. First, `extracted_value_never_decreases_after_step` shows that after extracting y and relaxing its neighbors, the value $\texttt{dist}(y)$ never decreases in future recursion steps. This uses a local invariant `MinGeYInvariant`, which states that any future extracted vertex u has $\texttt{dist}(u) \geq \texttt{dist}(y)$. Relaxing neighbors of u can only decrease values to $\texttt{dist}(u)+1$, which is still $\geq \texttt{dist}(y)+1 > \texttt{dist}(y)$, so the relaxation cannot decrease $\texttt{dist}(y)$. By strong induction on the post-extraction heap size, this propagates through all subsequent steps. Second, `extracted_value_is_final_lemma` combines this non-decrease property with monotonicity (the final map is $\leq$ the post-step map) and the fact that relaxing y's neighbors does not change $\texttt{dist}(y)$ itself (since y is not its own neighbor in a simple graph). Together, these yield equality.
Lean correspondence. `extracted_value_never_decreases_after_step` is a strong induction on `sizeOf(next.2)`, where `next` is the post-extraction-and-relaxation state. The motive carries the invariant `MinGeYInvariant y p`, which is assumed to be preserved across recursion steps via the hypothesis `hInvPreserve`. In the base case (empty queue), the recursion stops and the inequality is trivial. In the inductive step, we extract the next vertex u_1, use the invariant to get $\texttt{dist}(y) \leq \texttt{dist}(u_1)$, then show that relaxing neighbors of u_1 preserves $\texttt{dist}(y)$ because the update condition $\texttt{dist}(u_1) + 1 < \texttt{dist}(y)$ is false. We then apply the induction hypothesis to the smaller heap and compose inequalities. `extracted_value_is_final_lemma` wraps this: it first proves that $\texttt{next}.1(y) = \texttt{dist}(y)$ by showing that the fold over y's neighbors never updates the y-coordinate (using irreflexivity of adjacency), then combines the non-decrease lemma with monotonicity to get both directions of the inequality, yielding equality.

Search lemma (`exists_extract_or_top`)

Statement. For any vertex y, either the final Dijkstra output at y is $\top$, or there exists a recursion state $(\texttt{dist}, q)$ where y is the next vertex to be extracted, and the final output equals the result of one more recursion step from that state.

English explanation. This lemma provides a case distinction that is crucial for the edge-bound argument. Intuitively, if the algorithm ever processes y (extracts it from the queue), we can identify the exact recursion state immediately before that extraction. If y is never extracted, it means the algorithm terminates with y still in the queue (or never added), in which case the final distance at y remains $\top$. The proof proceeds by strong induction on the heap size. If the queue is empty, the recursion stops and returns `dist`, so either $\texttt{dist}(y) = \top$ (first case) or the final value is whatever was in `dist`. Otherwise, extract the minimum vertex: if it equals y, we have found the desired state; if not, recurse on the strictly smaller post-step heap. By the induction hypothesis, either the final value at y is $\top$ (propagating the first case) or we find a later state extracting y (propagating the second case).

Lean correspondence. The lemma `exists_extract_or_top` is currently marked as `sorry` in the Lean file. The intended proof structure uses strong induction on `sizeOf(queue)` with a predicate that quantifies over all states $(\texttt{dist}, q)$ satisfying a size bound. The tricky part is ensuring that the one-step equality $(\texttt{dijkstra\_rec dist}_0\ \texttt{queue}_0) = (\texttt{dijkstra\_rec next}.1\ \texttt{next}.2)$ holds definitionally, which requires carefully naming the intermediate let-bindings (`hstep` for `extract_min`, `next` for `relaxNeighbors`) so that `simp` can unfold the recursion and match the left-hand side with the right-hand side. The difficulty encountered was shaping the induction hypothesis to simultaneously generalize over `dist` and q while keeping the size bound, and ensuring the IH applies correctly to the post-step heap without requiring auxiliary equalities that break definitional reduction. This lemma remains the main open engineering challenge in the development.

Final edge bound (`relaxAdj_final_bound`)

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

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

English explanation. This lemma is the algorithmic heart of the correctness proof: it shows that the output respects the edge-stepping property of graph distances. The proof uses the search lemma to case-split on whether y was ever extracted. If the final value at y is $\top$, then the right-hand side is $\top$ and the inequality holds trivially. Otherwise, we identify the recursion state $(\texttt{dist}, q)$ where y is about to be extracted. After extracting y and relaxing its neighbors, the relaxation lemma `relaxNeighbors_adj_upper` gives that the tentative distance at u is at most $\texttt{dist}(y) + 1$. The stability lemma shows that $\texttt{dist}(y)$ equals the final value at y. Monotonicity then lifts the post-relaxation bound at u to the final value at u, yielding the desired inequality.

Lean correspondence. The proof unfolds `dijkstra` and applies `exists_extract_or_top`. In the `inl` case (y's final value is $\top$), we use `le_top`. In the `inr` case, we have witnesses `dist` and q with $q.\texttt{extract\_min}.1 = y$ and an equality `hfinEq` linking the final map to the recursion from the post-step state `next`. We compute the post-extraction-and-relaxation state and invoke three key lemmas: `relaxNeighbors_adj_upper` to get $\texttt{next}.1(u) \leq \texttt{dist}(y) + 1$, `extracted_value_is_final_lemma` (given the initial invariant `hInvInit` and the preservation hypothesis `hInvPreserve`) to get $\texttt{dist}(y) = (\texttt{dijkstra\_rec next}.1\ \texttt{next}.2)\ y$, and `dijkstra_rec_le_input_map` for monotonicity at u. We chain these together: the final value at u (via `hfinEq`) equals the recursion from `next`, which by monotonicity is $\leq \texttt{next}.1(u)$, which by relaxation is $\leq \texttt{dist}(y) + 1$, which by stability equals the final value at y plus one. The proof concludes by rewriting using `hfinEq` and simplifying with `grind`.

4 Efforts & Reflection

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

4.1 Main Challenges

Challenges for Binary Heap:

TODO, Array vs Tree

Challenges for Dijkstra:
Proving the Dijkstra lemmas turned out to be much harder than expected. One major issue was that the automated tools in Lean often failed. For example, when we used `simp` to simplify expressions, it often got stuck because the recursion depth limit was reached. This happened a lot when we tried to prove equalities that depended on how the recursive function `dijkstra_rec` was defined. To fix this, we had to manually break the proofs into smaller steps and carefully control how the simplifications were applied.
Another big challenge was figuring out how to use induction correctly. Many proofs required us to use strong induction on the size of the heap, which is the data structure that keeps track of the vertices. But we also had to keep track of extra information, like the current distances and the queue of vertices to process. This meant we had to write very precise statements about what we wanted to prove at each step. Writing these statements and proving them took a lot of trial and error. We could figure it out for all but one lemma: The search lemma (`exists_extract_or_top`) is still unproven because we were not able to get the induction hypothesis right.
Overall, the hardest part of formalizing Dijkstra's algorithm was turning the intuitive ideas into precise proofs. The proof found relies on a lot of intuition, which had to be completely formalized. This required breaking everything into very small pieces and proving each piece separately. This process was slow and involved a lot of try and error.
In total, the time we needed to spend on the project was way greater than expected beforehand.

4.2 Reflection on LLM Usage

We tried using LLMs, but quickly found them to be largely unusable. A major reason is that Lean is not backward-compatible: many facts from older versions of mathlib that LLMs rely on no longer exist. As a result, it was extremely rare for an LLM to produce Lean code that actually compiled.
That said, LLMs were occasionally helpful for very small steps. Given a specific tactic state and a clear goal—such as applying a particular lemma together with a known fact—the model could sometimes suggest one or a few Lean lines that worked. However, this only applied to minor tactic state changes and did not scale beyond that.

4.3 One design decision we would reconsider

Make distances and priority queue a fixed pair.

4.4 What we have learned