Project Report
Binary Heap and Dijkstra's Algorithm

Josefine Lindmar Sonja Joost

January 15, 2026

1 Introduction

This report presents a Lean formalization of a verified binary heap and a correctness development for Dijkstra's shortest-path algorithm that uses the heap as its priority queue. The heap part adopts a tree-based representation and implements the core operations ('heapify', 'insert', 'extract_min', 'decrease_key'), together with correctness proofs that these operations preserve the heap invariant, maintain membership and size properties, and satisfy behavioral laws required by Dijkstra's algorithm. These verified properties form a clean priority-queue abstraction that the algorithmic layer relies on. The Dijkstra development implements neighbor relaxation as a fold and the recursive driver 'dijkstra_rec', and proves a number of lemmas used in the standard correctness argument; these are combined via a minimal-counterexample proof to establish the algorithm's correctness.

2 Binary Heap

Goal. The goal of the binary heap part of the project is to formalize the binary heap together with the operations needed for the Dijstra algorithm. This included the following:

- `heapify`: Create a heap from a given array
- `insert`: Add a new key to the heap and restore the binary heap invariant by bubbling the element to its correct position.
- `extract-min (/max)`: Pop the minimum (resp. maximum) element from the heap and restore the binary heap invariant by placing a leaf node at the root and trickling it down to the position it belongs.
- `decrease-key (/increase-key)`: Decrease (resp. increase) the value of an element's key to the new value, which is assumed to be less than or equal to the current key's value and restoring the heap invariant such that the node is at the correct position according to its new value.

Data Structure

A binary heap is a data structure that represents data as a nearly complete binary tree [**?**]. The binary heap must satisfy the heap condition, which depends on the type of heap. There are min and max heaps, where the root is the smallest or highest value, respectively. In a min heap, the value of each node is less than or equal to its parent's value. In a max heap, the value of each node is greater than or equal to its parent's value.
A binary heap as two possible representations. It can be viewed as a binary tree as well as an array that is structured as binary tree through the indices. As the binary tree representation allows for a straight forward possibility to perform structural induction, we chose to take a binary heap as underlying data structure for the binary heap.

```
inductive BinaryTree (α :  Type)
| leaf :  BinaryTree α
| node :  BinaryTree α → α → BinaryTree α → BinaryTree α
```

We chose to allow a the binary heap to be of any type α. As we will later see, there are some restrictions on α needed for the implementation of some operations. The values of the nodes are assigned by a function f that is passed along with the binary tree. f assigns to each node a value of type `ENat`. `ENat` allows for the representation of infinity which is needed for the Dijksta algorithm. The binary min heap invariant is then defined as a proposition on the binary tree that holds whenever all nodes have a smaller value assigned to by f than its parents:

```
def is_min_heap :  BinaryTree α → (ordering :  α → ENat) → Prop
| leaf, _ => true
| node l v r, f => match l, r with
| leaf, leaf => true
| node _ lv _, leaf => f v ≤ f lv ∧ isMinHeap l f
| leaf, node _ rv _ => f v ≤ f rv ∧ isMinHeap r f
| node _ lv _, node _ rv _ =>
f v  ≤ f lv ∧ isMinHeap l f ∧  f v ≤  f rv ∧ isMinHeap r f
```

2.1 Operations

For the operations on the binary heap, we show that they function as expected. In particular, we show that the functionality is correct and the binary heap invariant is established (or preserved, respectively).

2.1.1 heapify

Heapify is a subroutine that is used by several of the operations. It takes a binary tree as input for which the heap invariant holds for both of its children if they exist and moves the root node to the correct position such that the heap invariant holds for the whole binary tree. We show for the formalization that it establishes the heap invariant and the output contains the exact same nodes as the input. We formulate the theorem to be over general trees (as opposed to only nodes that have children) to be able to perform induction in the proof. In particular:

Theorem 2.1 (Theorem. Correctness `heapify`.) *Let bt be a binary tree over elements of type α, and let $f : \alpha \to \mathbb{N} \cup \{\infty\}$ be a function. Then:*

$$(\forall v.v \in bt \iff v \in \mathit{heapify}(bt, f)) \wedge \tag{1}$$

$$(\exists l, v, r.bt = \mathit{node}(l, v, r) \Rightarrow \mathit{is\_min\_heap}(l, f) \wedge \mathit{is\_min\_heap}(r, f) \Rightarrow \mathit{is\_min\_heap}(\mathit{heapify}(bt, f), f)) \tag{2}$$

English explanation. To prove the first part, we just show that no node was added or removed during the heapify operation. The second part requires a proof by functional induction over `heapify`. The base cases are trivial as the `leaf` and the node with only `leafs` as children satisfy the heap invariant by definition. The cases that don't call `heapify` recursively can be solved with the definition and the assumptions availiable at the respective proof states. For the other three cases, the structure of the proof is similar: We use the fact that a binary tree satisfies the heap condition if both children are min heaps and the value assigned the root node is less of equal to the one of the children. This allows us to use the induction hypothesis for the two children and we are able to use the conditions for the cases to show that the root indeed holds the minimum value.
Lean correspondence. We show both statements separately. We split the first one into the both directions (`contains_then_heapify_contains, heapify_contains_then_contains`) and can prove both with functional induction over `heapify` and the definition of `contains` (which corresponds to $\in$). We do `fun_induction heapify generalizing l r v; all_goals expose_names`. Note that we need to generalize the induction hypothesis in order to apply it to the children of the tree. For the leaf cases and the ones that do not recurs, we can simply use the definition of heapify together with the assumptions. For the cases that call `heapify`, we define a helper lemma that shows that if the two children are min heaps and the value assigned to the root is less than or equal to that of the children, then the tree is a min heap (`left_and_right_are_min_heap_and_root_is_min_of_children_is_min_heap`). To show that the children are min heaps we can simply use the induction hypothesis (as the children are min heaps, its children are also min heaps, so the induction hypothesis can be applied (by `min_heap_then_left_and_right_are_min_heap`))

and the assumptions. To show that the root value is smaller that the values of the children, we use the fact, that in the `heapify` we switch the root with the smaller child if it is larger and then, the smallest value of each of the children is smaller than the root because the children are min heaps (`root_is_min_of_children`, `min_heap_member_le_root`) and its members did not change except for the potentially switched root whose value is larger (`heapify_contains_then_contains`).

2.1.2 insert

For the `insert` operation, we show that all members are equal with the addition of the inserted node.

Theorem 2.2 (Insert Correctness) *Let* $bt : \mathsf{BinaryTree}\,\alpha$ *and* $f : \alpha \to \mathbb{N} \cup \{\infty\}$. *Then*

$$\big(\forall v\, v',\ v \in bt\ \vee v = v' \iff v \in (\mathsf{insert}\ bt\ v'\ f)\ \big) \wedge \tag{3}$$

$$\big(\mathsf{is\_min\_heap}\ bt\ f\ \implies\ \mathsf{is\_min\_heap}\ (\mathsf{insert}\ bt\ v\ f)\ f\big). \tag{4}$$

English explanation. For the correctness proof, we show two parts. The first one can be further decomposed into the two directions. Showing that the tree with inserted node still contains all nodes can be shown via functional induction over insert assuming there is a node vi n bt. The base case is trivial. If the binary tree is initially a leaf, then the case can be solved by contradiction as a leaf can not contain a value. In the step case, we assume v is contained in the current node and analyze whether it is at the root or in one of the subtrees. If it is in a subtree, we apply the induction hypothesis to that subtree and then reconstruct the proof for the inserted tree, showing that insertion preserves membership of existing elements. To show that the inserted node is in the new tree, we can again perform a simple function induction. If the tree was a leaf, the value is added in the root. In the other case it either is inserted at the root which concludes the proof or it is propagated to the children. In that case, we can use the insuction hypothesis. For the other direction, we can again perform functional induction over insert. The leaf case is trivial, in that case $v = v'$. In the two step cases, we make a case distinction over the membership of v. It is either at the root in which case we have $v = v'$ or it is a member of the children. In that case, we can use the induction hypothesis.
In the second part of the proof we show that insertion maintains the min-heap invariant, assuming the original tree is a min-heap. We use functional induction over insert. The base case is trivial as any node with only leafs as children is a min heap by definition. For the two step cases, we again make use of the fact, that if the children are min heaps and their roots are smaller than the root of bt, then bt is a min heap. As the condition in the definition ensures that the value is inserted at the root if it is smaller than the current node, we can show that the root remains the minimum. And as the node is propagated to the children if it is larger, we can make use of the induction hypothesis.
Lean correspondence.
We use a similar structure as described above. We show the first part with the lemmas `insert_contains_inserted` and `contains_then_insert_contains` for one direction and `insert_contains_then_contains_or_inserted` for the other one, respectively. All are shown with functional induction over `insert`.
The preservation of the heap invariant is shown in `insert_preserves_min_heap`. It makes use of `left_and_right_are_min_heap_and_root_is_min_of_children_is_min_heap` and `min_heap_then_left_and_right_are_min_heap` which shows that the children of a min heap are min heaps if they exist.

2.1.3 extract_min

For the `extract_min` operation, we show that removing the minimum element preserves all remaining elements and maintains the min-heap invariant. Additionally, we show that the extracted element corresponds to the minimum priority stored in the heap. The `extract_min` operation returns a pair: the left part is the minimal element, and the right part is the tree with the element removed.

Theorem 2.3 (Extract-Min Correctness) *Let* $bt, l, r : \mathsf{BinaryTree}\,\alpha$, $v, v' : \alpha$, *and* $f : \alpha \to \mathbb{N} \cup \{\infty\}$. *Then*

$$\big(\mathsf{contains}\ (\mathsf{extract\_min}\ bt\ f).2\ v\ \implies\ \mathsf{contains}\ bt\ v\big) \wedge \tag{5}$$

$$\big(bt = \mathsf{node}\ l\ v\ r\ \implies\ \mathsf{contains}\ bt\ v'\ \implies\ v \neq v'\ \implies\ \mathsf{contains}\ (\mathsf{extract\_min}\ bt\ f).2\ v'\big) \wedge \tag{6}$$

$$\big(bt = \mathsf{node}\ l\ v\ r\ \implies\ \mathsf{is\_min\_heap}\ bt\ f\ \implies\ \exists bt'\ v',\ \mathsf{extract\_min}\ bt\ f = (\mathsf{some}\ v', bt')\big) \tag{7}$$

$$\wedge\ \mathsf{is\_min\_heap}\ bt'\ f\ \wedge\ f\,v = \mathsf{heap\_min}\ bt\ f\big). \tag{8}$$

English explanation. The correctness proof of `extract_min` consists of several interconnected parts.

First, we show that every element contained in the resulting tree after extraction was already present in the original tree. This establishes that `extract_min` does not introduce any new elements. The proof proceeds by structural reasoning over the definition of `extract_min`. Since the operation only removes the root element and restructures the remaining tree, all elements in the resulting tree must have originated from the original heap.
Second, assuming that the original tree satisfies the min heap invariant, we show that `extract_min` returns a value v' together with a tree bt' and the resulting tree bt' is again a min heap and the extracted value v' corresponds to the minimum priority stored in the original heap. This follows from the definition of `extract_min`, which removes the root of the heap and then restores the heap property by rearranging the remaining nodes. The induction hypothesis ensures that all recursive calls preserve the heap invariant.
Finally, we show that extraction preserves the membership of all elements except for the removed root. If the original tree is a node with the root value v, the
Lean correspondence.
In Lean, we followed a similar structure as described above: We defined the auxiliary lemmas `extract_min_correct_node`, `extract_min_contains_then_contains`, and `contains_then_extract_min_contains_except_root`, each of which proving one piece of the theorem. To implement the `extract_min` function, we defined a `get_last` function that finds the last element. `extract_min` then puts it at the root position instead of the minimum element which is taken out of the tree and returned. `extract_min` applies the `heapify` function to restore the heap invariant. Because of this setup, we first showed membership lemmas for `get_last` and then used them in the proofs for `extract_min`. For the proof of the heap invariant, we showed that if bt is a min heap, then, for the result of `get_last` holds that the left aud right sub trees are min heaps. This is the precondition to show that `heapify` restores the heap invariant.

2.1.4 decrease_priority

For the `decrease_priority` operation, we show that all existing members of the binary tree are preserved and that the min-heap invariant is maintained when the priority of a node is decreased.

Theorem 2.4 (Decrease Priority Correctness) *Let* $bt : \mathsf{BinaryTree}\,\alpha$, $v, v' : \alpha$, *and* $f : \alpha \to \mathbb{N} \cup \{\infty\}$, *assuming decidable equality on* α. *Then*

$$\mathsf{contains}\ bt\ v \iff \mathsf{contains}\ (\mathsf{decrease\_priority}\ bt\ v'\ f)\ v \wedge \tag{9}$$

$$\big(\mathsf{is\_min\_heap}\ bt\ f \implies \mathsf{is\_min\_heap}\ (\mathsf{decrease\_priority}\ bt\ v\ f)\ f\big). \tag{10}$$

English explanation. The correctness proof for `decrease_priority` consists of two parts.
First, we show that decreasing the priority of a node does not change the set of elements contained in the tree. This is split up in two parts and by shown by induction on the definition of `decrease_priority` which is a combination of first removing and then inserting the node in the tree. We show that executing them in sequence leads to the same members in the tree.
Second, we show that the min heap invariant is preserved when decreasing a key. As `decrease_priority` makes use of heapify, this is shown by the proof of correctness of heapify.
Lean correspondence. The proof relies on the auxiliary lemmas `contains_then_decrease_priority_contains`, `decrease_priority_contains_then_contains`, and `decrease_priority_preserves_min_heap` following the same procedure as described above. As mentioned, `decrease_priority` is built on top of `insert`, `remove` and `merge`, so we showed the corresponding correctness lemmas for `remove` and `merge` (`as correctness of insert is already shown above`) using functional induction and could combine the lemmas with the auxiliary lemmas from `insert` to prove the final goals.

3 Dijkstra's algorithm

Goal. Our goal was to formalize Dijkstra's algorithm over finite (connected) graphs and prove correctness, using a priority queue abstraction.

Solution. We represent the graph as a finite simple graph so that vertices have decidable equality (exactly as we have seen 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` V). The heap that was implemented in the first part of the project exposes abstract operations (empty check, extract-min, decrease-priority, etc) and the proof for the algorithm only relíes on their contracts.

3.1 Implemenentation

The implementation of the algorithm is split up into three parts:

- Relaxation (Lean: `relax_neighbors`) 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 updates the queue (implemented as BinaryHeap) accoridngly.
- 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.

3.2 Termination

We split the implementation into three parts: `dijkstra` (a non-recursive wrapper), `dijkstra_rec` (the recursive core), and `relax_neighbors` (implemented as a fold over the finite list of neighbors). Since we only need to prove termination for recursive definitions, we only have to show termination explicitly for `dijkstra_rec`.
We prove termination by the measure `queue.sizeOf`: We prove that the heap measure strictly decreases, i.e. $queue''.sizeOf < queue.sizeOf$. Starting from a nonempty heap (the hypothesis hq that $queue \neq queue.empty$ is used), the proof exposes the concrete definitions of the intermediate queues with hq'_eq and hq''_eq (both proven via reflexitivity because queue' and queue" are exactly the second components of extract_min and relax_neighbors respectively). The `calc` chain then rewrites $queue''.sizeOf$ to $(relax\_neighbors\ g\ u\ dist\ queue').2.sizeOf$, applies the lemma `sizeOf_relax_neighbors_le` to get a non-strict inequality to $queue'.sizeOf$, rewrites $queue'.sizeOf$ to the second component of extract_min, and finally uses an exposed lemma from the BinaryHeap interface to get the strict decrease after extraction; together these steps yield the required strict decrease.

Besides the exposed lemma from the BinaryHeap interface, we also relied in this proof on the lemma `sizeOf_relax_neighbors_le`. This lemma states that if we relax the neighbors using `relax_neighbors` that the queue can only ever decrease in size. In Lean that is expressed as:

```
BinaryHeap.sizeOf (Prod.snd (relax_neighbors g u dist q)) ≤ BinaryHeap.sizeOf q
```

This lemma is important as it is used in total 4 times throughout our proofs. We can prove it using induction on the the neighbors of a node generalizing the distance map and priority queue. If the neighbors list is empty, the statement is obviously true (in Lean a simple `simp` is enough). If the neighbors are a list, however, we name dist' and queue' the result of applying the function f (the function applied to all entries in the fold) and show by case distinction that `queue'.sizeOf` $\leq$ `q.sizeOf`. By applying the induction hypothesis on the tail we know that the size of BinaryHeap after the fold over the tail is $\leq$ `queue'.sizeOf`. By chaining both equalities we can then finish the proof.

3.3 Correctness proof (`dijkstra_correctness`)

Our goal is to proof that the algorithm is correct, i.e., for any vertex v, the algorithm's output equals the graph distance δ defined by Mathlib:

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

The proof proceeds by fixing the algorithm's output (concretely, set `dist` := `dijkstra` $g\ s\ v$) and deriving a contradiction from any alleged disagreement with the true graph distance.

We assume, for contradiction, that there exists a vertex u with `dist` $u \neq \delta(s, u)$. Choose such a u that minimizes $\delta(s, u)$, given by `minimal_counterexample`. Note also that by `dijkstra_source_zero` the algorithm assigns 0 to the source, i.e. `dist` $s = 0 = \delta(s, s)$, so any counterexample cannot be the source. By the previous remark we have $\delta(s, u) > 0$, proven by the lemma `positive_distance_of_counterexample`. Take a shortest path from s to u; this yields a predecessor y with $y \sim u$ and

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

given by the lemma `exists_pred_on_shortest_path`.
Since $\delta(s, y) < \delta(s, u)$, we have

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

because u was minimal.
Now apply the algorithmic per-edge final bound `relax_adj_final_bound` to the edge $y \sim u$. That lemma yields the upper bound

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

Substituting $\texttt{dist}\ y = \delta(s, y)$ and $\delta(s, y) + 1 = \delta(s, u)$ gives $\texttt{dist}\ u \leq \delta(s, u)$.
We establish the global lower bound

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

provided by the lemma `never_underestimates`.

The global and lower bound together force the equality $\texttt{dist}\ u = \delta(s, u)$, contradicting the choice of u as a counterexample. Hence no counterexample exists, and we conclude that `dijkstra_correctness` is correct.

Preconditions and placeholders. The development assumes the standard BinaryHeap contracts and constructs the correctness proof from a collection of named lemmas, many of which themselves rely on smaller auxiliary results. Figure 1 illustrates the dependency structure among the main lemmas used in the correctness proof of Dijkstra's algorithm; for reasons of space and clarity, some minor lemmas are omitted from the diagram.
All lemmas shown in the diagram have been fully formalized and proven, with only few exceptions. The lemma
`exists_extract_or_top` has not yet been completed. The lower-bound lemma `never_underestimates` is not proven as it was assumed given / obvious in the original proof. In addition two small helpers that argue over the BinaryHeap are left as sorry (`extract_min_still_correct_1` and `extract_min_still_correct_2`), because that was not the focus of our work. The reasons these proofs remain unfinished are discussed in more detail in the corresponding sections on the following pages.

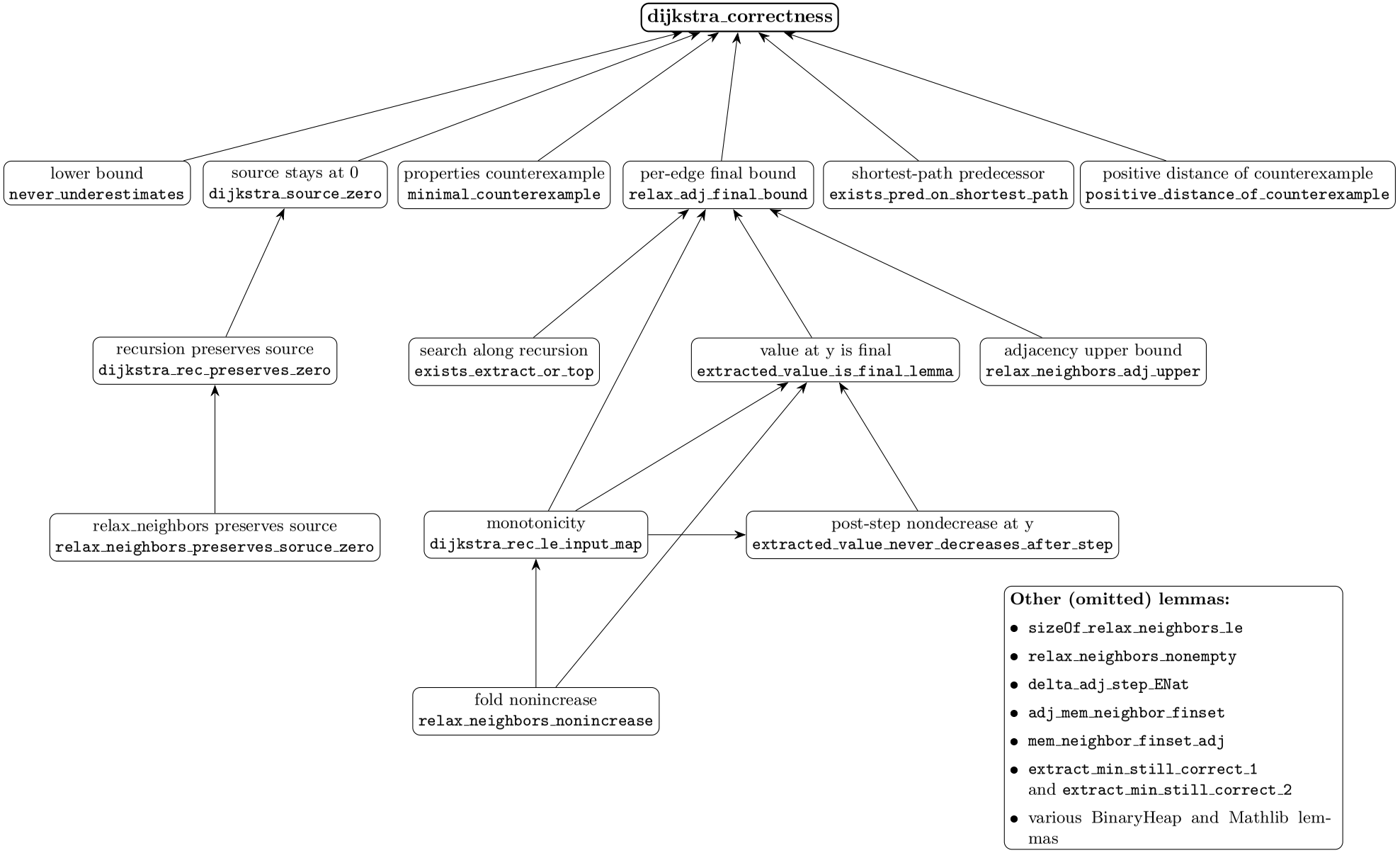

Figure 1: Lemma dependency graph shaping the proof of correctness.

3.3.1 Lower bound (`never_underestimates`)

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, this lemma is currently left as `admit`. All online references we found assume this property as a given, since it is so intuitive. We therefore treat this lemma as an axiomatic starting point for our correctness proof.

3.3.2 Source preservation (`dijkstra_source_zero`, `dijkstra_rec_preserves_zero`, `relax_neighbors_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 $\texttt{dist}(s) = 0$. We must show that this invariant is preserved through all recursive steps. The key observation is that `relax_neighbors` 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 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.

- `relax_neighbors_preserves_source_zero` proves preservation for a single relaxation step. For that we first show that applying the update on a single neighbor will preserve the invariant (`step_preserve`): If it is the source node, then relaxation is not possible, because no number plus 1 can be smaller than 0 in `ENat`. Otherwise, the result is immediate after a case distinction on if the neighbors (that are not the source) are relaxed or not. With induction on the neighbors we can show this for the entire fold (`fold_preserve`). We can then easily close this goal with rewrites and `simpa`.
- 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 combine the previous lemma to get `dist' source` $= 0$ after relaxation, then apply the induction hypothesis to the smaller post-step heap and can finish the goal.
- Finally, `dijkstra_source_zero` invokes the recursive lemma on the initial state.

3.3.3 Minimal counterexample (`minimal_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 $\texttt{dist w} = \delta(s, w)$.

English explanation. We can show by assumption 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 (this does not have to be unique). By definition of minimality, any vertex w with strictly smaller distance must not be a counterexample, hence $\texttt{dist}(w) = \delta(s, w)$.

Lean correspondence. `minimal_counterexample` 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.

3.3.4 A minimal counterexample has a positive graph distance (`positive_distance_of_counterexample`)

Statement. This is a simple lemma that state that if we have two distinct nodes, that their true graph distance is greater than zero.

English explanation. We prove this simply by contradiction. Since we have no negative numbers, assume for contradiction that $\delta(s, u) = 0$. We then use connectedness to pick a vertex from which both s and u are reachable; from those two reachability facts you get a path from s to u (so s and u are reachable). If there is a path from s to u and the shortest-path distance between them is 0, that forces s and u to be the same vertex (distance 0 only occurs for identical vertices). However, we assumed $u \neq s$, a contradiction.

Lean correspondence. The 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 $\delta(s, u) > 0$.

3.3.5 Shortest-path predecessor (`exists_pred_on_shortest_path`)

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`. We first invoke the library lemma `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 `tail.length` $+ 1 = \delta(s, u)$. Since `tail.length` $+ 1 = \delta(s, u)$ and $\delta(s, y) \leq$ `tail.length`, we have $\delta(s, y) + 1 \leq \delta(s, u)$. In contrast, we have $\delta(s, u) \leq \delta(s, y) + 1$ by `delta_adj_step_ENat`. Both inequalities together show that $\delta(s, u) = \delta(s, y) + 1$, which finishes the proof.

3.3.6 Monotonicity (`dijkstra_rec_le_input_map`, `relax_neighbors_nonincrease`)

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 (BinaryHeap lemma) and does not increase during relaxation (lemma `sizeOf_relax_neighbors_le` from Section 3.2), so `sizeOf(queue)''` $<$ `sizeOf(queue)`. This allows us now to use the induction hypothesis. By applying it to the smaller heap, the final result satisfies

$$(\texttt{dijkstra\_rec g s t dist' queue''})\ x \leq \texttt{dist}'\ x$$

The lemma `relax_neighbors_nonincrease` 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_isEmpty_eq_false` for the strict decrease and `sizeOf_relax_neighbors_le` for the nonincrease. The induction hypothesis then applies to the strictly smaller post-step heap size. Finally, `relax_neighbors_nonincrease` provides the one-step inequality $\texttt{dist}'\ x \leq \texttt{dist}\ x$, and we compose with `le_trans`.

This proof used the lemma `relax_neighbors_nonincrease`, which of course needs to be shown as well.

Statement. For all vertices x,

$$(\texttt{relax\_neighbors}\ g\ u\ \texttt{dist}\ q).1\ x \le \texttt{dist}\ x$$

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 $\le$ the input map.
Lean correspondence. Similar to the proof for `relax_neighbors_preserves_source_zero` we first show that the inequality is preserved if we apply the function f (the function of the fold that updates the distance map) to a state of the algorithm, which we call `step` in our proof. This can be shown by a simple case distinction on the if-condition `(d u + 1) < d v` and simplifications. Now we can lift this result to the entire fold by induction on the neighbors list to prove `fold`. The base case (empty list) is trivial. In the inductive step (the neighbors list is a cons), we can use our result, that one application of f preserves the inequality on the head of the list. We then apply the induction hypothesis to the tail of the list and compose the inequalities with a few rewrites and `simp` statements. Using further simplifications and our intermediate statement `fold` we can then conclude the statement.

3.3.7 Relaxation only every decreases distances (`relax_neighbors_adj_upper`)

Statement. When relaxing all neigbors of a vertex y, the distance of any neighbor u cannot end up larger than the candidate distance obtained by y, $\texttt{dist y} + 1$. In other words, if $y \sim u$, then

$$(\texttt{relax\_neighbors}\ g\ y\ \texttt{dist}\ q).1\ u \le \texttt{dist y} + 1$$

English explanation. Several structural facts about the neighbor list have to be established first.

- First, because u is adjacent to y, it appears in the finset of neighbors of y, and therefore also in the list obtained from that finset.
- We show that `(Prod.fst (f (d, pq) u)) u ≤ d y + 1`. We analyze the effect of a single relaxation step (in Lean `step_u_bound`): When the relaxation function f is applied to u itself, the resulting distance at u is either updated to dist y + 1 or left unchanged because it was already smaller. If it is unchanged, the distances do not change and the result is immediate, because the inequality holds by the if-condition. If we change the distances, we establish the invariant by setting the distance to `d y + 1`. Thus, the inequality also holds.
- We now prove a similar fact `(Prod.fst (f (d, pq) v)) u ≤ d y + 1` for all $v \neq u$ using the same proof structure. We make a case distinction on $d\,y+1 < d\,v$ and use $v \neq u$ to establish the inequality. In Lean we will call this `step_other_preserve`.
- Now we want to list this result to the entire fold, given that $u \notin l$, called `preserve_no_u` in Lean, i.e., we show that

  $$u \notin l \to \texttt{(Prod.fst}\ (d, pq)) \ u \le dy + 1 \to \texttt{(Prod.fst}\ (\texttt{List.foldl}\ f\ (d, pq)\ l)) \ u \le dy + 1$$

  The proof proceeds by induction on the list l. In the base case, where l is empty, the fold does nothing and the result follows immediately. In the inductive case, we consider the head v and the tail l'. Since $u \notin v :: l'$, we have $v \neq u$ and $u \notin l'$. Applying the inductive hypothesis requires showing that the bound is preserved after relaxing v, which follows directly from `step_other_preserve`. This establishes that folding over the entire list preserves the bound when u does not occur in the list.
- Because $u \in l$ could also be true we also need to lift the previous results to the fold for this case, which we call `bound_if_mem_nodup`, i.e., we show that

  $$\texttt{l.Nodup} \to (\forall v, v \in l \to v \neq y) \to u \in l \to (\texttt{Prod.fst}\ (\texttt{List.foldl}\ f\ (d, pq)\ l)) \ u \le dy + 1$$

  The proof again proceeds by induction on the list l. In the base case, the statement is vacuous since $u \in [\,]$ is impossible. In the inductive case, we distinguish whether u is equal to the head of the list or occurs in the tail. If u is the head, then relaxing u establishes the bound immediately by `step_u_bound`. The no-duplicate assumption implies that u does not appear in the tail, so `preserve_no_u` can be applied to show that folding over the remainder of the list preserves the bound. If u occurs in the tail, then the head is different from u, and relaxing it preserves the bound by `step_other_preserve`. The induction hypothesis then applies to the tail, yielding the desired result.

With these results in place, we can conclude the proof by instantiating the above lemma with the concrete neighbor list of y. Since this list is duplicate-free, contains only vertices distinct from y, and contains u by adjacency, all assumptions of `bound_if_mem_nodup` are satisfied. Rewriting the resulting statement using the definition of `relax_neighbors` then yields the desired inequality which completes the proof.

Lean correspondence. The proof proceeds by unfolding `relax_neighbors` into a `List.foldl` over the list of neighbors of y, introducing a local relaxation function f. Membership facts are established first: adjacency gives u ∈ `g.neighborFinset` y, which is transferred to the list representation using `Finset.mem_toList`, and `nodup_toList` yields the Nodup hypothesis needed later. The first part, `step_u_bound`, analyzes `(f (d, pq) u)` by a `by_cases` split on $dy + 1 < du$; in the true branch, `simp` shows that the updated distance is exactly $dy + 1$, and in the false branch the inequality follows from `le_of_not_gt`. The second part, `step_other_preserve`, handles the case $v \neq u$ and shows that relaxing any other vertex preserves a previously established bound at u; here the proof again splits again on $dy+1 < dv$, with simp and equality cases discharging the goals.
These stepwise results are then lifted to folds by induction on lists. The third part `preserve_no_u` proves that folding over a list not containing u preserves the bound, using induction on the list and `step_other_preserve` at the head, together with `List.not_mem_of_not_mem_cons` to maintain the side condition.
The main part four `bound_if_mem_nodup` handles the complementary case $u \in l$ under Nodup and the condition that no element of l equals y. The inductive step splits on whether u is the head of the list or occurs in the tail using `List.mem_cons`; in the head case, `step_u_bound` establishes the bound immediately and `preserve_no_u` propagates it over the tail using Nodup to show $u \notin vs$, while in the tail case `step_other_preserve` preserves the bound and the induction hypothesis applies.
Throughout, equalities such as $d'y = dy$ are shown by unfolding f and ruling out impossible equalities using the $v \neq y$ hypothesis.
Finally, the result is instantiated with the concrete neighbor list, and a concluding `simpa [relax_neighbors, neighbors, f]` rewrites the folded statement back into the original form.

3.3.8 Extraction stability (`extracted_value_never_decreases_after_step`, `extracted_value_is_final_lemma`)

Statement (`extracted_value_is_final_lemma`). 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. Formally, given the invariant `min_y_invariant (V := V) y next hhNext` → it holds that:

```
dist y = (dijkstra_rec g s t next.1 next.2) y
```

English explanation & Lean correspondence. The proof begins by fixing the result of the extraction step. Writing q' for the queue obtained by removing the minimum element and next for the state after relaxing the neighbors of y, the first part of the argument establishes an upper bound on the final distance. Using a general monotonicity lemma for the Dijkstra recursion, it is shown that the value returned by `dijkstra_rec` at y is bounded above by the distance stored in the current map `next.1`. A separate lemma about `relax_neighbors` then shows that relaxing neighbors cannot increase the distance at y, so `next.1` $y \leq$ `dist` y. Chaining these two facts yields the inequality

$$(\texttt{dijkstra\_rec g s t next.1 next.2})\ y \leq \texttt{dist}\ y \tag{11}$$

Now we prove `preserve_y`, which states that

$$(\forall v, v \in l \to v \neq y) \to (\texttt{Prod.fst}\ (\texttt{List.foldl}\ f\ (d, pq)\ l))\ y = d\ y$$

by induction on the neighbor list: assuming that no element of the list is equal to y, folding f over the list leaves the value at y unchanged. The proof analyzes a single fold step by unfolding f and performing a case split on whether a relaxation occurs; the hypothesis that each neighbor is distinct from y rules out the only case in which the update could affect the value at y. Applying this lemma to the concrete neighbor list shows that next.1 y = dist y.
Then, the lower bound is obtained from a previously established lemma `extracted_value_never_decreases_after_step` stating that, under the preserved Dijkstra invariant, extracted values never decrease in subsequent recursive calls. Instantiating this lemma for the current step yields

$$\texttt{min\_y\_invariant y next hhNext} \to \texttt{next.1 y} \leq \texttt{dijkstra\_rec g s t next.1 next.2 y} \tag{12}$$

where the invariant is given to us already. Finally, since `next.1 y = dist y` by `preserve_v` we can combine 11 and 12 to achieve equality.

Statement (`extracted_value_never_decreases_after_step`). After extracting a vertex y from the priority queue and relaxing its neighbors, the distance value assigned to y never decreases in any subsequent recursive call of Dijkstra's algorithm. Formally, assuming that the invariant

$$\texttt{min\_y\_invariant (V := V) y next hhNext}$$

holds after the relaxation step, we have

$$\texttt{next.1}\ y \le (\texttt{dijkstra\_rec g s t next.1 next.2})\, y.$$

English explanation & Lean correspondence. The proof is organized around a strong induction on the size of the priority queue and is implemented in the auxiliary lemma `nondec`. This lemma states that for any state $p = (d, q)$ whose queue has size n, if the invariant `min_y_invariant y p` holds and the queue is nonempty, then the current distance $d\,y$ is bounded above by the value produced by `dijkstra_rec g s t d q` at y. The base case corresponds to an empty queue, where `dijkstra_rec` unfolds definitionally and the desired inequality holds trivially. In the inductive case, the queue is nonempty, so `dijkstra_rec` unfolds to an extraction step followed by a recursive call.
Let u_1 be the vertex extracted from the queue, q_1 the resulting queue, and let $next2$ denote the state obtained by relaxing the neighbors of u_1. The proof splits the goal $d\,y \le (\texttt{dijkstra\_rec g s t next2.1 next2.2})\, y$ into two components. First, a monotonicity lemma `dijkstra_rec_le_input_map` shows that the recursive result is bounded above by the current distance map, that is,

$$(\texttt{dijkstra\_rec g s t next2.1 next2.2})\, y \le \texttt{next2.1}\, y.$$

Second, it is shown that the relaxation step does not decrease the distance value at y; this fact is established by the lemma `hpreserve_y`, which proves

$$\texttt{p.1}\, y \le \texttt{next2.1}\, y.$$

The proof of `hpreserve_y` unfolds `relax_neighbors` as a fold over the neighbor list of u_1 with a local relaxation function f_1. The key local property is captured by the lemma `step_preserve_y`, which shows that applying f_1 to any vertex $v \neq u_1$ preserves the value at y, assuming the bound $\texttt{p.1}\, y \le d\, u_1 + 1$ and the equality $d\, y = \texttt{p.1}\, y$. This lemma is proved by unfolding f_1 and performing a case split on whether the relaxation condition $d\, u_1 + 1 < d\, v$ holds; the assumption $v \neq u_1$ rules out the only case in which the update could affect the value at y.
The stepwise preservation result is lifted to the entire neighbor list by the inductive lemma `preserve_eq_list`. This lemma shows that folding f_1 over a list of vertices none of which equals u_1 leaves the value at y unchanged. This is proven by a simple induction over the neighbors list using `step_preserve_y` at the head of the list.
Applying `preserve_eq_list` to the concrete neighbor list yields the equality

$$\texttt{(List.foldl f1 (p.1, q1) neighbors1).1}y = \texttt{p.1}y$$

which is then used to conclude `hpreserve_y`, since `List.foldl f1 (p.1, q1) neighbors1 = next2`.
Finally, the induction hypothesis is applied to the strictly smaller queue `next2.2`; the strict decrease in queue size is obtained using a BinaryHeap lemma together with `sizeOf_relax_neighbors_le`, and the invariant for the next state is supplied by the hypothesis `hInvPreserve`. This completes the proof.

3.3.9 Search lemma (`exists_extract_or_top`)

Statement. The lemma `exists_extract_or_top` establishes a dichotomy for the value of Dijkstra's algorithm at a fixed vertex y. Either the algorithm computes $(\texttt{dijkstra g s t})\, y = \top$, meaning that y is unreachable from the source, or there exists a concrete execution state $(\texttt{dist}, q)$ in which y is extracted as the minimum element of the priority queue, and the overall Dijkstra computation coincides with a single unfolding step followed by the recursive call starting from the relaxed successor state. This lemma thus formalizes the intuitive fact that if the final distance to y is finite, then at some point during the execution y must be removed from the queue and processed.

English explanation. The proof proceeds by classical case distinction on whether $(\texttt{dijkstra g s t})\, y = \top$. In the trivial case, the left disjunct holds immediately. In the nontrivial case, the algorithm is unfolded from its initial state $(\texttt{dist}_0, q_0)$, where $\texttt{dist}_0$ assigns distance 0 to the source and $\top$ elsewhere, and q_0 contains all vertices. The key idea is to perform a bounded search over the recursion depth, measured by the size of the priority queue. A strong induction on this size shows that for any state (d, q), either the queue is empty and $d\, y = \top$, or the queue is nonempty and `dijkstra_rec` unfolds to a next state `next`

obtained by extracting the minimum and relaxing its neighbors. Since the assumption excludes the case where the final value at y is $\top$, the empty-queue branch is impossible at the initial state. Consequently, one obtains a nonempty queue and a successor state witnessing the right-hand disjunct of the statement, in which y is extracted and the global Dijkstra computation agrees with the recursive call from that successor state.

Lean correspondence and implementation issues. In Lean, this strategy is reflected by the auxiliary predicate `search`, which performs a strong induction on a natural number bounding `BinaryHeap.sizeOf q`. The intended invariant is that `dijkstra_rec g s t d q` either terminates immediately because the queue is empty, or unfolds definitionally to a recursive call on a strictly smaller queue. However, formalizing this argument runs into difficulties at two critical points. First, in the empty-queue branch, the proof obligation requires showing $d\,y = \top$, but this fact is not derivable from the available hypotheses alone: it would require a global invariant relating empty queues to unreachable vertices, which is not encoded in the current assumptions. This gap appears explicitly in the first `sorry` inside the `hq : q.isEmpty` branch of `search`.
Second, in the nonempty-queue branch, although `dijkstra_rec` definitionally unfolds to an extraction followed by a recursive call, Lean requires a precise equality between the original recursive call and the one on the successor state `next`. Establishing this equality demands careful rewriting with the definitional equation of `dijkstra_rec`, along with explicit proofs that the size of the queue strictly decreases and that all required invariants are preserved. While these facts are conceptually ensured by `BinaryHeap.sizeOf_extract_min_lt`, `sizeOf_relax_neighbors_le`, and the hypotheses `hInvPreserve` and `hInvInit`, orchestrating them into a single equality proof exceeds what can be obtained by rewriting alone. This mismatch manifests in the second `sorry`, where Lean cannot be convinced that the unfolded recursive call is definitionally equal to the original `dijkstra_rec`. Thus, this lemma remains the main open challenge in the development.

3.3.10 Final edge bound (`relax_adj_final_bound`)

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

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

English explanation. The argument proceeds by unfolding the definition of `dijkstra` and performing a case distinction based on the execution behavior at vertex y. Using the auxiliary lemma `exists_extract_or_top`, we either conclude that the final distance at y is $\top$, in which case the inequality is trivial, or we obtain a concrete execution step in which y is extracted from the queue and its neighbors are relaxed.
In the nontrivial case, the algorithm reaches a state $(\texttt{dist}, q)$ where y is extracted and a successor state `next` is produced by `relax_neighbors`. The proof then chains three facts. First, by monotonicity of `dijkstra_rec`, the final distance is bounded above by the current distance map, so the final value at u is at most `next.1` u: For all vertices x

$$(\texttt{dijkstra\_rec g s t dist q})\ x \le \texttt{next.1}\ x$$

Second, by the adjacency hypothesis and the specification of `relax_neighbors`, relaxing neighbors of y enforces the local bound

$$\texttt{next.1}\ u \le \texttt{dist}\ y + 1$$

Third, one shows that the distance at y is stable, i.e.,

$$\texttt{dist}\, y = (\texttt{dijkstra\_rec g s t next.1 next.2})\, y.$$

This is established by a case distinction on whether the queue in the successor state is empty.

- In the first case, if the queue is empty immediately after relaxing y's neighbors, no further recursive steps occur in `dijkstra_rec`. The algorithm has reached a terminal state, and the only updates that could affect y come from relaxing its neighbors. Here, the lemma `preserve_y` applies: it states that if we fold the relaxation function f over a list of neighbors that does not include y, the value at y remains unchanged. This is proven by induction on the neighbor list. For each neighbor v, the fold either performs a relaxation or leaves the distance map unchanged. The key observation is that $v \neq y$ for all neighbors, which ensures that no relaxation step can overwrite the distance at y. Inductively, this shows that after the entire neighbor list is processed, y's value is preserved, yielding `next.1 y` $=$ `dist` y, and hence the recursive call also returns the same value at y.

- In the second case, if the queue is nonempty, further recursive calls of `dijkstra_rec` occur. Here, the maintained invariant `min_y_invariant` together with `extracted_value_is_final_lemma` guarantees that once y has been extracted from the queue, its distance value never decreases in any subsequent recursive call. Thus, `next.1 y` is already the final Dijkstra distance, and we have

$$(\texttt{dijkstra\_rec g s t next.1 next.2})\, y = \texttt{next.1 y} = \texttt{dist}\, y.$$

In both cases, the value at y is stable, which allows us to relate the relaxed neighbor bound at u to the final Dijkstra distance at y
Now we can conclude the proof by a careful application of transitivity of the $\leq$ relation. By chaining these three inequalities, we obtain

$$(\texttt{dijkstra g s t})\, u \leq \texttt{next.1 u} \leq \texttt{dist}\, y + 1 = (\texttt{dijkstra g s t})\, y + 1,$$

which is exactly the statement of the lemma. This explicit chain of bounds, first through the immediate post-relaxation distances and then via the stability of y's distance, allows us to conclude that the final distance at u is indeed at most one more than the final distance at y, completing the proof.

Lean correspondence. In Lean, the proof mirrors this structure closely. After unfolding `dijkstra` and fixing the initial state $(\texttt{dist}_0, \texttt{queue}_0)$, the lemma `exists_extract_or_top` is invoked to split the proof into two cases. In the $\top$-case, rewriting `dijkstra_rec` immediately yields the desired inequality using the fact that $\top + 1 = \top$. In the extraction case, Lean provides witnesses `dist`, q, and `next`, together with an equality `hfinEq` identifying the global Dijkstra result with the recursive call from `next`. This equality is used pointwise via `congrFun` to transport inequalities between the two executions.
The core reasoning is encoded by three auxiliary facts. The lemma `dijkstra_rec_le_input_map` supplies monotonicity (`hmono`), bounding the final result by the current distance map. The lemma `relax_neighbors_adj_upper` provides the adjacency bound on `next.1 u`. Finally, stability of the value at y is obtained by a case split on `next.2.isEmpty`: in the empty case, a local fold argument shows that relaxing neighbors preserves the value at y; in the nonempty case, the preserved invariant allows direct application of `extracted_value_is_final_lemma`. Lean then combines these inequalities using transitivity and rewrites via `hfinEq` to conclude the final bound.

3.4 Summary

This section provides an overview of the formalization work on Dijkstra's algorithm and outlines the remaining proof obligations.

Completed work. The formal development captures the core algorithmic structure in Lean, including the recursive driver `dijkstra_rec`, the neighbor relaxation fold `relax_neighbors`, and a suite of auxiliary lemmas that encode key operational properties. Among the results established are:

- the pointwise nonincrease of the relaxation fold (`relax_neighbors_nonincrease`)
- the local adjacency bound for a single relaxation step (`relax_neighbors_adj_upper`)
- the monotonicity of the recursion (`dijkstra_rec_le_input_map`)
- stability properties for extracted vertices (`extracted_value_never_decreases_after_step` and `extracted_value_is_final_lemma`)
- the per-edge final bound (`relax_adj_final_bound`)
- the minimal-counterexample (`minimal_counterexample`)
- the predecessor machinery (`exists_pred_on_shortest_path`)
- source-preservation lemmas (`relax_neighbors_preserves_source_zero`, `dijkstra_rec_preserves_source_zero` and `dijkstra_source_zero`)
- positive-distance fact for a counterexample (`positiveDistance_of_counterexample`)
- the main correctness statement (`dijkstra_correctness`)

Outstanding obligations (marked as `sorry`). A few components were intentionally left unfinished or require additional engineering effort. The global lower-bound lemma `never_underestimates`, which asserts that Dijkstra's output never underestimates the true shortest-path distances, was not formalized here; most proofs rely on it as an intuitive argument, and a formalization would require a distinct reasoning approach. The main remaining proof is the search dichotomy `exists_extract_or_top`: although

a strong-induction proof on heap size was attempted, aligning the definitional equalities and induction hypotheses proved challenging, so this lemma currently remains a `sorry`. Finally, two auxiliary heap lemmas (`extracted_min_correct_1` and `extracted_min_correct_2`) were introduced to bridge the binary-heap implementation with the abstract algorithm; these assert that specific heap operations behave as expected at certain points and are considered minor supporting facts rather than core contributions, so they were left as `sorry` as well.

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:

Deciding how to represent the binary heap was a big challenge initially. We elaborate on this in section 4.3.
Another challenge we encountered was coordinating the two parts of the project. Both parts took longer than expected, so while we initially thought that we could work on the parts sequentially, we ended up parallelizing the work. We solved this by assuming operations and lemmas, thereby building an interface to the binary heap that we later merged by using the proven lemmas to implement it.

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 trial and error.
In total, the amount of time we needed to spend on the project was much 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

One of the main challenges was the decision on how to represent the binary heap. The two options were an array based and a tree based representation. While the array based representation gives easier access to specific indices and to the parent nodes, the tree structure has an inherent structural induction possibility. We decided to use the tree based representation. While working on the project, we would reconsider this choice as an array based implementation would have allowed for a simpler adaptation of the algorithms for the operations of the binary heap. Moreover, for the potential future goal to do runtime analysis on the operation, an array based implementation would give way for more straight-forward reasoning.

Make distances and priority queue a fixed pair.

4.4 What we have learned