Project Report
Binary Heap and Dijkstra's Algorithm

Josefine Lindmar Sonja Joost

January 15, 2026

1 Introduction

In this project we formalized the binary heap data structure and proved correctness of its operations as well as Dijkstra's shortest path algorithm. (noch mehr vllt TODO)

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*

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

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

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

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

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.

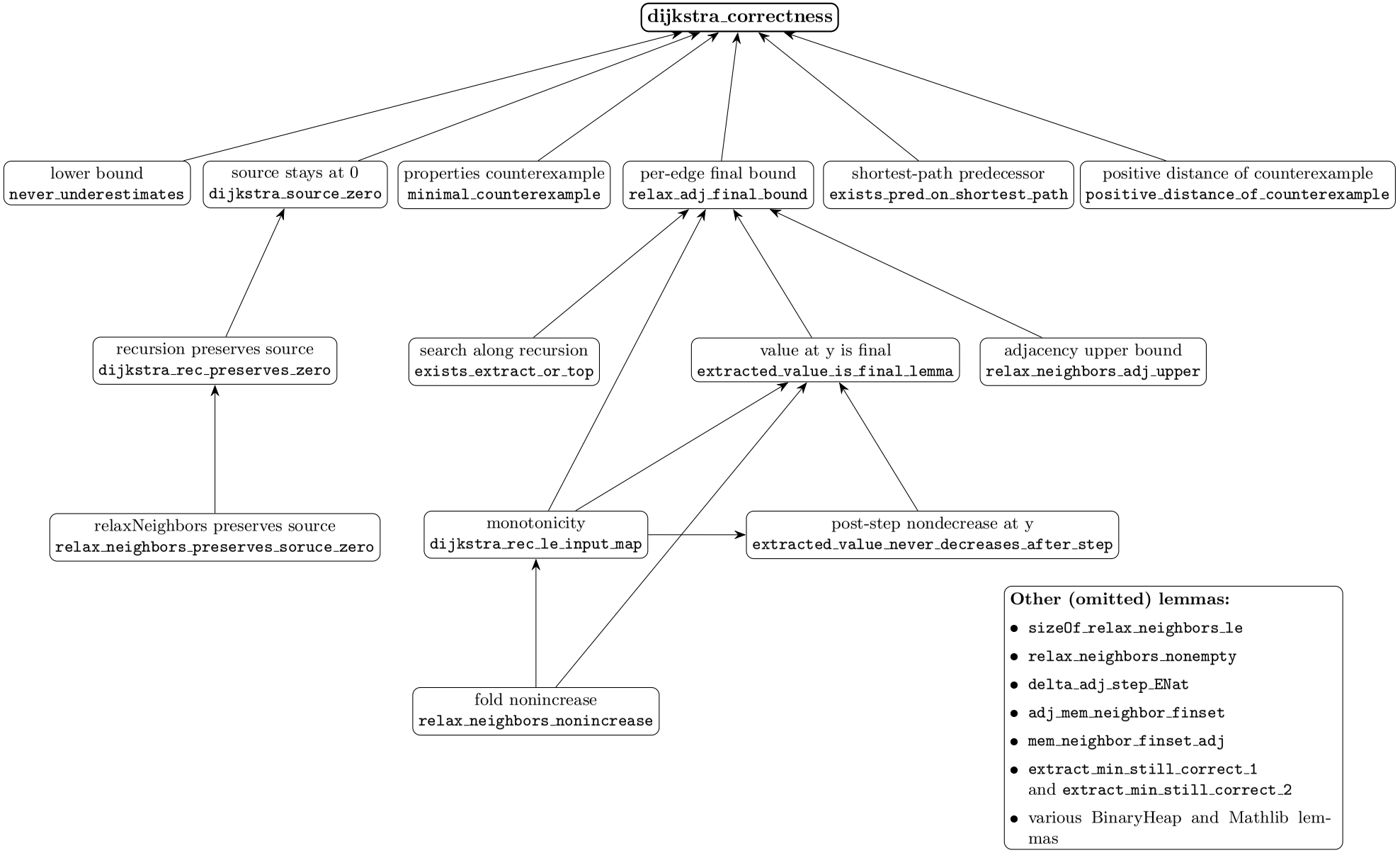

Figure 1: Lemma dependency graph shaping the proof of correctness.

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, 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.

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 $\texttt{dist}(s) = 0$. 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 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 $\delta(s,y) \le \delta(s,u)-1$. Conversely, the triangle inequality and the edge $y \sim u$ give $\delta(s,u) \le \delta(s,y)+1$. Combining these yields equality.

Lean correspondence. The proof uses Mathlib's `SimpleGraph.Walk` API. We first invoke `SimpleGraph.Reachable.exists_wal` 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, $\delta(s,y) \le \texttt{tail.length}$, giving $\delta(s,y)+1 \le \delta(s,u)$. The reverse inequality $\delta(s,u) \le \delta(s,y)+1$ 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:

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

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 $(\texttt{dijkstra\_rec}\ \cdots\ \texttt{dist}'\ \texttt{queue}')\ x \le \texttt{dist}'\ x$. The pointwise nonincrease property of `relaxNeighbors` gives $\texttt{dist}'\ x \le \texttt{dist}\ x$. 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_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 $\texttt{dist}'\ x \le \texttt{dist}\ x$, and we compose with `le_trans`.

Relaxation properties (`relaxNeighbors_nonincrease`, `relaxNeighbors_adj_upper`)

Statement (nonincrease). For all vertices x,

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

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. 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 $\texttt{dist}'(x) \le \texttt{dist}(x)$. 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

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

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 $\texttt{dist}(u) \le \texttt{alt}$ 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 $u \in$ `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 $\leq$ `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: `dist`$(y) =$ (`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 `dist`(y) never decreases in future recursion steps. This uses a local invariant `MinGeYInvariant`, which states that any future extracted vertex u has `dist`$(u) \geq$ `dist`(y). Relaxing neighbors of u can only decrease values to `dist`$(u) + 1$, which is still $\geq$ `dist`$(y) + 1 >$ `dist`(y), so the relaxation cannot decrease `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 `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 `dist`$(y) \leq$ `dist`(u_1), then show that relaxing neighbors of u_1 preserves `dist`(y) because the update condition `dist`$(u_1) + 1 <$ `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 `next`.$1(y) =$ `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 (`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 `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 (`dist`$, q$) satisfying a size bound. The tricky part is ensuring that the one-step equality (`dijkstra_rec` `dist`$_0$ `queue`$_0$) = (`dijkstra_rec` `next`.1 `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 $\texttt{dist}(y) = (\texttt{dijkstra\_rec}\ \texttt{next}.1\ \texttt{next}.2)\ y$, 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:

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