Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms

Anshuman Mohan(✉), Wei Xiang Leow,
and Aquinas Hobor

School of Computing, National University of Singapore,
Singapore, Republic of Singapore
amohan@cs.cornell.edu

Abstract. We develop machine-checked verifications of the full functional correctness of C implementations of the eponymous graph algorithms of Dijkstra, Kruskal, and Prim. We extend Wang *et al.*'s CertiGraph platform to reason about labels on edges, undirected graphs, and common spatial representations of edge-labeled graphs such as adjacency matrices and edge lists. We certify binary heaps, including Floyd's bottom-up heap construction, heapsort, and increase/decrease priority.

Our verifications uncover subtle overflows implicit in standard textbook code, including a nontrivial bound on edge weights necessary to execute Dijkstra's algorithm; we show that the intuitive guess fails and provide a workable refinement. We observe that the common notion that Prim's algorithm requires a connected graph is wrong: we verify that a standard textbook implementation of Prim's algorithm can compute minimum spanning forests without finding components first. Our verification of Kruskal's algorithm reasons about two graphs simultaneously: the undirected graph undergoing MSF construction, and the directed graph representing the forest inside union-find. Our binary heap verification exposes precise bounds for the heap to operate correctly, avoids a subtle overflow error, and shows how to recycle keys to avoid overflow.

Keywords: Separation logic · Graph algorithms · Coq · VST

1 Introduction

Dijkstra's eponymous shortest-path algorithm [22] finds the cost-minimal paths from a distinguished *source* vertex to all reachable vertices in a directed graph. Prim's [61] and Kruskal's [42] algorithms return minimal spanning trees for undirected graphs. Binary heaps are the first priority queue one typically encounters. These algorithms/structures are classic and ubiquitous, appearing widely in textbooks [20,33,36,65,66,68] and in real routing protocol libraries.

In addition to decades of use and textbook analysis, recent efforts have verified one or more of these algorithms in proof assistants and formally proved

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 801–826, 2021.
https://doi.org/10.1007/978-3-030-81688-9_37

claims about their behavior [12,15,30,45,53]. A reasonable person might think that all that can be said, has been. However, we have found that textbook code glosses over a cornucopia of issues that routinely crop up in real-world settings: under/overflows, integration with performant data structures, manual memory (de-)allocation, error handling, casts, memory alignment, *etc.* Further, previous verification efforts with formal checkers often operate within idealized formal environments, which likewise leads them to ignore the same kinds of issues.

In our work, we provide C implementations of each of these algorithms/data structures, and prove in Coq [71] the functional correctness of the same with respect to the formal semantics of CompCert C [50]. By "functional correctness" we mean natural algorithmic specifications; we do not prove resource bounds. Although our C code is developed from standard textbooks, we uncover several subtleties that are absent from the algorithmic and formal methods literature:

§3.2 an overflow in Dijkstra's algorithm, avoiding which requires a nontrivial refinement to the algorithm's precondition to bound edge weights;
§4.2 that the specification of Prim's algorithm can be improved to apply to disconnected graphs without any change to textbook (pseudo-)code;
§4.2 the presence of a wholly unneeded line of (pseduo-)code in Prim's algorithm, and an associated unneeded function argument;
§5 several potential overflows in binary heaps equipped with Floyd's linear-time build-heap function and an edit-priority operation.

We wish to develop general and reusable techniques for verifying graph-manipulating programs written in real programming languages. This is a significant challenge, and so we choose to leverage and/or extend three large existing proof developments to state and prove the full functional correctness of our code in Coq: CompCert; the Verified Software Toolchain [4] (VST) separation logic [59] deductive verifier; and our own previous efforts [73], hereafter dubbed the CertiGraph project. Our primary extensions are to the third, and include:

§2.1 pure/abstract reasoning for graphs with edge labels, (*e.g.*, a distinguished edge-label value for "infinity" that indicates invalid/absent edges);
§2.2 spatial representations and associated reasoning for edge-labeled graphs (several flavors of adjacency matrices as well as edge lists);
§2.3 pure reasoning for undirected graphs (*e.g.*, notions of connectedness).

We prove that our pure machinery and our spatial machinery are well-isolated from each other by verifying several implementations (of each of Dijkstra and Prim) that represent graphs differently in memory but reuse the entire pure portion of the proof. Likewise, we show that our spatial reasoning is generic by reusing graph representations across Dijkstra and Prim. Our verification of Kruskal proves that we can reason about two graphs simultaneously: a directed graph with vertex labels for union-find and an undirected graph with edge labels for which we are building a spanning forest. In addition to our verification of

Dijkstra, Prim, and Kruskal, we develop increased lemma support for the preexisting CertiGraph union-find example [73]. Our extension to "base VST" (*e.g.*, verifications without graphs) primarily consists of our verified binary heap.

The remainder of this paper is organized as follows:

§2 We explain our extensions to CertiGraph: edge-labeled graphs, spatial representations of such graphs, and undirected graphs.
§3 We explain our verification of Dijkstra's algorithm in some detail, discuss a potential overflow, and refine the precondition to avoid it.
§4 We overview our verifications of the Minimum Spanning Tree/Forest algorithms of Prim and Kruskal, focusing on high-level points such as our improved novel specification of Prim's.
§5 We overview our verification of binary heaps, including a discussion of Floyd's bottom-up heap construction and the `edit_priority` operation.
§6 We briefly discuss engineering, *e.g.* statistics for our formal development.
§7 We discuss related work, outline future research directions, and conclude.

Our results are completely machine-checked in Coq and publicly available [1].

2 Extensions to CertiGraph

We begin with the briefest of introductions to CertiGraph's core structure and then detail the extensions we make to various levels of CertiGraph in service of our Dijkstra, Prim, and Kruskal verifications. Ignoring modularity and eliding elements not used in this work, a mathematical graph in CertiGraph is a tuple: ($\mathcal{V}$, $\mathcal{E}$, `vvalid`, `evalid`, `src`, `dst`, `vlabel`, `elabel`, `sound`). Here $\mathcal{V}/\mathcal{E}$ are the carrier types of vertices/edges, `vvalid`/`evalid` place restrictions specifying whether a vertex/edge is valid[1], and `src`/`dst` : $\mathcal{E} \to \mathcal{V}$ map edges to their source/destination. Labels are allowed on vertices and edges, and a `sound`ness condition allows custom application-specific restrictions [73]. Mathematical graphs connect to graphs in computer memory via spatial predicates in separation logic.

2.1 Pure Reasoning for Adjacency Matrix-Represented Graphs

Two of our algorithms operate over graphs represented as adjacency matrices. Not every legal graph can be represented as an adjacency matrix, so we develop a unified, reusable, and extendable `sound`ness condition `SoundAdjMat` that a graph must satisfy in order for it to be represented as an adjacency matrix.

`SoundAdjMat` is parameterized by the graph's `size` and a distinguished number `inf`. We restrict most fields in the tuple: ($\mathcal{V} = \mathbb{Z}$, $\mathcal{E} = \mathbb{Z} \times \mathbb{Z}$, `vvalid` $= \lambda v.\ 0 \le v <$ `size`, `evalid` $= \ldots$, `src` $=$ *fst*, `dst` $=$ *snd*, `vlabel`, `elabel`, `sound` $= \ldots$). We also restrict the carrier type of vertex labels to `unit`

[1] Validity denotes presence in the graph: *e.g.*, if we are using $\mathbb{Z}$ as the carrier type $\mathcal{V}$, and have only 7 vertices, then `vvalid`(x) is probably the proposition $0 \le x < 7$).

and edge labels to $\mathbb{Z}$. We require the parameters `size` and `inf` be strictly positive and representable on the machine. Most critical, however, is the semantics of `evalid`: a valid edge must have a machine-representable label and that label cannot have value `inf`; an invalid edge *must* have label `inf`. Last, the graph must be finite.

The restriction on edge labels is necessary because we are working with labeled adjacency matrices on a real system: we need to set aside a distinguished number `inf` such that edgeweight `inf` indicates the *absence* of an edge. We cannot prescribe some `inf` because client needs can vary widely. For instance, our verifications of Dijkstra's and Prim's algorithms require subtly different `inf`s.

`SoundAdjMat` guarantees spatial representability as an adjacency matrix, but it can be extended with further algorithm-specific restrictions before being plugged in for `sound`. Dijkstra's algorithm requires nonnegative edge weights, and—as we will discuss in §3.2—nontrivial restrictions on `size` and `inf`.

2.2 New Spatial Representations for Edge-Labeled Graphs

We give predicates for adjacency matrices and edge lists for edge-labeled graphs.

Adjacency Matrices. Adjacency matrices enable efficient label access for edge-labeled graphs. We support three common adjacency matrix representations: a stack-allocated 2D array `int graph[size][size]`, a stack-allocated 1D array `int graph[size×size]`, and a heap-allocated 2D array `int **graph`. To the casual observer, these are essentially interchangeable, but that is a mistake when thinking spatially. Apart from the arithmetic that the second flavor uses to access cells, there is a more subtle point: the first and second enjoy a contiguous block of memory, but the third does not: it is an allocated "spine" with pointers to separately-allocated rows. For a taste, the spatial representation of the first is:

$$
\begin{aligned}
\mathit{arr\_addr}(\mathit{ptr}, i, \texttt{size}) &\triangleq \mathit{ptr} + (i \times \texttt{size})\\
\mathsf{array}(\mathit{ptr}, \mathit{list}) &\triangleq \mathop{\Huge\ast}_{i\in[0,|\mathit{list}|)} (\mathit{ptr}+i) \mapsto \mathit{list}[i])\\
\mathsf{arr\_rep}(\gamma, i, \mathit{ptr}) &\triangleq \texttt{let } \mathit{row} \texttt{ := } \mathsf{graph2mat}(\gamma)[i] \texttt{ in}\\
&\ \ \mathsf{array}(\mathit{arr\_addr}(\mathit{ptr}, i, |\mathit{row}|), \mathit{row})\\
\mathsf{graph\_rep}(\gamma, \mathit{g\_addr}, \_) &\triangleq \mathop{\Huge\ast}_{v\in\gamma} \mathsf{arr\_rep}(\gamma, v, \mathit{g\_addr})
\end{aligned}
$$

We use the separation logic $*$ in its iterated form to say that the arrays are separate in memory. We elide details relating to object sizes, pointer alignment, and so forth, although our formal proofs handle such matters. Of particular note are graph2mat, which performs two projections to drag out the graph's nested edge labels into a 2D matrix, and *arr_addr*, which in this instance simply computes the address of any legal row i from the base address of the graph. Notice that this graph_rep predicate ignores its third argument. To represent a heap-allocated 2D array we can still use graph2mat but can no longer use address arithmetic; the third parameter is then a list of pointers to the row sub-arrays.

While ironing out these spatial wrinkles, we develop utilities that easily unfold and refold our adjacency matrices, thus smoothing user experience when reading and writing arrays and cells. Of course these utilities themselves vary by flavor of representation, but the net effect is that users of our adjacency matrices really can be agnostic to the style of representation they are using (see §3.1).

Edge Lists. Edge lists are the representation of choice for sparse graphs. Our C implementation defines an `edge` as a `struct` containing `src`, `dst`, and `weight`, and defines a `graph` as a `struct` containing the graph's size, edge count, and an array of `edges`. Our spatial representation follows this pattern:

$$\mathsf{graph\_rep}(\gamma, g\_addr, e\_addr) \overset{\Delta}{=}$$
$$\big(g\_addr \mapsto (|\gamma.V|, |\gamma.E|, e\_addr)\big) * \mathsf{array}(e\_addr, \gamma.E)$$

2.3 Undirectedness in a Directed World

The CertiGraph library presented in [73] supports only directed graphs, and, as we have seen, bakes direction-reliant idioms such as `src` and `dst` deep into its development. Our challenge is to add support for undirected graphs atop of this.

Our approach is to observe that every directed graph can be treated as an undirected graph by ignoring edge direction. We develop a lightweight layer of "undirected flavored" definitions atop of the existing "directed flavored" definitions, state and prove connections between these, and then build the undirected infrastructure we need. The result is that we retain full access to CertiGraph's graph theory formalizations modulo some mathematical bridging.

Our basic "undirected flavored" definitions are standard [20]. Vertices u and v are `adjacent` if there is an edge between them in either direction; vertices are self-adjacent. A valid `upath` (undirected path) is list of valid vertices that form a pairwise-adjacent chain. Two vertices are `connected` when a valid `upath` features them as head and foot (essentially the transitive closure of `adjacenct`).

The definitions above sync up with preexisting "directed flavored" definitions. Intuitively, undirectedness is more lax than directedness, and so it is unsurprising that these connections are straightforward weakenings of directed properties. We next give standard definitions [20] that culminate in `minimum_spanning_forest`, which is exactly our postcondition of both Prim's and Kruskal's algorithms.[2]

An undirected cycle (`ucycle`) is a valid non-empty `upath` whose first and last vertices are equal. A `connected_graph` means that any two valid vertices are `connected`. `is_partial_graph f g` means everything in `f` is in `g`. We proceed:

```
Definition uforest g :=
 (∀ e, evalid g e → strong_evalid g e) ∧
 (∀ p l, ¬ucycle g p l).
Definition spanning g g' :=
 ∀ u v, connected g u v ↔ connected g' u v.
```

[2] That Prim's postcondition has a *forest* may raise an eyebrow. See §4.2.

```
Definition spanning_uforest f g :=
  is_partial_graph f g ∧ uforest f ∧ spanning f g.
```

The `strong_evalid` predicate means that the `src` and `dst` of the edge are also valid, so *e.g.*, a valid edge cannot point to a deleted/absent vertex. The second conjunct of `uforest` is critical: a forest has no undirected cycles. The other definitions are straightforward from there, and `minimum_spanning_forest f g` means that no other spanning forest has lower total edge cost than `f`.

Our undirected work is also compatible with our new developments in §2.1 and §2.2. An adjacency matrix-representable undirected graph has all the pure properties discussed in `SoundAdjMat`, and also has symmetry across the left diagonal. We extend `SoundAdjMat` into `SoundUAdjMat` by requiring that all valid edges have `src` $\leq$ `dst`. This effectively "turns off" the matrix on one half of the diagonal and avoids double-counting. Prim's algorithm uses `SoundUAdjMat` and places no further restrictions. Further, spatial representations and fold/unfold utilities are shared across directed and undirected adjacency matrices.

3 Shortest Path

We verify a standard C implementation of Dijkstra's algorithm. We first sketch our proof in some detail with an emphasis on our loop invariants, then uncover and remedy a subtle overflow bug, and finish with a discussion of related work.

3.1 Verified Dijkstra's Algorithm in C

Figure 1 shows the code and proof sketch of Dijkstra's algorithm. Red text is used in the figure to highlight changes compared to the annotation immediately prior. Our code is implemented exactly as suggested by CLRS [20], so we refer readers there for a general discussion of the algorithm. The adjacency-matrix-represented graph γ of `size` vertices is passed as the parameter `g` along with the source vertex `src` and two allocated arrays `dist` and `prev`. The spatial predicate $\mathsf{array}(\mathtt{x}, \boldsymbol{v})$, which connects an array pointer `x` with its contents $\boldsymbol{v}$, is standard and unexciting. $\mathsf{PQ}(\mathsf{pq}, \mathit{heap})$ is the spatial representation of our priority queue (PQ) and $\mathsf{Item}(\mathtt{i}, (\mathit{key}, \mathit{pri}, \mathit{data}))$ lays out a struct that we use to interact with the PQ; we leave the management of the PQ to the operations described in§ 5. Of greater interest is $\mathsf{AdjMat}(\mathtt{g}, \gamma)$, which as explained in §2.2, links the concrete memory values of `g` to an abstract mathematical graph γ, which in turn exposes an interface in the language of graph theory (*e.g.*, vertices, edges, labels). Graph γ contains the general adjacency matrix restrictions given in §2.1 along with some further Dijkstra-specific restrictions to be explained in §3.2. We verify Dijkstra three times using different adjacency-matrix representations as explained in §2.2. Thanks to some careful engineering, the C code and the Coq verification are both almost completely agnostic to the form of representation. The only variation between implementations is when reading a cell (line 15), so we refactor this out into a straightforward helper method and verify it separately; accordingly, the proof bases for the three variants differ by less than 1%.

```
1  void dijkstra (int **g, int src, int *dist,
2                 int *prev, int size, int inf {
3  // { AdjMat(g, γ) ∗ array(dist, _) ∗ array(prev, _) ∧ src ∈ γ ∧ connected(γ, src) }
4   Item* temp = (Item*) mallocN(sizeof(Item));
5   int* keys = mallocN (size * sizeof (int));
6   PQ* pq = pq_make(size); int i, u, cost;
7   for (i = 0; i < size; i++)
8   { dist[i] = inf; prev[i] = inf; keys[i] = pq_push(pq,inf,i); }
9   dist[src]= 0; prev[src]= src; pq_edit_priority(pq,keys[src],0);
10  while (pq_size(pq) > 0) {
      //  { ∃dist, prev, popped, heap. AdjMat(g, γ) ∗ PQ(pq, heap) ∗ Item(temp, _) ∗
11          array(dist, dist) ∗ array(prev, prev) ∗ array(keys, keys) ∧
            linked_correctly(γ, heap, keys, dist, popped) ∧
            dijk_correct(γ, src, popped, prev, dist) }
12   pq_pop(pq, temp); u = temp->data;
13   for (i = 0; i < size; i++) {
      //  { ∃dist′, prev′, heap′. AdjMat(g, γ) ∗ PQ(pq, heap′) ∗
            array(dist, dist′) ∗ array(prev, prev′) ∗ array(keys, keys) ∗
14          Item(temp, (keys[u], dist[u], u)) ∧ min(dist[u], heap′) ∧
            linked_correctly(γ, heap′, keys, dist′, popped ⊎ {u}) ∧
            dijk_correct_weak(γ, src, popped ⊎ {u}, prev′, dist′, i, u) }
15    cost = getCell(g, u, i);
16    if (cost < inf) {
17     if (dist[i] > dist[u] + cost) {
18      dist[i] = dist[u] + cost; prev[i] = u;
19      pq_edit_priority(pq, keys[i], dist[i]);
               { ∃dist″, prev″. AdjMat(g, γ) ∗ PQ(pq, ∅) ∗ Item(temp, _) ∗
20   }}}} //     array(dist, dist″) ∗ array(prev, prev″) ∗ array(keys, keys) ∧
                 ∀dst. dst ∈ γ → inv_popped(γ, src, γ.V, prev″, dist″, dst) }
21  freeN (temp); pq_free (pq); freeN (keys); return; }
```

Fig. 1. C code and proof sketch for Dijkstra's algorithm.

Dijkstra's algorithm uses a PQ to greedily choose the cheapest unoptimized vertex on line 12. The best-known distances to vertices are expected to improve as various edges are relaxed, and such improvements need to be logged in the PQ: Dijkstra's algorithm implicitly assumes that its PQ supports the additional operation `decrease_priority`. Our "advanced" PQ (§5.3) supports this operation in logarithmic time with the `pq_edit_priority` function[3].

The first nine lines are standard setup. The *keys* array, assigned on line 8, is thereafter a mathematical constant. The pure predicate *linked_correctly* contains the plumbing connecting the various mathematical arrays. The verification turns on the loop invariants on lines 11 and 14. The pure `while` invariant

[3] Because `decrease_priority` is relatively complex to implement, several popular workarounds (*e.g.* [12]) use simpler PQs at the cost of decreased performance.

$\mathit{dijk\_correct}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist})$ essentially unfolds into:

$$\begin{aligned}\forall \mathit{dst}.\ \mathit{dst} \in \gamma \rightarrow\ & \mathit{inv\_popped}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst}) \wedge \\ & \mathit{inv\_unpopped}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst}) \wedge \\ & \mathit{inv\_unseen}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst})\end{aligned}$$

That is, a destination vertex *dst* falls into one of three categories:

1. *inv_popped*: if $\mathit{dst} \in \mathit{popped}$, then *dst* has been fully processed, *i.e.*, *dst* is reachable from *src* via a globally-optimal path *p* whose vertices are all in *popped*. Path *p* has been logged in *prev* and *p*'s cost is given in *dist*.
2. *inv_unpopped*: if $\mathit{dst} \notin \mathit{popped}$, but its known *dist*ance is less than `inf`, then *dst* is reachable in one step from a popped vertex *mom*. This route is locally optimal: we cannot improve the cost via an alternate popped vertex. Moreover, *prev* logs *mom* as the best-known way to reach *dst*, and *dist* logs the path cost via *mom* as the best-known cost.
3. *inv_unseen*: if $\mathit{dst} \notin \mathit{popped}$ and its known *dist*ance is `inf`, then there is no edge from any *popped* vertex to *dst*; in other words, *dst* is located deeper in the graph than has yet been explored.

After line 12, the above invariant is no longer true: a minimum-cost item *u* has been popped from the PQ, and so the *dist* and *prev* arrays need to be updated to account for this pop. The `for` loop does exactly this repair work. Its pure invariant $\mathit{dijk\_correct\_weak}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, u, i)$ essentially unfolds into:

$$\begin{aligned}&\big(\forall \mathit{dst}.\ \mathit{dst} \in \gamma && \rightarrow \mathit{inv\_popped}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst})\big) \wedge \\ &\big(\forall \mathit{dst}.\ \mathtt{0} \le \mathit{dst} < i && \rightarrow \mathit{inv\_unpopped}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst}) \wedge \\ & && \quad\ \mathit{inv\_unseen}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst})\big) \wedge \\ &\big(\forall \mathit{dst}.\ i \le \mathit{dst} < \mathtt{size} && \rightarrow \mathit{inv\_unpopped\_weak}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst}, u) \wedge \\ & && \quad\ \mathit{inv\_unseen\_weak}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, \mathit{dst}, u)\big)\end{aligned}$$

We now have five cases, many of which are familiar from *dijk_correct*:

1. *inv_popped*: as before; if $\mathit{dst} \in \mathit{popped}$, then it has been fully processed. For all "previously-popped vertices" (*i.e.,* except for *u*), this is trivial from *dijk_correct*. For *u* itself, we reach the heart of Dijkstra's correctness: the locally-optimal path to the cheapest unpopped vertex is *globally* optimal.
2. *inv_unpopped* (less than *i*): as before; if *dst* is reachable in one hop from a popped vertex *mom*, where now *mom* could be *u*. Initially this is trivial since $i = 0$, and we restore it as *i* increments by updating costs when they can be improved, as on lines 18 and 19.
3. *inv_unseen* (less than *i*): as before; some previously unseen neighbors of *u* may be transferred to unpopped status. This is also restored as *i* increments.
4. *inv_unpopped_weak* (between *i* and `size`): if *dst* is reachable in one hop from a previously-popped vertex *mom*, with potentially further improvements possible via *u*. As *i* increments, we strengthen it into *inv_unpopped* after considering whether routing via *u* improves the best-known cost to *dst*.

5. *inv_unseen_weak* (between i and `size`): no edge exists from any previously-popped vertex to *dst*, but there may be one from u. As i increments, we consider whether routing via u reveals a path to *dst*. This is strengthened into *inv_unpopped* if so, and into *inv_unseen* if not.

At the end of the `for` loop the fourth and fifth cases fall away ($i =$ size), and the PQ and the *dist* and *prev* arrays finish "catching up" to the pop on line 12. This allows us to infer the `while` invariant *dijk_correct*, and thus continue the `while` loop. The `while` loop itself breaks when all vertices have been popped and processed. The second and third clauses of the `while` loop invariant *dijk_correct* then fall away, as seen on line 20: all vertices satisfy *inv_popped*, and are either optimally reachable or altogether unreachable. We are done.

3.2 Overflow in Dijkstra's Algorithm

Dijkstra's algorithm clearly cannot work when a path cost is more than `INT_MAX`. A reasonable-looking restriction is to bound edge costs by $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor$, since the longest optimal path has $\texttt{size} - 1$ links and so the most expensive possible path costs no more than `INT_MAX`. However, this has two flaws.

First, since we are writing real code in C, rather than pseudocode in an idealized setting, we must reserve some concrete `int` value `inf` for "infinity". Suppose we set $\texttt{inf} = \texttt{INT\_MAX}$, and that $\texttt{size} - 1$ divides `INT_MAX`. Now the longest path can have cost $(\texttt{size} - 1) \cdot \left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor = \texttt{INT\_MAX} = \texttt{inf}$. This creates an unpleasant ambiguity: we cannot tell if the farthest vertex is unreachable, or if it is reachable with legitimate cost `INT_MAX`. We need to adjust our maximum edge weights to leave room for `inf`; using $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor$ solves this first issue.

Second, even though the best-known distances start at `inf` (see line 8) and only ever decrease from there, the code can overflow on lines 17 and 18. Consider applying Dijkstra's algorithm on a 32-bit unsigned machine to the graph in Fig. 2. The `size` of the graph is 3 nodes, and the proposed edge-weight upper bound is $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor = \left\lfloor \frac{(2^{32}-1)-1}{3-1} \right\rfloor = 2^{31} - 1$, for example as in the graph pictured in Fig. 2. A glance at the figure shows that the true distance from the source A to vertices B and C are $2^{31} - 1$ and $2^{32} - 2$ respectively. Both values are representable with 32 bits, and neither distance is $\texttt{inf} = 2^{32} - 1$, so naïvely all seems well. Unfortunately, Dijkstra's algorithm does not exactly work like that.

After processing vertices A and B, $2^{31} - 1$ and $2^{32} - 2$ *are* the costs reflected in the `dist` array for B and C respectively—*but unfortunately vertex C is still in the priority queue.* After vertex C is popped on line 12, we fetch its neighbors in the `for` loop; the cost from C to B ($2^{31} - 1$) is fetched on line 15. On line 17 the currently optimal cost to B ($2^{31} - 1$) is compared with the sum of the optimal cost to C ($2^{32} - 2$) plus the just-retrieved cost of the edge from C to B ($2^{31} - 1$). Naïvely, $(2^{32} - 2) + (2^{31} - 1)$ is *greater than* the currently optimal cost $2^{31} - 1$, so the algorithm should stick with the latter. However, $(2^{32} - 2) + (2^{31} - 1)$ overflows, with $\left((2^{32} - 2) + (2^{31} - 1)\right) \texttt{ mod } 2^{32} = 2^{31} - 3$, which is *less than*

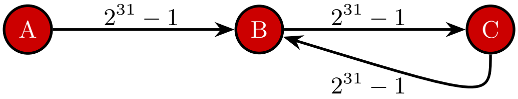

Fig. 2. A graph that will result in overflow on a 32-bit machine.

$2^{31}-1$! Thus the code decides that a new cheaper path from A to B exists (in particular, A⇝B⇝C⇝B) and then trashes the `dist` and `prev` arrays on line 18.

Our code uses signed `int` rather than `unsigned int` so we have undefined behavior rather than defined-but-wrong behavior, but the essence of the overflow is identical. We ensure that the "probing edge" does not overflow by restricting the maximum edge cost further, from $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor$ to $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor$. In Fig. 2, edge weights should be bounded by $\left\lfloor \frac{2^{32}-1}{3} \right\rfloor = 1{,}431{,}655{,}765$; call this value w. Suppose we change the edge weights in Fig. 2 from $2^{31}-1$ to w. Now vertex B has distance w and C has distance $2 \cdot w$. When we remove C from the priority queue, the comparison on line 17 is between the known best cost to B (*i.e.*, w) and the candidate best cost to B via C (*i.e.*, $3 \cdot w = 2^{32} - 1 = \texttt{INT\_MAX}$). There is no overflow, so the candidate is rejected and the code behaves as advertised.

We fold these new restrictions into the mathematical graph γ. In addition to the bounds discussed above, we require a few other more straightforward bounds: edge costs be non-negative, as is typical for Dijkstra; $4 \cdot \texttt{size} \leq \texttt{INT\_MAX}$, to ensure that the multiplication in the `malloc` on line 5 does not overflow; and that $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor \cdot (\texttt{size} - 1) < \texttt{inf}$, so no valid path has cost `inf`. These bounds are optimal: if the input is any less restricted, the postcondition will fail. The last restriction on `inf` is not sufficient when $\texttt{size} = 1$, so in that special case we further require that any (self-loop) edges cost less than `inf`. Whenever $0 < 4 \cdot \texttt{size} \leq \texttt{INT\_MAX}$, the restrictions on `inf` are satisfiable with $\texttt{inf} \overset{\Delta}{=} \texttt{INT\_MAX}$.

3.3 Related Work on Dijkstra in Algorithms and Formal Methods

We were not able to find a reference that gives a robust, precise, and full description of the overflow issue we describe above. Dijkstra's original paper [22] ignores the issue, as do the standard textbooks *Introduction to Algorithms* (*a.k.a.* CLRS) by Cormen *et al.* [20] and *Algorithm Design* by Kleinberg and Tardos [38]. Sedgewick's book on graph algorithms in C [66] sidesteps the overflow in line 17 by requiring weights be in `double`, which *does* have a well-defined positive infinity value and cannot overflow in the traditional sense; Sedgewick and Wayne's *Algorithms* textbook in Java does the same [67]. However, Sedgewick's sidestep entails enduring the inevitable round-off intrinsic to floating-point arithmetic, and so his algorithm computes approximate optimal costs rather than exact ones. Sedgewick does not specify any bounds on input edge weights, and accordingly does not (and cannot) provide any bound on this accumulated error. Sedgewick is silent on how to handle an `int`-weighted input graph. Skiena's *Algorithm*

Design Manual [68] contains code with exactly the bug we identify: he uses integer weights and does not specify any bounds. To its credit, Heineman *et al.*'s *Algorithms in a Nutshell* [33] takes `int` edge weights as inputs and mentions overflow as a possibility. Heineman *et al.* hustle their way around this overflow by performing the arithmetic in line 17 in `long`. However, this cast does not really handle the problem in a fundamental way: if edge weights are given in `long` rather than `int`, then it would be necessary to cast to `long long`; if edge weights are given in `long long`, then Heineman's hustle breaks as there is no bigger type to which to cast. Moreover, Heineman *et al.* do not bound edge weights, so when the cumulative edge weights are too high their code fails silently.

Chen verified Dijkstra in Mizar [15], Gordon *et al.* formalized the reachability property in HOL [29], Moore and Zhang verified it in ACL2 [53], Mange and Kuhn verified it in Jahob [52], Filliâtre in Why3 [25], and Klasen verified it in KeY [37]. Liu *et al.* took an alternative SMT-based approach to verify a Java implementation of Dijkstra [51]. The most recent effort (2019) is by Lammich *et al.*, working within Isabelle/HOL, although they only return the weight of the shortest path rather than the path itself [45]. In general the previous mechanized proofs on Dijkstra verify code defined within idealized formal environments, *e.g.* with unbounded integers rather than machine `int`s and a distinguished non-integer value for infinity. No previous work mentions the overflow we uncover.

4 Minimum Spanning Trees

Here we discuss our verifications of the classic MST algorithms Prim and Kruskal. Although our machine-checked proofs are about real C code, in this section we take a higher-level approach than we did in §3, focusing on our key algorithmic findings and overall experience. Accordingly, we only provide pseudocode for Prim's algorithm rather than a decorated program and do not show any code for Kruskal's. Our development contains our C code and formal proofs [1].

4.1 Prim's Algorithm

We put the pseudocode for Prim's algorithm in Fig. 3; the code on the left-hand side is directly from CLRS, whereas the code on the right omits line 5 and will be discussed in §4.2. Note that line 12 contains an implicit call to the PQ's `edit_priority`. Since the pseudocode only compares `key`s (*i.e.*, edge weights) rather than doing arithmetic on them *à la* Dijkstra, there are no potential overflows and it is reasonable to set `INF` to `INT_MAX` in C.

Indeed, our initial verifications of C code were largely "turning the crank" once we had the definitions and associated lemma support for pure/abstract undirected graphs, forests, *etc.* discussed in §2.3. Accordingly, our initial contribution was a demonstration that this new graph machinery was sufficient to verify real code. We also proved that our extensions to CertiGraph from §2 were generic rather than verification-specific by reusing much pure and spatial reasoning that had been originally developed for our verification of Dijkstra.

```
1  MST-PRIM(G,w,r):
2   for each u in G.V
3    u.key = INF
4    u.parent = NIL
5   r.key = 0
6   Q = G.V
7   while Q $\neq \emptyset$
8    u = EXTRACT-MIN(Q)
9    for each v in G.Adj[u]
10    if v $\in$ Q and w(u,v) $<$ v.key
11     v.parent = u
12     v.key = w(u,v)
```

```
MST-NOROOT-PRIM(G,w):
 for each u in G.V
  u.key = INF
  u.parent = NIL

 Q = G.V
 while Q $\neq \emptyset$
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if v $\in$ Q and w(u,v) $<$ v.key
    v.parent = u
    v.key = w(u,v)
```

Fig. 3. Left: Prim's algorithm from CLRS [20]. Right: the same omitting line 5.

4.2 Prim's Algorithm Handles Multiple Components Out of the Box

Textbook discussions of Prim's algorithm are usually limited to single-component input graphs (*a.k.a.* connected graphs), producing a minimum spanning tree. It is widely believed that Prim's is not directly applicable to graphs with multiple components, which should produce a minimum spanning forest. For example, both Rozen [65] and Sedgewick *et al.* [66,67] leave the extension to multiple components as a formal exercise for the reader, whereas Kepner and Gilbert suggest that multiple-component graphs should be handled by first finding the components and then running Prim on each component [36].

After we completed our initial verification, a close examination of our formal invariants showed us that the algorithm *exactly as given by standard textbooks* will properly handle multi-component graphs *in a single run*. The confusion starts because, in a fully connected graph, any vertex `u` removed from the PQ on line 8 must have `u.key` $<$ `INF`; *i.e.*, `u` must be immediately reachable from the spanning tree that is in the process of being built. However, nothing in the code relies upon this connectedness fact! All we need is that `u` is the "closest vertex" to the "current component." If `u.key` $=$ `INF` *and* `u` is a minimum of the PQ, then it simply means that the "previous component" is done, and we have started spanning tree construction on a new unconnected component "rooted" at `u`, yielding a forest. The node `u`'s parent will remain `NIL`, at it was after the setup loop on line 4, indicating that it is the root of a spanning tree. Its `key` will be `INF` rather than 0, but the `key`s are *internal to Prim's algorithm*: clients only get back the spanning forest as encoded in the `parent` pointers[4].

Having made this discovery, we updated our proofs to support the new weaker precondition, which is what we currently formally verify in Coq [71]. A little further thought led to the realization that since Prim can handle arbitrary numbers

[4] The `key`s simply record the edge-weight connecting a vertex to its candidate parent; recall that line 12 is really a call to the PQ's `edit_priority`. If a client wishes to know this edge weight, it can simply look up the edge in the graph.

of components, the initialization of the root's `key` in line 5 is in fact unnecessary. Accordingly, if we remove this line and the associated function argument `r` from `MST-PRIM` (*i.e.*, the code on the right half of Fig. 3), the algorithm still works correctly. Moreover, *the program invariants become simpler* because we no longer need to treat a specified vertex (`r`) in a distinguished manner. Our formal development verifies this version of the algorithm as well [1].

4.3 Related Work on Prim in Algorithms and Formal Methods

Prim's algorithm was in fact first developed by the Czech mathematician Vojtěch Jarník in 1930 [35] before being rediscovered by Robert Prim in 1957 [61] and a third time by Edsger W. Dijkstra in 1959 [22]. Both Prim's and Dijkstra's treatment explicitly assumes a connected graph; although we cannot read Czech, some time with Google translate suggests that Jarník's treatment probably does the same. The textbooks we surveyed [20,36,38,65–68] seem to derive from Prim's and/or Dijkstra's treatment. More casual references such as Wikipedia [3] and innumerable lecture slides are presumably derived from the textbooks cited. We have not found any references that state that Prim's algorithm *without modification* applies to multi-component graphs, even when executable code is provided: *e.g.*, Heineman *et al.* provide C++ code that aligns closely with our C code [33], but do not mention that their code works equally well on multi-component graphs. Sadly, many sources promulgate the false proposition that modifications to the algorithm are needed to handle multi-component graphs (*e.g.*, [3,36,65–67]). Likewise, we have found no reference that removes the initialization step (line 5 in Fig. 3) from the standard algorithm.

Prim's algorithm has been the focus of a few previous formalization efforts. Guttman formalised and proved the correctness of Prim's algorithm using Stone-Kleene relation algebras in Isabelle/HOL [30]. He works in an idealized formal environment that does not require the development of explicit data structures; his code does not appear to be executable. Lammich *et al.* provided a verification of Prim's algorithm [45]. Lammich *et al.* also work within the idealized formal environment of Isabelle/HOL, but, in contrast to Guttman, develop efficient purely functional data structures and extract them to executable code. Both Guttman and Lammich explicitly require that the input graph be connected.

4.4 Kruskal's Algorithm

Although Kruskal's algorithm is sometimes presented as taking connected graphs and producing spanning trees, the literature also discusses the more general case of multi-component input graphs and spanning forests. However, Kruskal has only recently been the focus of formal verification efforts, partly because it relies on the notoriously difficult-to-verify union-find algorithm; fortunately, the CertiGraph project has an existing fully-verified union-find implementation that we can leverage [73]. Kruskal also requires a sorting function; we implemented `heapsort` as explained in §5.2. Kruskal is optimized for compact representations of sparse graphs, so the $O(1)$ space cost of `heapsort` is a reasonable fit.

The primary interest of our verification of Kruskal is in our proof engineering. Kruskal inputs graphs as edge lists rather than adjacency matrices. In addition to requiring an addition to our spatial graph predicate menu, this means that Kruskal's input graphs can have multiple edges between a given pair of vertices (*i.e.*, a "multigraph"). Pleasingly, we can reuse most of the undirected graph definitions (§2.3), demonstrating that they are generic and reusable.

Another challenge is integrating the pre-existing CertiGraph verification of union-find. We are pleased to say that no change was required for CertiGraph's existing union-find definitions, lemmas, specifications and verification. Kruskal actually manipulates two graphs simultaneously: a directed graph with vertex labels (to store parent pointers and ranks) within union-find, and an undirected multigraph with edge labels (for which the algorithm is constructing a spanning forest). Beyond showing that CertiGraph was capable of this kind of systems-integration challenge, we had to develop additional lemma support to bridge the directed notion of "reachability," used within the directed union-find graph to the undirected notion of "connectedness," used in the MSF graph (§2.3).

4.5 Related Work on Kruskal in Algorithms and Formal Methods

Joseph Kruskal published his algorithm in 1956 [42] and it has appeared in numerous textbooks since (*e.g.*, [20,38,66,68]). Kruskal's algorithm is usually preferred over Prim's for sparse graphs, and is sometimes presented as "the right choice" when confronted with multi-component graphs under the mistaken assumption that Prim's first requires a component-finding initial step.

Guttman generalized minimum spanning tree algorithms using Stone relation algebras [31], and provided a proof of Kruskal's algorithm formatted in said algebras. Like in his work on Prim's [30], Guttmann works within Isabelle/HOL and does not include concrete data structures such as priority-queues and union-find, instead capturing their action as equivalence relations in the underlying algebras. In Guttmann's Kruskal paper, he mentions that his Prim paper axiomatizes the fact that "every finite graph has a minimum spanning forest," which he is then able to prove *using his Kruskal algorithm*. Interestingly, our Prim verification needs the same fact, but we prove it directly.

In a similar vein, Haslbeck *et al.* verified Kruskal's algorithm [32] by building on Lammich *et al.*'s earlier work on Prim [45]. Like Lammich *et al.*, Haslbeck *et al.* work within Isabelle/HOL with a focus on purely functional data structures.

One of the stumbling blocks in verifying Kruskal's algorithm is the need to verify union-find. In addition to CertiGraph [73], two recent efforts to certify union-find are by Charguéraud and Pottier, who also prove time complexity [14]; and by Filliâtre [26], whose proof benefits from a high degree of automation.

5 Verified Binary Heaps in C

A binary heap embeds a heap-ordered tree in an array and uses arithmetic on indices to navigate between a parent and its left and right children [20]. In addition to providing the standard `insert` and `remove-min`/`remove-max` operations (depending on whether it is a min- or max-ordered heap) in logarithmic time, binary heaps can by upgraded to support two nontrivial operations. First, Floyd's `heapify` function builds a binary heap from an unordered array in linear time, and as a related upgrade, `heapsort` performs a worst-case linearithmic-time sort using only constant additional space. Second, binary heaps can be upgraded to support logarithmic-time `decrease-` and `increase-priority` operations, which we generalize straightforwardly into `edit_priority`.

Binary heaps are a good fit for our graph algorithms because Dijkstra's and Prim's algorithms need to edit priorities, and a constant-space `heapsort` is appropriate for the sparse edge-list-represented graphs typically targeted by Kruskal's. The C language has poor support for polymorphic higher-order functions, and a binary heap that supports `edit_priority` is half as fast as a binary heap that does not. Accordingly, we implement binary heaps in C three times:

Name	Heap order	`edit_priority`	`heapify`	Payload
basic	min	no	yes	`void*`
advanced	min	yes	no	`int`
Kruskal	max	no	yes	`int, int` (*i.e.*, unboxed)

Priorities are of type `int`. The Kruskal-specific implementation is stripped down to the bare minimum required to implement `heapsort` (*e.g.*, it does not support `insert`). We next overview these verifications in three parts: basic heap operations, `heapify` and `heapsort` operations, and the `edit_priority` operation.

5.1 The Basic Heap Operations of Insertion and Min/Max-Removal

Because we are juggling three implementations, we take some care to factor our verification to maximize reuse. First, each C implementation has its own exchange and comparison functions that handle the nitty-gritty of the payload and choose between a min or max heap. The following lines are from the "basic" implementation, in which the "payload" (`data` field) is of type `void*`:

```
void exch(unsigned int j, unsigned int k, Item arr[]) {
 int priority = arr[j].priority; void* data = arr[j].data;
 arr[j].priority = arr[k].priority; arr[j].data = arr[k].data;
 arr[k].priority = priority; arr[k].data = data; }
int less(unsigned int j, unsigned int k, Item arr[]) {
 return (arr[j].priority <= arr[k].priority); }
```

These C functions are specified as refinements of Gallina functions that exchange polymorphic data in lists and compare objects in an abstract preordered set; we verify them in VST after a little irksome engineering. The payoff is that the key heap operations, which, following Sedgewick [66], we call `swim` and `sink`, can use identical C code (up to alpha renaming) in all three implementations:

```
void swim(unsigned int k, Item arr[]) {
 while (k > ROOT_IDX && less (k, PARENT(k), arr)) {
  exch(k, PARENT(k), arr); k = PARENT(k);         } }
void sink (unsigned int k, Item arr[], unsigned int available) {
```

```
15 while (LEFT_CHILD(k) < available) {
16  unsigned j = LEFT_CHILD(k);
17  if (j+1 < available && less(j+1, j, arr)) j++;
18  if (less(k, j, arr)) break; exch(k, j, arr); k = j;      } }
```

These functions involve a number of complexities, both at the algorithms level and at the semantics-of-C level. At the C level, there is the potential for a rather subtle bug in the macros ROOT_IDX, PARENT, etc. Abstractly, these are simple: the root is in index 0; the children of x at roughly $2x$ and the parent at roughly $\frac{x}{2}$, with ± 1 as necessary. The danger is thinking that because the variables are `unsigned int`, all arithmetic will occur in this domain; in fact we must force the associated constants into `unsigned int` as well:

```
#define ROOT_IDX   0u
#define PARENT(x)  (x-1u)/2u
#define LEFT_CHILD(x)  (2u*x)+1u
#define RIGHT_CHILD(x)  2u*(x+1u)
```

A second C-semantics issue is the potential for overflow within LEFT_CHILD and RIGHT_CHILD (as well as the increments on line 17), and underflow within the PARENT macro (if x should ever be 0). To avoid this overflow, the precondition of `sink` requires that when `k` is in bounds (*i.e.*, $\texttt{k} < \texttt{available}$), then $2 \cdot (\texttt{available} - 1) \leq \texttt{max\_unsigned}$. An edge case occurs when deleting the last element from a heap ($\texttt{k} = \texttt{available}$); we then require $2 \cdot \texttt{k} \leq \texttt{max\_unsigned}$.

At the algorithmic level, both the `swim` and `sink` functions involve nontrivial loop invariants; `sink` is complicated by the further need to support Floyd's `heapify`, during which a large portion of the array is unordered. Accordingly, we build Gallina models of both functions and show that they restore heap order given a mostly-ordered input heap. There are two different versions of "mostly-ordered". Specifically, `swim` uses a "bottom-up" version:

```
Definition weak_heapOrdered2 (L : list A) (j : nat) : Prop :=
 (∀ i b, i ≠ j → nth_error L i = Some b →
   ∀ a, nth_error L (parent i) = Some a → a ≼ b) ∧
 (grandsOk L j root_idx).
```

whereas `sink` uses a "top-down" version:

```
Definition weak_heapOrdered_bounded (L:list A) (k:nat) (j:nat) :=
 (∀ i a, i ≥ k → i ≠ j → nth_error L i = Some a →
  (∀ b, nth_error L (left_child i) = Some b → a ≼ b) ∧
  (∀ c, nth_error L (right_child i) = Some c → a ≼ c)) ∧
 (grandsOk L j k).
```

The parameter `j` indicates a "hole", at which the heap may not be heap-ordered; `grandsOk` bridges this hole by ordering the parent and the children of `j`:

```
Definition grandsOk (L : list A) (j : nat) (k : nat) : Prop :=
  j ≠ root_idx → parent j ≥ k →
   ∀ gs bb, parent gs = j → nth_error L gs = Some bb →
     ∀ a, nth_error L (parent j) = Some a → a ≼ bb.
```

The parameter `k` is used to support Floyd's `heapify`: it bounds the portion of the list in which elements are heap-ordered (with the exception of `j`). The proofs that the Gallina swim and sink can restore (bounded) heap-orderedness involve a number of edge cases, but given the above definitions go through. The invariants of the C versions of swim and sink are stated via the associated Gallina versions, thereby delegating all heap-ordering proofs to the Gallina versions.

The insertion and remove functions we verify are in fact "non-checking" versions (insert_nc and remove_nc): their preconditions assume there is room in the heap to add or an item in the heap to remove. In the context of Dijkstra and Prim, these preconditions can be proven to hold. The associated verifications involve a little separation logic hackery (specifically, to Frame away the "junk" part of the heap-array from the "live" part), but are straightforward using VST. We avoid the overflow issue in sink by bounding the maximum capacity of the heap: $4 \leq 12 \cdot \texttt{capacity} \leq \texttt{max\_unsigned}$; the magic number 12 comes from the size of the underlying data structure in C. We require users to prove this bound on heap creation, and thereafter handle it under the hood.

5.2 Bottom-Up Heapify and Heapsort

Floyd's bottom-up procedure for constructing a binary heap in linear time, and using a binary heap to sort, are classics of the literature [20,66]. Happily, while the asymptotic bound on heap construction is nontrivial, the implementations of both are basically repeated calls to sink (and exchanges to remove the root):

```
19 void build_heap(Item arr[], unsigned int size) {
20  unsigned int start = PARENT(size);
21  while(1) { sink(start, arr, size);
22             if (start == 0) break; start--;  } }
23 void heapsort_rev(Item* arr, unsigned int size) {
24  build_heap(arr,size);
25  while (size > 1) { size--;
26   exch(ROOT_IDX, size, arr); sink(ROOT_IDX, arr, size); } }
```

Given that in §5.1 we already generalized the specification for sink to handle a portion of the array being unordered, the verification of these functions is straightforward. There is, however, the possibility of a subtle underflow on line 20, in the case when building an empty heap (*i.e.*, size = 0). In turn, this means that heapsort_rev as given above cannot sort empty lists; in our "basic" implementation we strengthen the precondition accordingly, whereas in our "Kruskal" implementation we add a line before 24 that returns when size = 0. We use a max-heap for Kruskal because heapsort yields a *reverse* sorted list.

5.3 Modifying an Element's Priority

To support edit-priority, each live item is associated not only with its usual int priority but also given a unique unsigned int "key", generated during insert and returned to the client. The binary heap internally maintains a secondary array key_table that maps each key to the current location of the associated item within the primary heap array. The client calls `edit_priority` by supplying the key for the item that it wishes to modify, which the binary heap looks up in the `key_table` to locate the item in the heap array before calling `sink` or `swim`. To keep everything linked together, the `key_table` is modified during `exchange`.

To generate the keys on insert, we store a key field within each heap-item in the main array. These keys are initialized to 0..(`capacity` − 1), and thereafter are never modified other than when two cells are swapped during `exchange`. An invariant can then be maintained that the keys from the "live" and "junk" parts have no duplicates. On insertion, we "recycle" the key of the first "junk" item, which is by the invariant known to be appropriately fresh.

5.4 Related Work on Binary Heaps in Algorithms and Formal Methods

J. W. J. Williams published the binary heap data structure, along with heapsort, in June 1964 [28]. Floyd proposed his linear-time bottom-up method to construct such heaps that December [27]. Since then, binary heaps, including Floyd's construction and heapsort, have become a staple of the introductory data structure diet [20]. On the other hand, standard textbooks are surprisingly vague on the implementation of `edit_priority` [20,38,66], and completely silent on the generation of fresh keys during insertion. Our method above of "recycling keys" avoids a subtle overflow in a naïve approach, and does not appear in the literature we examined. The naïve idea is to have a global counter starting at 0, which is then increased on each insert. Unfortunately, this is unsound: during (very) long runs involving both `insert` and `remove-min`, this key counter will overflow. Although overflow is defined in C for `unsigned int`, this overflow is fatal algorithmically: multiple live items could be assigned the same key.

Binary heaps have been verified several times in the literature. They were problem 2 of the VACID-0 benchmark [49], and solved in this regard as well by the Why3 team [69]. These solutions did not implement bottom-up heap construction or edit priority. Summers verified heapsort in Viper, again without bottom-up heap construction [56]. Lammich verified Introsort, which includes a heapsort subroutine [44]. Previous formal work ignores nitty-gritty C issues such as the difference between signed and unsigned arithmetic. We believe we are the first formally verified binary heap to support edit-priority.

6 Engineering Considerations

Verifying real code is meaningfully harder than verifying toy implementations. On top of such challenges, verifying graph algorithms requires a significant amount of mathematical machinery: there are many plausible ways to define basic notions such as reachability, but not all of them can handle the challenges of verifying real code [72]. Moreover, we would like our mathematical, spatial, and verification machinery to be generic and reusable.

All of the above suggests that it is important to work within existing formal proof developments due a strong desire to not reinvent very large wheels (the existing proof bases we work with contain hundreds of thousands of lines of formal proof). We chose to work with the CompCert certified compiler [50]; the Verified Software Toolchain [4], which provides significant tactic support for separation logic-based deductive verification of CompCert C programs; and the CertiGraph framework [73], which provides much pure and spatial reasoning support for verifying graph-manipulating programs within VST. We did so because these frameworks can handle the challenges of real code and because the CertiGraph included several fully verified implementations of union-find that we wished to reuse in our verification of Kruskal's algorithm.

Modular formal proof development involves major software engineering challenges [64]. Accordingly, we took care factoring our extensions to CertiGraph into generic and reusable pieces. This factoring allows us to reuse machinery between verifications, including in the mathematical, spatial, and verification levels. So, *e.g.*, we share significant pure and spatial machinery between Dijkstra, Prim, and Kruskal. Moreover, we maintain good separation between pure and spatial reasoning. So, *e.g.*, both our Dijkstra and Prim verifications can handle multiple spatial variants of adjacency matrices without significant change.

On the other hand, working within existing developments involves some challenges, primarily in that some design decisions have been already made and are hard to change. Moreover, our verifications tickled numerous bugs within VST, including: overly-aggressive automatic entailment simplifying, poor error messages, improper handling of C `structs`, and performance issues. We have been fortunate that the VST team has been willing to work with us to fix such bugs, although some work still remains. Performance remains one area of focus: for example, checking our verification of Kruskal with a 3.7 GHz processor and 32 gb of memory takes more than 22 min even after all of the generic pure and spatial reasoning has been checked, *i.e.* approximately 7 s per line of C code (including whitespace and comments). This performance is unviable for verifying an industrial-sized application of equivalent difficulty: *e.g.*, it would take 13 years for Coq to check the proof for 1,000,000 lines of C. Before some optimizations to our proof structure, the time was significantly longer still.

Our contributions to CertiGraph include pieces that are reused repeatedly and pieces that are more bespoke. Below, we give a sense of both the size of our development (lines of formal Coq proof) and the mileage we get out of our own work via reuse. Items "added with +" are very similar (within 1%) of each other; Prim #4 is the version that does not set the root, *i.e.* on the right in Fig. 3.

Name	Used	LoC	Name	LoC
MathAdjMat	7x	165	DijkSpec1+2+3	301
Undirected	5x	2,139	VerifDijk1+2+3	3,554
MathUAdjMat	4x	1,024	PrimSpec1+2+3+4	508
SpaceAdjMat1+2+3	7x	499	VerifPrim1+2+3+4	7,455
EdgeListGraph	1x	911	KruskalSpec	302
MathDijkGraph	3x	165	VerifKruskal	1,606
DijkPureProof	3x	2,124	VerifHeapSort	568
UndirectedUF	1x	183	VerifBasicBinaryHeap	777
BinaryHeapModel	1x	1,870	VerifAdvBinaryHeap	2,253
Total (pure/spatial)		9,080	Total (verifications)	17,234

In total we have 26,314 novel lines of Coq proof to verify 1,155 lines of C code divided among 12 files, including 3 variants of Dijkstra, 4 variants of Prim, 1 of Kruskal (which includes its `heapsort`), and 2 binary heaps.

7 Concluding Thoughts: Related and Future Work

We have already discussed work directly related to Dijkstra's (§3.3), Prim's (§4.3), and Kruskal's (§4.5) algorithms, as well as binary heaps (§5.4). Summarizing briefly to the point of unreasonableness, our observations about Dijkstra's overflow and Prim's specification are novel, and existing formal proofs focus on code working within idealized environments rather than handling the real-world considerations that we do. We have also discussed the three formal developments we build upon and extend: CompCert, VST, and CertiGraph (Sect. 6). Our goal now is to discuss mechanized graph reasoning and verification more broadly.

Reasoning About Mathematical Graphs. There is a 30+ year history of mechanizing graph theory, beginning at least with Wong [74] and Chou [19] and continuing to the present day; Wang discusses many such efforts [72, §3.3]. The two abstract frameworks that seem closest to ours are those by Noschinski [58]; and by Lammich and Nipkow [45]. The latter is particularly related to our work, because they too start with a directed graph library and must extend it to handle undirected graphs so that they can verify Prim's algorithm.

More-Automated Verification. Broadly speaking, mechanized verification of software falls in a spectrum between more-automated-but-less-precise verifications and less-automated-but-more-precise verifications. Although VST contains some automation, we fall within the latter camp. In the former camp, landmark initial separation logic [63] tools such as Smallfoot [7] have grown into Facebook's industrial-strength Infer [11]. Other notable relatively-automated separation logic-based tools include HIP/SLEEK [17], Bedrock [18], KIV [24], VerCors [9], and Viper [57]. More-automated solutions that use techniques other than separation logic include Boogie [6], Blast [8], Dafny [48], and KeY [2]. In Sect. 3.3 we discuss how some of these more-automated approaches have been applied to verify Dijkstra's algorithm. Petrank and Hawblitzel's Boogie-based verification of a garbage collector [60], Bubel's KeY-based verification of the Schorr-Waite algorithm, and Chen *et al.*'s Tarjan's strongly connected components algorithm in (among others) Why3 [16] are three examples of more-automated verification of graph algorithms. Müller verified *binomial* (not binary) heaps in Viper, although his implementation did not support an edit-priority function [55]. The VOCAL project has verified a number of data structures, including binary and other heaps (all without edit-priority) and union-find [13].

We are not confident that more-automated tools would be able to replicate our work easily. We prove full functional correctness, whereas many more-automated tools prove only more limited properties. Moreover, our full functional correctness results rely upon a meaningful amount of domain-specific knowledge about graphs, which automated tools usually lack. Even if we restrict ourselves to more limited domains such as overflows, several more automated efforts did not uncover the overflow that we described in Sect. 3.3. The proof that certain bounds on edge weights and `inf` suffice depends on an intimate understanding of Dijkstra's algorithm (in particular, that it explores one edge beyond the optimum paths); overall the problem seems challenging in highly-automated settings. The more powerful specification we discover for Prim's algorithm in Sect. 4.2 is likewise not something a tool is likely to discover: human insight appears necessary, at least given the current state of machine learning techniques.

In contrast, several of the potential overflows in our binary heap might be uncovered by more-automated approaches, especially those related to the `PARENT` and `LEFT_CHILD` macros from Sect. 5.1. Although the arithmetic involves both addition/subtraction and multiplication/division, we suspect a tool such as Z3 [54] could handle it. Moreover, a sufficiently-precise tool would probably spot the necessity of forcing the internal constants into `unsigned int`. The issue of sound key generation described in Sect. 5.3 might be a bit trickier. On the one hand, `unsigned int` overflow is defined in C, so real code sometimes relies upon it. Accordingly, merely observing that the counter could overflow does not guarantee that the code is necessarily buggy. On the other hand, some tools might flag it anyway out of caution (*i.e.* right answer, wrong reason).

Less-Automated Verification. Although as discussed above some more-automated tools have been applied to verify graph algorithms, the problem domain is sufficiently complex that many of the verifications discussed in Sect. 3.3, Sect. 4.3, and Sect. 4.5 use less-automated techniques. Two basic approaches are popular. The "shallow embedding" approach is to write the algorithm in the native language of a proof assistant. The "deep embedding" approach is to write the algorithm in another language whose semantics has been precisely defined in the proof assistant. VST uses a deep embedding, and so we do too; one of VST's more popular competitors in the deep embedding style is "Iris Proof Mode" [39]. In contrast, Lammich *et al.* have produced a series of results verifying a variety of graph algorithms using a shallow embedding (*e.g.*, [32,43,45–47]). From a bird's-eye view Lammich *et al.*'s work is the most related to our results in this paper: they verify all three algorithms we do and are able to extract fully-executable code, even if sometimes their focus is a bit different, *e.g.* on novel purely-functional data structures such as a priority queue with `edit_priority`.

Pen-and-Paper Verification of Graph Algorithms. We use separation logic [63] as our base framework. Initial work on graph algorithms in separation logic was minimal; Bornat *et al.* is an early example [10]. Hobor and Villard developed the technique of ramification to verify graph algorithms [34], using a particular "star/wand" pattern to express heap update. Wang *et al.* later integrated ramification into VST as the CertiGraph project we use [73]. Krishna *et al.* [40] have developed a flow algebraic framework to reason about local and global properties of *flow graphs* in the program heap; their flow algebra is mainly used to tackle local reasoning of global graphs in program heaps. Flow algebras should be compatible with existing separation logics; implementation and integration with the Iris project appears to be work in progress [41].

Krishna *et al.* are interested in concurrency [40]; Raad *et al.* provide another example of pen-and-paper reasoning about concurrent graph algorithms [62].

Future Work. We see several opportunities for decreasing the effort and/or increasing the automation in our approach. At the level of Hoare tuples, we see opportunities for improved VST tactics to handle common cases we encounter in graph algorithms. At the level of spatial predicates, we can continue to expand our library of graph constructions, for example for adjacency lists. We also believe there are opportunities to increase modularity and automation at the interface between the spatial and the mathematical levels, *e.g.* we sometimes compare C pointers to heap-represented graph nodes for equality, and due to the nature of our representations this equality check will be well-defined in C when the associated nodes are present in the mathematical graph, so this check should pass automatically.

We believe that more automation is possible at the level of mathematical graphs: for example reachability techniques based on regular expressions over matrices and related semirings [5,23,70]. We are also intrigued by the recent development of various specialized graph logics such as by Costa *et al.* [21] and hope that these kinds of techniques will allow us to simplify our reasoning. The key advantage of having end-to-end machine-checked examples such as the ones we presented above is that they guide the automation efforts by providing precise goals that are known to be strong enough to verify real code.

Conclusion. We extend the CertiGraph library to handle undirected graphs and several flavours of graphs with edge labels, both at the pure and at the spatial levels. We verify the full functional correctness of the three classic graph algorithms of Dijkstra, Prim, and Kruskal. We find nontrivial bounds on edge costs and infinity for Dijkstra and provide a novel specification for Prim. We verify a binary heap with Floyd's `heapify` and `edit_priority`. All of our code is in CompCert C and all of our proofs are machine-checked in Coq.

Acknowledgements. We thank Shengyi Wang for his help and support.

References

1. Functional Correctness of C implementations of Dijkstra's, Kruskal's, and Prim's Algorithms (2021). https://doi.org/10.5281/zenodo.4744664
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification-The KeY Book-From Theory to Practice (2016)
3. Anonymous: Prim's algorithm. https://en.wikipedia.org/wiki/Prim%27s_algorithm
4. Appel, A.W., et al.: Program Logics for Certified Compilers. Cambridge University Press, Cambridge (2014)
5. Backhouse, R., Carré, B.: Regular algebra applied to path-finding problems. J. Inst. Math. Appl. **15**, 161–186 (1975)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_6
8. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**, 505–525 (2007)
9. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
10. Bornat, R., Calcagno, C., O'Hearn, P.: Local reasoning, separation and aliasing. In: SPACE (2004)
11. Calcagno, C., et al.: Moving fast with software verification. In: NASA Formal Methods Symposium (2015)
12. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP (2011)
13. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL - a verified OCaml library. ML Family Workshop (2017)
14. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. **62**, 331–365 (2019)
15. Chen, J.C.: Dijkstra's shortest path algorithm. JFM **15**, 237–247 (2003)
16. Chen, R., Cohen, C., Lévy, J., Merz, S., Théry, L.: Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabelle. In: ITP (2019)
17. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**, 1006–1036 (2010)

18. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI (2011)
19. Chou, C.T.: A formal theory of undirected graphs in HOL. In: HOL (1994)
20. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.S.: Introduction to Algorithms, 3rd edn. (2009)
21. Costa, D., Brotherston, J., Pym, D.: Graph decomposition and local reasoning (2020). Under submission
22. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
23. Dolan, S.: Fun with semirings: a functional pearl on the abuse of linear algebra. In: ICFP (2013)
24. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV - overview and VerifyThis competition. STTT **17**, 677–694 (2015)
25. Filliâtre, J.C.: Dijkstra's shortest path algorithm in Why3 (2011). http://toccata.lri.fr/gallery/dijkstra.en.html
26. Fillitre, J.C.: Simpler proofs with decentralized invariants. J. Log. Algebraic Methods Program. **121**, 100645 (2021)
27. Floyd, R.W.: Algorithm 245: treesort. Commun. ACM **7**(12), 701 (1964)
28. Forsythe, G.E.: Algorithms. Commun. ACM **7**(6), 347–349 (1964)
29. Gordon, M., Hurd, J., Slind, K.: Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 200–215. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_19
30. Guttmann, W.: Relation-algebraic verification of prim's minimum spanning tree algorithm. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 51–68. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_4
31. Guttmann, W.: Verifying minimum spanning tree algorithms with stone relation algebras. J. Log. Algebraic Methods Program. **101**, 132–150 (2018)
32. Haslbeck, M.P.L., Lammich, P.: Refinement with time - refining the run-time of algorithms in Isabelle/HOL. In: ITP (2019)
33. Heineman, G., Pollice, G., Selkow, S.: Algorithms in a Nutshell. O'Reilly (2008)
34. Hobor, A., Villard, J.: Ramifications of sharing in data structures. In: POPL (2013)
35. Jarník, V.: O jistém problému minimálním. (z dopisu panu o. Borůvkovi) (1930)
36. Kepner, Jeremy; Gilbert, J.: Graph algorithms in the language of linear algebra. Soc. Ind. Appl. Math. (2011)
37. Klasen, V.: Verifying Dijkstra's algorithm with KeY. Diploma thesis (2010)
38. Kleinberg, J.M., Tardos, É.: Algorithm Design. Addison-Wesley (2006)
39. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL (2017)
40. Krishna, S., Shasha, D., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. In: POPL (2017)
41. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP (2020)
42. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. **7**, 48–50 (1956)
43. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21
44. Lammich, P.: Efficient verified implementation of Introsort and Pdqsort. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 307–323. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_18

45. Lammich, P., Nipkow, T.: Proof pearl: Purely functional, simple and efficient priority search trees and applications to Prim and Dijkstra. In: ITP (2019)
46. Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds-Karp algorithm. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 219–234. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_14
47. Lammich, P., Sefidgar, S.R.: Formalizing network flow algorithms: a refinement approach in Isabelle/HOL. J. Autom. Reason. **62**(2), 261–280 (2019)
48. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
49. Leino, K.R.M., Moskal, M.: VACID-0: verification of ample correctness of invariants of data-structures. Edition 0 (2010)
50. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL (2006)
51. Liu, T., Nagel, M., Taghdiri, M.: Bounded program verification using an SMT solver: a case study. In: ICST (2012)
52. Mange, R., Kuhn, J.: Verifying Dijkstra's algorithm in Jahob (2007)
53. Moore, J.S., Zhang, Q.: Proof Pearl: Dijkstra's shortest path algorithm verified with ACL2. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 373–384. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_24
54. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
55. Müller, P.: The binomial heap verification challenge in Viper. In: Müller, P., Schaefer, I. (eds.) Principled Software Development. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-98047-8_13
56. Müller, P.: Private correspondence (2021)
57. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
58. Noschinski, L.: A graph library for Isabelle. Math. Comput. Sci. **9**, 23–39 (2015)
59. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
60. Petrank, E., Hawblitzel, C.: Automated verification of practical garbage collectors. Log. Methods Comput. Sci. **6** (2010)
61. Prim, R.C.: Shortest connection networks and some generalizations. Bell Syst. Tech. J. **36**(6), 1389–1401 (1957)
62. Raad, A., Hobor, A., Villard, J., Gardner, P.: Verifying concurrent graph algorithms. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 314–334. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_17
63. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
64. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: a survey of engineering of formally verified software. CoRR (2020)
65. Rosen, K.H.: Discrete Mathematics and Its Applications. 7th edn. (2012)
66. Sedgewick, R.: Algorithms in C, Part 5: Graph Algorithms (2002)
67. Sedgewick, R., Wayne, K.: Algorithms. 4th edn. Addison-Wesley (2011)

68. Skiena, S.: The Algorithm Design Manual, 2nd edn. Springer, Heidelberg (2008)
69. Tafat, A., Marché, C.: Binary heaps formally verified in Why3 (2011)
70. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3), 577–593 (1981)
71. Coq development team: The Coq Proof Assistant. https://coq.inria.fr/
72. Wang, S.: Mechanized verification of graph-manipulating programs. Ph.D. thesis, National University of Singapore (2019)
73. Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying graph-manipulating C programs via localizations within data structures. In: OOPSLA (2019)
74. Wong, W.: A simple graph theory and its application in railway signaling. In: HOL Theorem Proving System and Its Applications (1991)