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

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 [59] 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 MST 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 source to all reachable vertices in a directed graph. Prim's [50] and Kruskal's [36] 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 [18,56,28,55,31,54] 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 claims about their behavior [13,46,12,39,25]. 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 [17] the full functional correctness of the same with respect to the formal semantics of CompCert C [43]. Although our C code is developed from standard textbooks, we uncover a number of subtleties that appear to be absent from the algorithmic and formal methods literature:

§3.2 an overflow in Dijkstra's algorithm whose avoidance 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 [48] deductive verifier, and the CertiGraph project [59]. Our primary extensions are to the latter, 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 [59]. 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 available to reviewers [3].

2 Extensions to CertiGraph

We begin with the briefest of introductions to CertiGraph's core structure and then detail our 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 our work, a mathematical graph in CertiGraph is a tuple: ($\mathcal{V}$, $\mathcal{E}$, `vvalid`, `evalid`, `src`, `dst`, `vlabel`, `elabel`, `sound`). $\mathcal{V}/\mathcal{E}$ 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 [59]. 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` = ..., `src` = *fst*, `dst` = *snd*, `vlabel`, `elabel`, `sound` = ...). We also restrict the carrier type of vertex labels to `unit` 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 positive 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.

[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 $0 \le x < 7$).

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:

$$\begin{aligned}
\mathit{arr_addr}(\mathit{ptr}, i, \texttt{size}) &\triangleq \mathit{ptr} + (i \times \texttt{size}) \\
\mathsf{array}(\mathit{ptr}, \mathit{list}) &\triangleq \mathop{\Large *}_{i \in [0, |\mathit{list}|)} (\mathit{ptr} + i) \mapsto \mathit{list}[i]) \\
\mathsf{arr_rep}(\gamma, i, \mathit{ptr}) &\triangleq \texttt{let } \mathit{row} \texttt{ := } \mathsf{graph2mat}(\gamma)[i] \texttt{ in} \\
&\quad \mathsf{array}(\mathit{arr_addr}(\mathit{ptr}, i, |\mathit{row}|), \mathit{row}) \\
\mathsf{graph_rep}(\gamma, \mathit{g_addr}, _) &\triangleq \mathop{\Large *}_{v \in \gamma} \mathsf{arr_rep}(\gamma, v, \mathit{g_addr})
\end{aligned}$$

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 `edge`s. Our spatial representation follows this pattern:

$$\begin{aligned}
&\mathsf{graph_rep}(\gamma, \mathit{g_addr}, \mathit{e_addr}) \triangleq \\
&\qquad \big(\mathit{g_addr} \mapsto (|\gamma.V|, |\gamma.E|, \mathit{e_addr})\big) * \mathsf{array}(\mathit{e_addr}, \gamma.E)
\end{aligned}$$

2.3 Undirectedness in a directed world

The CertiGraph library presented in [59] 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 [18]. Vertices u and v are `adjacent` if there is an edge between them in either direction; vertices are self-adjacent. An 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 [18] 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.
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 we covered in `SoundAdjMat`, and also has symmetry across the left diagonal. We extend `SoundAdjMat` into `SoundUAdjMat` by adding the condition 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.

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

```
1  void dijkstra (int **g, int src, int *dist,
2                 int *prev, int size, int inf {
3  //  { AdjMat(g, γ) ∗ array(dist, _) ∗ array(prev, _)}
4   Item* temp = (Item*) mallocN(sizeof(Item));
5   int* keys = mallocN (size * sizeof (int));
6   PQ* pq = pq_make(size); int i, j, 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]);
20    }}}} // { ∃dist″, prev″. AdjMat(g, γ) ∗ PQ(pq, ∅) ∗ Item(temp, _) ∗
           //   array(dist, dist″) ∗ array(prev, prev″) ∗ array(keys, keys) ∧
           //   ∀dst. 0 ≤ dst < size → 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.

3.1 Verified Dijkstra's algorithm in C

Figure 1 shows the code and proof sketch of Dijkstra's algorithm. Red text in an annotation indicates changes compared to the annotation immediately prior. Our code is implemented exactly as suggested by CLRS [18], 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. $\mathsf{PQ}(\mathtt{pq}, \mathit{heap})$ 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 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 (vertices, edges, labels, *etc.*). We develop three variations of adjacency-matrix representation (see §2.2) and we verify Dijkstra using each of them. 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%.

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

$$\begin{aligned}\forall dst.\ \texttt{0} \le dst < \texttt{size} \to\ & inv\_popped(\gamma, src, popped, prev, dist, dst) \land \\ & inv\_unpopped(\gamma, src, popped, prev, dist, dst) \land \\ & inv\_unseen(\gamma, src, popped, prev, dist, dst)\end{aligned}$$

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

1. *inv_popped*: *dst* has been popped from the PQ and fully processed. Either *dst* is unreachable, meaning that there exists no finite-cost path from *src* to *dst*, or *dst* is reachable, meaning that a globally optimal path from *src* to *dst* exists, the cost of this path is logged in *dist*, all vertices visited by the path are also popped, and the links of the path are logged in *prev*.
2. *inv_unpopped*: *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. *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*: no finite-cost path exists from any popped vertex to *dst*.

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 $dijk\_correct\_weak(\gamma, src, popped, prev, dist, u, i)$ essentially unfolds into:

$$\begin{aligned}&\forall dst.\ \texttt{0} \le dst < \texttt{size} \to inv\_popped(\gamma, src, popped, prev, dist, dst) \land \\ &\forall dst.\ \texttt{0} \le dst < i \to inv\_unpopped(\gamma, src, popped, prev, dist, dst) \land \\ &\qquad\qquad inv\_unseen(\gamma, src, popped, prev, dist, dst) \land \\ &\forall dst.\ i \le dst < \texttt{size} \to inv\_unpopped\_weak(\gamma, src, popped, prev, dist, dst, u) \land \\ &\qquad\qquad inv\_unseen\_weak(\gamma, src, popped, prev, dist, dst, u)\end{aligned}$$

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

1. *inv_popped*: *dst* has been fully processed and popped from the PQ. 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.

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

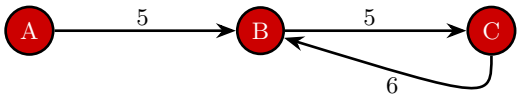

Fig. 2: A graph that will result in overflow on a 4-bit machine.

2. *inv_unpopped* (less than i): *dst* is reachable in one hop from a popped vertex *mom*. 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): no finite-cost path exists from any popped vertex to *dst*. As above, this is restored as i increments.
4. *inv_unpopped_weak* (between i and `size`): *dst* is reachable in one hop from a previously-popped vertex *mom*, with further improvements possible via u. This fact is key: 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 finite-cost path 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 finite-cost 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 `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", with the semantics that if the best-known distance to a vertex x is `inf`, then x is as-yet unreachable. A consequence of this is that reachable destination vertices cannot have a path cost of `inf`: if they did, this would be logged in the `dist` array and create an ambiguity. 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 hypothetical 4-bit unsigned machine to the graph in figure 2. The `size` of the graph is 3 nodes, and so the naïve edge-weight upper bound is $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor = \left\lfloor \frac{15}{3-1} \right\rfloor = 7$, for example as in the graph pictured in figure 2. Indeed, a glance at the figure is enough to tell that the true distance from the source A to vertices B and C are 5 and 10 respectively. Both 5 and 10 are representable with 4 bits, and so naïvely all seems well. Indeed, after

processing vertices A and B, 5 and 10 *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 (6) is fetched on line 15. On line 17 the currently optimal cost to B (5) is compared with the sum of the optimal cost to C (10) plus the just-retrieved cost of the edge from C to B (6). Since $10 + 6$ overflows in 4-bit arithmetic, the comparison is not between 5 and 16 but in fact between 5 and 0! Thus the code decides that a new cheaper path from A to B exists (in particular, A$\rightsquigarrow$B$\rightsquigarrow$C$\rightsquigarrow$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. Our solution is twofold. First, we restrict the maximum edge cost to $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor - 1$, which in the 4-bit setting just described forces an edge cost of no more than 4. Consider modifying figure 2 to have edge weights of 4 rather than 5s and 6s. After processing vertices A and B, the distances to B and C are no more than 4 and 8 respectively. When we process vertex C, the comparison on line 17 is thus between the previous best cost to B (4) and the candidate best cost to B via C (12); there is no overflow and the code behaves as advertised.

The second part of our solution is that we require in the function precondition that $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor - 1 < \texttt{inf} \leq \texttt{INT\_MAX} - \left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor + 1$. As long as $\texttt{size} > 1$, this is perfectly realizable by setting `inf` to $\texttt{INT\_MAX} - \left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor + 1$, *i.e.* in the 4-bit machine we set `inf` to 11. It may be surprising that infinity has a nontrivial upper bound. The reason is that line 17 implicitly assumes that $\infty + \texttt{cost} = \infty$, which certainly is not true for machine integers! Lastly, when $\texttt{size} = 1$, the range bound on infinity collapses and `inf` becomes unrealizable. Our verification is thus not applicable to single-vertex graphs, a special case for which the shortest-path problem is rather uninteresting. We place these new restrictions into Dijktra's soundness condition, in addition to the standard `SoundAdjMat` from §2.1.

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 ignores the issue [22], as does the landmark algorithms textbook CLRS [18]. Sedgewick's book on graph algorithms in C [54] sidesteps the overflow in line 17 by requiring weights be in `double`, which *does* have a well-defined positive infinity value. 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* [56] contains exactly the bug we identify: he sets `inf` to `INT_MAX`, so his code will overflow on line 17 when adding $\texttt{inf} + \texttt{cost}$. To its credit, Heineman *et al.*'s *Algorithms in a Nutshell* [28] takes `int` edge weights as inputs and mentions overflow as a possibility. To avoid this overflow, Heineman *et al.* performs the arithmetic in line 17 in `long` to avoid

```
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 ≠ ∅
8    u = EXTRACT-MIN(Q)
9    for each v in G.Adj[u]
10    if v ∈ 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 ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if v ∈ Q and w(u,v) < v.key
    v.parent = u
    v.key = w(u,v)
```

Fig. 3: Left: Prim's algorithm from CLRS [18]. Right: the same omitting line 5.

this overflow. However, Heineman does not give any bounds on edge weights, and when the cumulative edge weights are too high then his code fails silently.

Chen verified Dijkstra in Mizar [13], Gordon *et al.* formalized the reachability property in HOL [24], Moore and Zhang verified it in ACL2 [46], Mange and Kuhn verified it in Jahob [45], and Klasen verified it in KeY [32]. Liu *et al.* took an alternative SMT-based approach to verify a Java implementation of Dijkstra [44]. The most recent effort (2019) is by Lammich *et al.*, working within Isabelle/HOL, although they simply return the weight of the shortest path rather than the path itself [39]. In general the existing 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. None of the previous formal 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 [3].

4.1 Prim's Algorithm

We put the pseudocode for Prim's algorithm in figure 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.

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 [55] and Sedgewick [54] leave the extension to multiple components as an 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 [31]. This appears to be the standard solution, appearing in numerous lectures and implementations[4].

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.* 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[5].

Having made this discovery, we updated our proofs to support the new weaker precondition, which is what we currently formally verify in Coq [17]. A little further thought led to the realization that since Prim can handle arbitrary numbers 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 figure 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 [3].

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 [30] before being rediscovered by Robert Prim in 1957 [50]

[4] Another standard solution is to use Kruskal's algorithm instead.

[5] 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.

and a third time by Edsger W. Dijkstra in 1959 [21]. 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. Most modern textbook treatment (*e.g.* [31,54,55,56,18]) seems to derive from either Prim's or Dijkstra's treatment. More casual references such as Wikipedia [2] 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 [28], but do not mention that their code works equally well on multi-component graphs. Indeed, many sources promulgate the false proposition that modifications to the algorithm are necessary to handle multi-component graphs (*e.g.*, [31,54,55,2]). Likewise, we have found no reference that removes the initialization step (line 5) in figure 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 [25]. 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 [39]. 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 [59]. 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 [36] and it has appeared in numerous textbooks since (*e.g.*, [18,56,54]). 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 generalised minimum spanning tree algorithms using Stone relation algebras [26], and provided a verified proof of Kruskal's algorithm formatted in said algebras. Like his work on Prim's [25], 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 [27] by building on Lammich *et al.*'s earlier work on Prim [39]. Like Lammich *et al.*, Haslbeck *et al.* work within Isabelle/HOL with a focus on purely functional data structures.

5 Verified binary heaps in C

Binary heaps are a classic data structure for imperative programs [18,54]. Briefly, binary heaps embed a heap-ordered tree in an array and use arithmetic on indices to navigate between a parent and its left and right children. 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*`:

```
5  void exch(unsigned int j, unsigned int k, Item arr[]) {
6   int priority = arr[j].priority; void* data = arr[j].data;
7   arr[j].priority = arr[k].priority; arr[j].data = arr[k].data;
8   arr[k].priority = priority; arr[k].data = data; }
9  int less(unsigned int j, unsigned int k, Item arr[]) {
10  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 [54], we call `swim` and `sink`, can use identical C code (up to alpha renaming) in all three implementations:

```
11 void swim(unsigned int k, Item arr[]) {
12  while (k > ROOT_IDX && less (k, PARENT(k), arr)) {
13   exch(k, PARENT(k), arr); k = PARENT(k);          } }
14 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:

```
1 #define ROOT_IDX  0u                    3 #define LEFT_CHILD(x)  (2u*x)+1u
2 #define PARENT(x) (x-1u)/2u             4 #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:

```
5 Definition weak_heapOrdered2 (L : list A) (j : nat) : Prop :=
6  (forall i b, i <> j -> nth_error L i = Some b ->
7    forall a, nth_error L (parent i) = Some a -> a <<= b) /\
8  (grandsOk L j root_idx).
```

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

```
9 Definition weak_heapOrdered_bounded (L:list A) (k:nat) (j:nat) :=
10  (forall i a, i >= k -> i <> j -> nth_error L i = Some a ->
11   (forall b, nth_error L (left_child i) = Some b -> a <<= b) /\
12   (forall c, nth_error L (right_child i) = Some c -> a <<= c)) /\
13  (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`:

```
1 Definition grandsOk (L : list A) (j : nat) (k : nat) : Prop :=
2   j <> root_idx -> parent j >= k ->
3     forall gs bb, parent gs = j -> nth_error L gs = Some bb ->
4       forall 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 [18,54]. 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 `return`s when `size` $= 0$. We use a max-heap for Kruskal because heapsort yields a *reverse* sorted list.

5.3 Modifying an element's priority

In contrast to `heapify` and `heapsort`, standard algorithms textbooks are vague on the implementation of `edit_priority` [18,54]. The idea is that, during `insert`, each item is associated not only with its usual `int` priority but also a unique `unsigned int` "key". This key is generated during `insert` and returned to the client. Internally, the binary heap 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 associated with the item it wishes to modify, and the binary heap looks up the key 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 `exch`ange.

One detail entirely missing from standard textbooks is how to generate the keys on insert. The initial 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 items could be assigned the same key.

A better method is to store a key field within each heap-item in the main array. These keys are initialized to $0..(\texttt{capacity} - 1)$, and thereafter are never modified other than when two cells are swapped during `exch`ange. 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.

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 [58]. 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 [43]; the Verified Software Toolchain [4], which provides significant tactic support for separation logic-based deductive verification of CompCert C programs; and the CertiGraph framework [59], 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 [53]. 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 `struct`s, 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 3.7GHz processor and 32gb of memory takes more than 22 minutes even after all of the generic pure and spatial reasoning has been checked, *i.e.* approximately 7 seconds 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 figure 3.

Name	Used	LoC	Name	LoC
MathAdjMat	6x	165	DijkSpec1+2+3	301
Undirected	4x	2,139	VerifDijk1+2+3	3,554
MathUAdjMat	3x	1,024	PrimSpec1+2+3+4	508
SpaceAdjMat1+2+3	6x	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

7.1 Related work

We have already discussed work directly related to Dijkstra's (§3.3), Prim's (§4.3), and Kruskal's (§4.5) algorithms in detail, including work from both the algorithms and formal methods literature. 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 already discussed the three formal developments we build upon and extend: CompCert, VST, and CertiGraph (§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 [60] and Chou [16] and continuing to the present day; Wang discusses many such efforts [58, §3.3]. The two abstract frameworks that seem closest to ours are those by Noschinski [47]; and by Lammich and Nipkow [39]. 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 [52] 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 [14], Bedrock [15], and VerCors [9].

More-automated solutions that use techniques other than separation logic include Boogie [6], Blast [8], Dafny [42], and KeY [1]. In §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 [49] gives another more-automated verification of a graph algorithm.

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 §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 §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 §5.1. Although the arithmetic involves both addition/subtraction and multiplication/division, we suspect a tool such as Z3 [20] 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 §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 §3.3, §4.3, and §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 semantics 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" [33]. In contrast, Lammich *et al.* have produced a series of results verifying a variety of graph algorithms using a shallow embedding (*e.g.*, [37,39,27,41,40]). 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`.

Although not related to the main thrust of this paper, Lammich has verified Introsort [38], which includes a heapsort much like the one we present in §5.2. He generates LLVM code, *i.e.* uses a deep embedding.

Pen-and-paper verification of graph algorithms. We use separation logic [52] 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 [29], using a particular "star/wand" pattern to express heap update. Wang *et al.* later integrated ramification into VST as the CertiGraph project we use [59]. Krishna *et al.* [34] 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 [35].

Krishna *et al.* are interested in concurrency [34]; Raad *et al.* provide another example of pen-and-paper reasoning about concurrent graph algorithms [51].

7.2 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 [5,57,23]. We are also intrigued by the recent development of various specialized graph logics such as by Costa *et al.* [19] 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.

7.3 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 have verified 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 have verified 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.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M., Eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*, vol. 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
2. anonymous. Prim's algorithm. https://en.wikipedia.org/wiki/Prim%27s_algorithm.
3. Anonymous. Codebase, 2021. https://github.com/anon-cav/CertiDPK.
4. Appel, A. W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., and Leroy, X. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.
5. Backhouse, R., and Carré, B. Regular Algebra Applied to Path-Finding Problems. *J.Inst.Maths.Applics (1975) 15* (1975), 161–186.
6. Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects* (2005), Springer, pp. 364–387.
7. Berdine, J., Calcagno, C., and O'Hearn, P. W. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO* (2005), pp. 115–137.
8. Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. The software model checker Blast. *Int. J. Softw. Tools Technol. Transf. 9*, 5-6 (2007), 505–525.
9. Blom, S., and Huisman, M. The vercors tool for verification of concurrent programs. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings* (2014), C. B. Jones, P. Pihlajasaari, and J. Sun, Eds., vol. 8442 of *Lecture Notes in Computer Science*, Springer, pp. 127–131.
10. Bornat, R., Calcagno, C., and O'Hearn, P. Local reasoning, separation and aliasing. In *SPACE* (2004), vol. 4.
11. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. Moving fast with software verification. In *NASA Formal Methods Symposium* (2015), Springer, pp. 3–11.
12. Charguéraud, A. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011* (2011), M. M. T. Chakravarty, Z. Hu, and O. Danvy, Eds., ACM, pp. 418–430.
13. Chen, J.-C. Dijkstra's shortest path algorithm. *Journal of Formalized Mathematics 15*, 9 (2003), 237–247.
14. Chin, W. N., David, C., Nguyen, H. H., and Qin, S. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming 77(9)* (2010), 1,006–1,036.
15. Chlipala, A. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), pp. 234–245.
16. Chou, C.-T. A Formal Theory of Undirected Graphs in Higher-Order Logic. In *Higher Order Logic Theorem Proving and Its Applications*. Springer, 1994, pp. 144–157.
17. Coq development team. The Coq proof assistant.
18. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. S. *Introduction to algorithms, 3rd edition*. MIT Press and McGraw-Hill, 2009.

19. Costa, D., Brotherston, J., and Pym, D. Graph decomposition and local reasoning. under submission.
20. de Moura, L. M., and Bjørner, N. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (2008), pp. 337–340.
21. Dijkstra, E. Numerische mathematik, volume 1, chapter a note on two problems in connexion with graphs.
22. Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.
23. Dolan, S. Fun with semirings: a functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming* (2013), pp. 101–110.
24. Gordon, M., Hurd, J., and Slind, K. Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (2003), Springer, pp. 200–215.
25. Guttmann, W. Relation-algebraic verification of prim's minimum spanning tree algorithm. In *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings* (2016), A. Sampaio and F. Wang, Eds., vol. 9965 of *Lecture Notes in Computer Science*, pp. 51–68.
26. Guttmann, W. Verifying minimum spanning tree algorithms with stone relation algebras. *J. Log. Algebraic Methods Program. 101* (2018), 132–150.
27. Haslbeck, M. P. L., and Lammich, P. Refinement with time - refining the runtime of algorithms in isabelle/hol. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA* (2019), J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 20:1–20:18.
28. Heineman, G., Pollice, G., and Selkow, S. *Algorithms in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media: Springfield, MO, USA, 2008.
29. Hobor, A., and Villard, J. The ramifications of sharing in data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)* (2013), pp. 523–536.
30. Jarník, V. O jistém problému minimálním.(z dopisu panu o. Borůvkovi).
31. Kepner, Jeremy; Gilbert, J. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.
32. Klasen, V. Verifying Dijkstra's Algorithm with KeY. *Diploma Thesis* (2010).
33. Krebbers, R., Timany, A., and Birkedal, L. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 205–217.
34. Krishna, S., Shasha, D., and Wies, T. Go with the flow: compositional abstractions for concurrent data structures. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 1–31.
35. Krishna, S., Summers, A. J., and Wies, T. Local reasoning for global graph properties. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30,*

2020, Proceedings (2020), P. Müller, Ed., vol. 12075 of *Lecture Notes in Computer Science*, Springer, pp. 308–335.
36. Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc. 7* (1956), 48–50.
37. Lammich, P. Verified efficient implementation of gabow's strongly connected component algorithm. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings* (2014), G. Klein and R. Gamboa, Eds., vol. 8558 of *Lecture Notes in Computer Science*, Springer, pp. 325–340.
38. Lammich, P. Efficient verified implementation of introsort and pdqsort. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II* (2020), N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12167 of *Lecture Notes in Computer Science*, Springer, pp. 307–323.
39. Lammich, P., and Nipkow, T. Proof pearl: Purely functional, simple and efficient priority search trees and applications to prim and dijkstra. 23:1–23:18.
40. Lammich, P., and Sefidgar, S. R. Formalizing the edmonds-karp algorithm. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings* (2016), J. C. Blanchette and S. Merz, Eds., vol. 9807 of *Lecture Notes in Computer Science*, Springer, pp. 219–234.
41. Lammich, P., and Sefidgar, S. R. Formalizing network flow algorithms: A refinement approach in isabelle/hol. *J. Autom. Reason. 62*, 2 (2019), 261–280.
42. Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers* (2010), pp. 348–370.
43. Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006* (2006), pp. 42–54.
44. Liu, T., Nagel, M., and Taghdiri, M. Bounded Program Verification Using an SMT Solver: A Case Study. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (April 2012), pp. 101–110.
45. Mange, R., and Kuhn, J. Verifying Dijkstra's Algorithm in Jahob. Student project.
46. Moore, J. S., and Zhang, Q. Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2. In *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2005), J. Hurd and T. Melham, Eds., Springer Berlin Heidelberg, pp. 373–384.
47. Noschinski, L. A Graph Library for Isabelle. *Mathematics in Computer Science 9*, 1 (2015), 23–39.
48. O'Hearn, P., Reynolds, J., and Yang, H. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic* (Berlin, Heidelberg, 2001), L. Fribourg, Ed., Springer Berlin Heidelberg, pp. 1–19.
49. Petrank, E., and Hawblitzel, C. Automated verification of practical garbage collectors. *Logical Methods in Computer Science 6* (2010).
50. Prim, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal 36*, 6 (1957), 1389–1401.
51. Raad, A., Hobor, A., Villard, J., and Gardner, P. Verifying concurrent graph algorithms. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings* (2016), A. Igarashi, Ed., vol. 10017 of *Lecture Notes in Computer Science*, pp. 314–334.

52. Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 2002), LICS '02, IEEE Computer Society, pp. 55–74.
53. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., and Tatlock, Z. QED at large: A survey of engineering of formally verified software. *CoRR abs/2003.06458* (2020).
54. Robert, S. Algorithms in C, part 5: Graph algorithms, 2002.
55. Rosen, K. H. *Discrete Mathematics and Its Applications, 7th Edition*. McGraw-Hill, 2012.
56. Skiena, S. *The Algorithm Design Manual, Second Edition*. Springer, 2008.
57. Tarjan, R. E. A unified approach to path problems. *J. ACM 28*, 3 (1981), 577–593.
58. Wang, S. *Mechanized Verification of Graph-Manipulating Programs*. PhD thesis, National University of Singapore, 2019.
59. Wang, S., Cao, Q., Mohan, A., and Hobor, A. Certifying graph-manipulating C programs via localizations within data structures. *PACMPL 3*, OOPSLA (2019), 171:1–171:30.
60. Wong, W. A Simple Graph Theory and Its Application in Railway Signaling. In *HOL Theorem Proving System and Its Applications, 1991., International Workshop on the* (Aug 1991), pp. 395–409.