A Machine-Checked C Implementation of Dijkstra's Shortest Path Algorithm
(Short Paper)

Anshuman Mohan[(†)], Shengyi Wang[(†)], and Aquinas Hobor[(†,‡)]

(†) School of Computing and (‡) Yale-NUS College, National University of Singapore

Abstract. We report on a machine-checked Coq proof of correctness for Dijkstra's one-to-all shortest path algorithm. We prove full functional correctness. We use classic textbook code written in CompCert C, and since our code is executable and realistic our verification must deal with real-world complications. In particular, we encounter and explain an overflow issue in Dijkstra's algorithm. The precise bound in the relevant precondition is nontrivial: we show that the intuitive guess fails and provide a workable refinement. This work fits into an ongoing exploration of verified graph-manipulating algorithms in a realistic setting.

Keywords: Dijkstra's shortest-path · verification · CompCert · VST

1 Introduction

Dijkstra's eponymous shortest-path algorithm [1] finds the cost-minimal paths from a distinguished *source* vertex source to all reachable vertices in a finite directed graph. The algorithm is classic and ubiquitous, appearing widely in textbooks [2] and in real routing protocols. Further, the algorithm has been in use for over 60 years, suggesting that for all practical purposes its safety and correctness have been verified by decades of application.

Here we verify a C implementation of Dijkstra's one-to-all shortest path algorithm. We implement textbook C code [2] in CompCert C [3] so that we can use the Verified Software Toolchain [4] and Wang *et al.*'s recent framework for verifying graph algorithms [5] to state and prove the full functional correctness of the code using the Coq proof assistant [6]. We expose a subtle overflow issue in the C code and make a nontrivial refinement to the precondition of the algorithm so that users know when the code will calculate the correct paths.

The remainder of this paper is organized as follows:

§2 We briefly present and explain our C implementation of Dijkstra's algorithm and the key loop invariants we use to certify its specification.

§3 We explain how overflow can occur and give a refinement to the precondition of the function that restricts edge weights appropriately in the context of a concrete definition for infinity to prevent said overflow.

§4 We conclude this short paper by putting this result in the context of our previous and ongoing work, discussing related work in certifying Dijkstra's algorithm, and summarizing some presentations from algorithms textbooks.

Our code is available at https://github.com/anshumanmohan/VerifDijkstra.

```
1  #define INF INT_MAX - INT_MAX/SIZE
2  void dijkstra (int graph[SIZE][SIZE], int src,
3                             int *dist, int *prev) {
4  // { DijkGraph(γ) * array(dist, _) * array(prev, _) }
5   int pq[SIZE]; int i, j, u, cost;
6   for (i = 0; i < SIZE; i++)
7   {  dist[i] = INF; prev[i] = INF; pq[i] = INF; }
8   dist[src] = 0; pq[src] = 0; prev[src] = src;
9  // { ∃priq, dist, prev. DijkGraph(γ) * PQ(pq, priq) * array(dist, dist) *
   //   array(prev, prev) ∧ dijk_correct(γ, src, prev, dist, priq) }
10  while (!pq_emp(pq)) {
11   u = popMin(pq);
12 // { ∃prev', dist', priq'. DijkGraph(γ) * PQ(pq, priq') * array(dist, dist') *
   //   array(prev, prev') ∧ u ∈ popped(priq') ∧
   //   dijk_correct_weak(γ, src, prev', dist', priq', i, u) }
13   for (i = 0; i < SIZE; i++) {
14    cost = graph[u][i];
15    if (cost < INF) {
16     if (dist[i] > dist[u] + cost)
17     { dist[i] = dist[u] + cost; prev[i] = u; pq[i] = dist[i]; }
18    } // { ∃prev', dist', priq'. DijkGraph(γ) * PQ(pq, priq') * array(dist, dist') *
      //   array(prev, prev') ∧ dijk_correct_weak(γ, src, prev', dist', priq', i, u) }
19   } // { ∃prev', dist', priq'. DijkGraph(γ) * PQ(pq, priq') * array(dist, dist') *
     //   array(prev, prev') ∧ dijk_correct(γ, src, prev', dist', priq') }
20  } // { ∃priq, dist, prev. DijkGraph(γ) * PQ(pq, priq) * array(dist, dist) *
    //   array(prev, prev) ∧ ∀dst ∈ priq. priq[dst] ≮ INF ∧
    //   dijk_correct(γ, src, prev, dist, priq) }
21  return; }
```

Fig. 1: Clight code and proof sketch for Dijkstra's Algorithm

2 Verified Dijkstra in C

Figure 1 shows the code and proof sketch of Dijkstra's algorithm. Our code is implemented exactly as suggested by CLRS [2] 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 `graph` along with the source vertex `src` and two allocated arrays `dist` and `prev`. The details of the spatial predicates $\mathsf{array}(x, \boldsymbol{v})$, connecting an array pointer x with its contents $\boldsymbol{v}$, and the internals of the priority queue PQ are unexciting. Our spatial graph predicate is more interesting in that it nests arrays and connects the concrete memory values to an abstract mathematical graph γ, which in turn exposes an interface in the language of graph theory (vertices, edges, labels, etc.).

$$list_addr((block, offset), index) \triangleq (block, offset + (index \times \texttt{sizeof}(\texttt{int}) \times \texttt{SIZE}))$$

$$\mathsf{list_rep}(\gamma, i, base_ptr) \triangleq \mathsf{array}(\mathsf{graph2matrix}(\gamma)[i], list_addr(base_ptr, i))$$

$$\mathsf{graph_rep}(\gamma, g_addr) \triangleq \mathop{\bigstar}_{v \in \gamma} v \mapsto \mathsf{list_rep}(\gamma, v, g_addr)$$

In general these spatial representations are simple enough that they pose no special challenge in the proof and so we will not focus on issues such as *e.g.* making sure an array dereference is in bounds in our discussion below.

The key to the verification is the pure part of the loop invariants on lines 9 and 12. The `while` invariant $\mathit{dijk_correct}(\gamma, \mathtt{src}, \mathit{prev}, \mathit{dist}, \mathit{priq})$ has three parts:

$$\begin{aligned}\forall \mathit{dst}.\ 0 \le \mathit{dst} < \mathtt{SIZE} \rightarrow\ & \mathit{inv_popped}(\gamma, \mathtt{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \wedge \\ & \mathit{inv_unpopped}(\gamma, \mathtt{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \wedge \\ & \mathit{inv_unseen}(\gamma, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst})\end{aligned}$$

A destination vertex *dst* falls into one of three categories:

1. *inv_popped*: *dst* has been fully processed, and has been popped from the priority queue. A globally optimal path from `src` to *dst* exists, the cost of this path is logged in the `dist` array, and all the vertices visited by the path are also popped. Further, the links of this path are logged in the `prev` array.
2. *inv_unpopped*: *dst* is reachable in one hop from a popped "*mom*" vertex. This route is locally optimal: we cannot improve the cost with a *different* popped vertex. The `prev` array logs *mom* as the best-known way to reach *dst*, and the `dist` array logs the path cost via *mom* as the best-known cost.
3. *inv_unseen*: no path is currently known from `src` to *dst*.

After popping the lowest-cost vertex in line 11, we reach the invariant of the `for` loop $\mathit{dijk_correct_weak}(\gamma, \mathtt{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathtt{i}, \mathtt{u})$:

$$\begin{aligned}&\forall \mathit{dst}.\ 0 \le \mathit{dst} < \mathtt{SIZE} \rightarrow \mathit{inv_popped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \wedge \\ &\qquad\qquad\qquad\qquad\quad \mathit{inv_unseen}(\gamma, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \wedge \\ &\forall \mathit{dst}.\ 0 \le \mathit{dst} < \mathtt{i} \rightarrow \mathit{inv_unpopped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \wedge \\ &\forall \mathit{dst}.\ \mathtt{i} \le \mathit{dst} < \mathtt{SIZE} \rightarrow \mathit{inv_unpopped_weak}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}, \mathtt{u})\end{aligned}$$

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

1. *inv_popped*: *dst* has been fully processed, and has been popped from the priority queue. For all popped vertices *except for* `u` this is trivial from *dijk_correct*; for `u` itself we reach the heart of Dijkstra's correctness: the cost to the fringe vertex with minimum cost cannot be further improved.
2. *inv_unseen*: no path is known from `src` to *dst*. Trivial from *dijk_correct*.
3. *inv_unpopped* (less than `i`): *dst* is reachable in one hop from a "*mom*" vertex, which is itself popped. Initially this is trivial since $\mathtt{i} = 0$ and we restore it as `i` climbs by updating costs when they can be improved in line 17.
4. *inv_unpopped_weak* (between than `i` and `SIZE`): *dst* is reachable in one hop from a popped "*mom*" vertex. However, the path to *dst* specifically *does not go via* `u`. This fact is key to restoring *inv_unpopped* as `i` increments.

The formal definitions for the above predicates can be found in appendix 4.

At the end of the `for` loop, the fourth case will fall away (`i = SIZE`), allowing us to infer the `while` invariant *dijk_correct* from *dijk_correct_weak* and thus continue the `while` loop. The while loop will break precisely when no item in the priority queue has cost less than `INF`. The second clause of the `while` loop invariant *dijk_correct* then falls away: all reachable vertices have been optimized, and the rest are unreachable altogether. We are done.

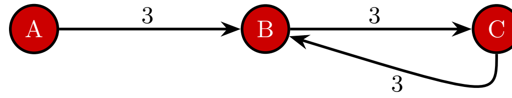

Fig. 2: A graph that will result in overflow on a 3-bit machine

3 I'm sorry Dave, I'm afraid I can't do that: Overflow

A little thought makes it clear that Dijkstra's algorithm cannot work when the cumulative path cost is greater than or equal to `INT_MAX`. A reasonable restriction would seem to be allowing edge costs to be no more than $\left\lfloor \frac{\texttt{INT_MAX}}{\texttt{SIZE}-1} \right\rfloor$, since the longest path can have no more than $\texttt{SIZE}-1$ links and so the maximum cumulative path cost will be no more than `INT_MAX`. However, this has two flaws. First, since are writing real code in C, rather than pseudocode in an idealized setting, we must reserve some concrete `int` value for "infinity" `INF`, which has the special semantics that when a vertex x's distance is `INF` then x must be unreached; if a reached destination vertex could have a legitimate path cost of `INF` then we would have an unpleasant ambiguity. Second, even though path costs start at `INF` and only decrease, the code can overflow in lines 16 and 17.

Consider applying Dijkstra's algorithm on a hypothetical 3-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 $\left\lfloor \frac{\texttt{INT_MAX}}{\texttt{SIZE}-1} \right\rfloor = \left\lfloor \frac{7}{3-1} \right\rfloor = 3$, exactly as pictured in figure 2. Indeed, a glance at the diagram is enough to tell that the true distance from the source A to vertices B and C are 3 and 6, respectively—both of which are representable with 3 bits, and so naïvely all seems well. Indeed, after processing vertices A and B, 3 and 6 **are** the costs reflected in the `dist` array for B and C, respectively—*but unfortunately vertex C is still in the priority queue* `pq`. After vertex C is popped on line 11, we fetch its neighbors in the `for` loop; vertex B's cost of 3 will be fetched on line 14. On line 16 the currently optimal cost of B (3) is compared against the sum of the optimal cost of C (6) plus the just-retrieved cost of the edge from C to B (3). Since $6 + 3$ overflows in 3-bit arithmetic, the comparison is not between 3 and 9 but in fact between 3 and 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 17.

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 $\left\lfloor \frac{\texttt{INT_MAX}}{\texttt{SIZE}} \right\rfloor$, which in the 3-bit setting just described forces an edge cost of no more than 2. Second, on line 1 we choose `INF` to be $\texttt{INT_MAX} - \left\lfloor \frac{\texttt{INT_MAX}}{\texttt{SIZE}} \right\rfloor$, which in the 3-bit setting is 5. Consider modifying figure 2 to have edge weights of 2 rather than 3. After processing vertices A and B, the distances to B and C are no more than 2 and 4, respectively. When we process vertex C, the comparison on line 16 is thus between the previous best cost to B of 2, and the candidate best cost to B via C of 6; there is no overflow and the code behaves as advertised.

4 Concluding thoughts: Future and Related Work

Our previous work. We have long been interested in the verification of graph-manipulating programs written in C [7]. We fortified our techniques to handle realistic (CompCert [3]) C to a machine-checked level of rigour [5]. Novel features of the present result include a previously-untried adjacency matrices spatial graph representation as well as non-trivial edge labels between graph nodes.

Ongoing and future work. We are investigating techniques to increase the automation of such verifications. Although we benefit from some automation at the Hoare-logic level provided by the Verified Software Toolchain [4], building these proofs is still highly labor intensive. We see potential for automation in four areas: (A) the Hoare level; (B) the spatial level; (C) the mathematical level; and (D) the interface between the spatial and the mathematical levels. Our ongoing work on these challenges include (A) improved tactics for VST for common cases we encounter in graph algorithms; (B) an expanded library of existing graph constructions such as the adjacency-matrix representation used in this result, as well as associated lemmas; (C) better lemmas about common mathematical graph patterns, investigations into reachability techniques based on regular expressions over matrices and related semirings [8,9,10,11]; and (D) improved modularity in our constructions and automation of common cases, *e.g.* we often 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. The key advantage of having end-to-end machine-checked examples such as the one 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.

Other verifications of shortest-path. Chen verified Dijkstra in Mizar [12], Gordan *et al.* formalized the reachability property in HOL [13], Moore and Zhang verified it in ACL2 [14], Mange and Kuhn verified it in Jahob [15], and Klasen verified it in KeY [16]. Liu *et al.* took an alternative SMT-based approach to verify a Java implementation of Dijkstra [17]. In general the above work operates within idealized formal environments and thus gloss over certain real-world issues, and so as best we can tell do not *e.g.* handle the overflow issue we identified.

Shortest-path in algorithms textbooks. We were not able to find a standard textbook that gives a robust, precise, and full description of the overflow issue we describe in §3. The landmark CLRS [2] does not address overflow, although—arguably—this is unnecessary in pseudocode. Sedgewick's book on graph algorithms in C [18] likewise does not address overflow, and moreover gives C code that contains exactly the bug we identify. Skiena's *Algorithm Design Manual* likewise glosses over the issue, both in pseudocode and C [19]. To its credit, Heineman *et al.*'s *Algorithms in a Nutshell* [20] mentions overflow as a possibility and proposes casting certain computations to `long` to avoid them in C. However, Heineman *et al.* do not give a nontrivial bound on the input edge weights, leaving users in the fog for exactly when the algorithm works.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.S.: Introduction to algorithms, 3rd edition. MIT Press and McGraw-Hill (2009)
3. 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) 42–54
4. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, New York, NY, USA (2014)
5. Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying graph-manipulating C programs via localizations within data structures. PACMPL **3**(OOPSLA) (2019) 171:1–171:30
6. Coq development team: The Coq proof assistant
7. Hobor, A., 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) 523–536
8. Backhouse, R., Carré, B.: Regular Algebra Applied to Path-Finding Problems. J.Inst.Maths.Applics (1975) 15 (1975) 161–186
9. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3) (1981) 577–593
10. 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) 101–110
11. Krishna, S., Shasha, D., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. Proceedings of the ACM on Programming Languages **2**(POPL) (2017) 1–31
12. Chen, J.C.: Dijkstra's shortest path algorithm. Journal of Formalized Mathematics **15**(9) (2003) 237–247
13. Gordon, M., Hurd, J., 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, Springer (2003) 200–215
14. Moore, J.S., Zhang, Q.: Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2. In Hurd, J., Melham, T., eds.: Theorem Proving in Higher Order Logics, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 373–384
15. Mange, R., Kuhn, J.: Verifying Dijkstra's Algorithm in Jahob. (2007) Student project.
16. Klasen, V.: Verifying Dijkstra's Algorithm with KeY. Diploma Thesis (2010)
17. Liu, T., Nagel, M., 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) 101–110
18. Robert, S.: Algorithms in c, part 5: Graph algorithms (2002)
19. Skiena, S.: The Algorithm Design Manual, Second Edition. Springer (2008)
20. Heineman, G., Pollice, G., Selkow, S.: Algorithms in a nutshell (in a nutshell (o'reilly)) (2008)

A Formal definitions used in decorated proof

In this appendix we put more formal and complete versions of the definitions discussed in the paper proper. Figure 3 shows how we represent graphs in memory using adjacency matrices. Figure 4 shows the pure mathematical definitions that were discussed in §2, and exposes several of the internal definitions that make them work.

$$\begin{aligned}
\mathit{list_addr}((\mathit{block}, \mathit{offset}), \mathit{index}) &\triangleq (\mathit{block}, \mathit{offset} + (\mathit{index} \times \texttt{sizeof(int)} \times \texttt{SIZE})) \\
\mathsf{list_rep}(\gamma, i, \mathit{base_ptr}) &\triangleq \mathsf{array}(\mathsf{graph2matrix}(\gamma)[i], \mathit{list_addr}(\mathit{base_ptr}, i)) \\
\mathsf{graph_rep}(\gamma, \mathit{g_addr}) &\triangleq \mathop{\bigstar}_{v \in \gamma} v \mapsto \mathsf{list_rep}(\gamma, v, \mathit{g_addr})
\end{aligned}$$

Fig. 3: Spatial representation of the graph

$$\mathit{path_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{mom}, p) \triangleq \mathit{valid_path}(\gamma, p) \land \mathit{path_ends}(\gamma, p, \mathit{src}, \mathit{dst}) \land$$
$$\mathit{path_cost}(\gamma, p) \neq \texttt{INF} \land \mathit{dist}[\mathit{dst}] = \mathit{path_cost}(\gamma, p) \land \forall a, b.\ (a, b) \in p \to \mathit{prev}[b] = a$$

$$\mathit{path_globally_optimal}(\gamma, \mathit{src}, \mathit{dst}, p) \triangleq \forall p'.\ \mathit{valid_path}(\gamma, p') \to \mathit{path_ends}(\gamma, p', \mathit{src}, \mathit{dst}) \to$$
$$\mathit{path_cost}(\gamma, p) \leq \mathit{path_cost}(\gamma, p')$$

$$\mathit{all_hops_in_popped}(p, \mathit{priq}) \triangleq \forall a, b.\ (a, b) \in p \to a \in \mathit{popped}(\mathit{priq}) \land b \in \mathit{popped}(\mathit{priq})$$

$$\mathit{all_hops_in_popped_weak}(p, \mathit{priq}, \texttt{u}) \triangleq \forall a, b.\ (a, b) \in p \to a \in \mathit{popped}(\mathit{priq}) \land$$
$$b \in \mathit{popped}(\mathit{priq}) \land a \neq \texttt{u} \land b \neq \texttt{u}$$

$$\mathit{dijk_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}) \triangleq$$
$$\forall \mathit{dst}.\ \texttt{0} \leq \mathit{dst} < \texttt{SIZE} \to \mathit{inv_popped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \land$$
$$\mathit{inv_unpopped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \land$$
$$\mathit{inv_unseen}(\gamma, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst})$$

$$\mathit{dijk_correct_weak}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \texttt{i}, \texttt{u}) \triangleq$$
$$\forall \mathit{dst}.\ \texttt{0} \leq \mathit{dst} < \texttt{SIZE} \to \mathit{inv_popped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \land$$
$$\mathit{inv_unseen}(\gamma, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \land$$
$$\forall \mathit{dst}.\ \texttt{0} \leq \mathit{dst} < \texttt{i} \to \mathit{inv_unpopped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \land$$
$$\forall \mathit{dst}.\ \texttt{i} \leq \mathit{dst} < \texttt{SIZE} \to \mathit{inv_unpopped_weak}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}, \texttt{u})$$

$$\mathit{inv_popped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \triangleq \mathit{dst} \in \mathit{popped}(\mathit{priq}) \to$$
$$\exists \mathit{p2dst}.\ \mathit{path_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{dst}, \mathit{p2dst}) \land$$
$$\mathit{all_hops_in_popped}(\mathit{p2dst}, \mathit{priq}) \land \mathit{path_globally_optimal}(\gamma, \mathit{src}, \mathit{dst}, \mathit{p2dst})$$

$$\mathit{inv_unpopped}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \triangleq \mathit{priq}[\mathit{dst}] < \texttt{INF} \to$$
$$\texttt{let}\ \mathit{mom}\ \texttt{:=}\ \mathit{prev}[\mathit{dst}]\ \texttt{in}\ \exists \mathit{p2mom}.\ \mathit{path_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{mom}, \mathit{p2mom}) \land$$
$$\mathit{all_hops_in_popped}(\mathit{p2mom}, \mathit{priq}) \land \mathit{path_globally_optimal}(\gamma, \mathit{src}, \mathit{mom}, \mathit{p2mom}) \land$$
$$\forall \mathit{mom}', \mathit{p2mom}'.\ \mathit{path_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{mom}', \mathit{p2mom}') \to$$
$$\mathit{all_hops_in_popped}(\mathit{p2mom}', \mathit{priq}) \to \mathit{path_globally_optimal}(\gamma, \mathit{src}, \mathit{mom}', \mathit{p2mom}') \to$$
$$\mathit{path_cost}(\mathit{p2mom} + (\mathit{mom}, \mathit{dst})) \leq \mathit{path_cost}(\mathit{p2mom}' + (\mathit{mom}', \mathit{dst}))$$

$$\mathit{inv_unpopped_weak}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \triangleq \mathit{priq}[\mathit{dst}] < \texttt{INF} \to$$
$$\texttt{let}\ \mathit{mom}\ \texttt{:=}\ \mathit{prev}[\mathit{dst}]\ \texttt{in}\ \exists \mathit{p2mom}.\ \mathit{path_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{mom}, \mathit{p2mom}) \land$$
$$\mathit{all_hops_in_popped_weak}(\mathit{p2mom}, \mathit{priq}) \land \mathit{path_globally_optimal}(\gamma, \mathit{src}, \mathit{mom}, \mathit{p2mom}) \land$$
$$\forall \mathit{mom}', \mathit{p2mom}'.\ \mathit{path_correct}(\gamma, \mathit{src}, \mathit{prev}, \mathit{dist}, \mathit{mom}', \mathit{p2mom}') \to$$
$$\mathit{all_hops_in_popped_weak}(\mathit{p2mom}', \mathit{priq}) \to \mathit{path_globally_optimal}(\gamma, \mathit{src}, \mathit{mom}', \mathit{p2mom}') \to$$
$$\mathit{path_cost}(\mathit{p2mom} + (\mathit{mom}, \mathit{dst})) \leq \mathit{path_cost}(\mathit{p2mom}' + (\mathit{mom}', \mathit{dst}))$$

$$\mathit{inv_unseen}(\gamma, \mathit{prev}, \mathit{dist}, \mathit{priq}, \mathit{dst}) \triangleq \mathit{priq}[\mathit{dst}] = \texttt{INF} \to (\mathit{dist}[\mathit{dst}] = \texttt{INF} \land \mathit{prev}[\mathit{dst}] = \texttt{INF})$$

Fig. 4: Key mathematical used in the decorated proof