Proof Pearl: Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra

Peter Lammich
The University of Manchester, UK

Tobias Nipkow
Technical University Munich, Germany
`http://www.in.tum.de/~nipkow`

Abstract

The starting point of this paper is a new, purely functional, simple and efficient data structure combining a search tree and a priority queue, which we call a *priority search tree*. The salient feature of priority search trees is that they offer a decrease-key operation, something that is missing from other simple, purely functional priority queue implementations. As two applications of this data structure we verify purely functional, simple and efficient implementations of Prim's and Dijkstra's algorithms. This constitutes the first verification of an executable and even efficient version of Prim's algorithm.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Priority queue, Dijkstra's algorithm, Prim's algorithm, verification, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.23

Supplement Material `https://www.isa-afp.org/entries/Priority_Search_Trees.html`, `https://www.isa-afp.org/entries/Prim_Dijkstra_Simple.html`

Funding *Peter Lammich*: DFG Grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases"
Tobias Nipkow: Supported by DFG Koselleck grant NI 491/16-1.

1 Introduction

The standard implementations (e.g. [6]) of a number of efficient algorithms (e.g. Prim [26] and Dijkstra [7]) require a priority queue with a decrease-key operation. The latter operation is easy to realize efficiently in an imperative setting but harder in a functional one because one cannot use a pointer into the priority queue. The starting point of this paper is an extremely simple and yet efficient functional data structure that supports the usual search tree operations plus the priority queue operations including decrease-key. It can be realized on top of any kind of binary search tree by augmenting it with priority information. We call it a *priority search tree*. Based on this data structure we implement and verify two classic efficient algorithms, Prim and Dijkstra, in a purely functional manner. This is the first formal verification of an executable version of Prim's algorithm and we discuss its details. The work is carried out in the theorem prover Isabelle/HOL [20, 21].

The paper is structured as follows: Section 2 introduces some Isabelle specific notations. Section 3 presents the ADT of priority maps and its efficient realization via priority search trees. Section 4 introduces undirected graphs. Section 5 details the verification of Prim's algorithm. Finally, Section 6 sketches the verification of Dijkstra's algorithm. The discussion of related work is found in each section.

The Isabelle/HOL sources of the formalizations discussed in this paper are available in the Archive of Formal Proofs [16, 17].

2 Notation

Type variables are denoted by $'a$, $'b$, etc. Most type constructors follow postfix syntax, e.g. τ *set* is the type of sets of elements of type τ. Function types are denoted by the infix $\Rightarrow$. Function update is written as $f(x{:=}y)$.

Type *nat* is the type of natural numbers.

On sets, the unary $-$ is complement and the binary $-$ is difference. The image of a set S under a function f is written f ‘ S.

Lists (type τ *list*) are constructed from the empty list [] via the infix cons-operator (#). The infix (@) concatenates two lists. Function *set* converts a list into a set.

The *option* type is also predefined: **datatype** $'a$ *option* = *None* | *Some* $'a$.

3 Priority Maps and Priority Search Trees

3.1 Priority Maps

A *priority map* is a map from keys (type $'a$) to values (type $'b$) where the values (“priorities”) are linearly ordered and a key with minimal value can be extracted. This is the interface:

$\mathit{empty} :: {'m}$
$\mathit{update} :: {'a} \Rightarrow {'b} \Rightarrow {'m} \Rightarrow {'m}$
$\mathit{delete} :: {'a} \Rightarrow {'m} \Rightarrow {'m}$
$\mathit{is\_empty} :: {'m} \Rightarrow \mathit{bool}$
$\mathit{lookup} :: {'m} \Rightarrow {'a} \Rightarrow {'b}\ \mathit{option}$
$\mathit{getmin} :: {'m} \Rightarrow {'a} \times {'b}$

The first five operations are the canonical ones for maps (which is why we omit their specification); function *update* subsumes decrease-key (which should be called decrease-priority). Function *getmin* extracts the key with the minimal value. Its specification is:

$$\mathit{getmin}\ m = (k,\ p) \wedge \mathit{invar}\ m \wedge \neg\ \mathit{lookup}\ m = (\lambda x.\ \mathit{None}) \longrightarrow$$
$$\mathit{lookup}\ m\ k = \mathit{Some}\ p \wedge (\forall\, p' \in \mathit{ran}\ (\mathit{lookup}\ m).\ p \le p')$$

where *invar* is the representation invariant and $\mathit{ran}\ m = \{b \mid \exists\, a.\ m\ a = \mathit{Some}\ b\}$ is the range of a map.

3.2 Priority Search Trees

The first contribution of this paper is a truly simple implementation of priority maps by means of augmented binary search trees. That is, the basic data structure is some arbitrary binary search tree, e.g. a red-black tree, implementing the map from $'a$ to $'b$ by storing pairs (k,p) in each node. At this point we need to assume that the keys are also linearly ordered. To implement *getmin* efficiently we annotate/augment each node with another pair (k',p'), the intended result of *getmin* when applied to that subtree. The specification of *getmin* tells us that (k',p') must be in that subtree and that p' is the minimal priority in that subtree. Thus the annotation can be computed by passing the (k',p') with the minimal p' up the tree. We will now make this more precise for balanced binary trees in general.

We assume that our trees are either leaves of the form $\langle\rangle$ or nodes of the form $\langle l, kp, b, r\rangle$ where l and r are subtrees, kp is the contents of the node (a key-priority pair) and b is some additional balance information (e.g. colour, height, size, …). Augmented nodes are of the form $\langle l, kp, (b, kp'), r\rangle$.

The implementation of *getmin* is trivial: $getmin\ \langle\_, \_, (\_, kp'), \_\rangle = kp'$. It remains to upgrade the existing map operations to work with augmented nodes. Therefore we now show how to transform any function definition on un-augmented trees into one on trees augmented with (k',p') pairs. A defining equation $f\ pats = e$ for the original type of nodes is transformed into an equation $f\ pats' = e'$ on the augmented type of nodes as follows:

- Every pattern $\langle l, kp, b, r\rangle$ in *pats* and e is replaced by $\langle l, kp, (b, \_), r\rangle$ to obtain $pats'$ and e_2.
- To obtain e', every expression $\langle l, kp, b, r\rangle$ in e_2 is replaced by $node\ l\ kp\ b\ r$ where

$$node\ l\ a\ c\ r = \langle l, a, (c, min\_kp\ a\ l\ r), r\rangle$$

$$\begin{aligned}
&min\_kp\ kp\ l\ r = \\
&(\mathsf{case}\ (l, r)\ \mathsf{of}\ (\langle\rangle, \langle\rangle) \Rightarrow kp \\
&\ \mid (\langle\rangle, \langle l_2, a_2, (b_2, kp_2), r_2\rangle) \Rightarrow min2\ kp\ kp_2 \\
&\ \mid (\langle l_1, a_1, (b_1, kp_1), r_1\rangle, \langle\rangle) \Rightarrow min2\ kp\ kp_1 \\
&\ \mid (\langle l_1, a_1, (b_1, kp_1), r_1\rangle, \langle l_2, a_2, (b_2, kp_2), r_2\rangle) \Rightarrow \\
&\qquad min2\ kp\ (min2\ kp_1\ kp_2))
\end{aligned}$$

$$min2 = (\lambda(k, p)\ (k', p').\ \mathsf{if}\ p \le p'\ \mathsf{then}\ (k, p)\ \mathsf{else}\ (k', p'))$$

Note that this transformation does not affect the asymptotic complexity of *f*. Therefore the priority search tree operations have the same complexity as the underlying search tree operations, i.e. typically logarithmic (*update*, *delete*, *lookup*) and constant time (*empty*, *is_empty*). For brevity we simply speak of *efficient* in the rest of the paper.

As an example, consider red-black trees where the balancing information b is one of the two colours *Red* or *Black*. In the functional definition of red-black trees due to Okasaki [25] there is a *balance* function that eliminates red-red configurations. We consider a slight variant *baliL* [14] where one of the defining equations is

$$baliL\ (R\ (R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 = R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)$$

where $R\ l\ a\ r = \langle l, a, Red, r\rangle$ and $B\ l\ a\ r = \langle l, a, Black, r\rangle$. The transformed version of this equation is

$$\begin{aligned}
&baliL\ (R\ (R\ t_1\ a_1\ \_\ t_2)\ a_2\ \_\ t_3)\ a_3\ t_4 = \\
&node\ (node\ t_1\ a_1\ Black\ t_2)\ a_2\ Red\ (node\ t_3\ a_3\ Black\ t_4)
\end{aligned}$$

where $R\ l\ a\ kp\ r = \langle l, a, (Red, kp), r\rangle$.

We obtained a priority search map based on red-black trees via the above transformations. The correctness proofs could be transformed incrementally, too. The main idea is to augment the data structure invariant to say that the annotations are correct as well. Function *invpst* expresses this property: $invpst\ \langle\rangle = True$ and

$$\begin{aligned}
&invpst\ \langle l, kp, (\_, kp'), r\rangle = \\
&(invpst\ l \wedge invpst\ r \wedge is\_min2\ kp'\ (set\ (inorder\ l\ @\ kp\ \#\ inorder\ r)))
\end{aligned}$$

where *set* (...) yields all key-priority bindings and *is_min2* asserts that kp' is minimal amongst them: $is\_min2\ kp'\ KP = (kp' \in KP \wedge (\forall kp{\in}KP.\ snd\ kp' \le snd\ kp))$.

It is straightforward to show $invpst\ (node\ l\ a\ c\ r) = (invpst\ l \wedge invpst\ r)$, and this easily discharges the additional proof obligations when transforming the correctness proof.

3.3 Related Work

Our priority map ADT is close to Hinze's [12] *priority search queue* interface, except that he also supports a few further operations that we could easily add but do not need for our applications. However, it is not clear if his implementation technique is the same as our priority search tree because his description employs a plethora of concepts, e.g. *priority search pennants*, *tournament trees*, *semi-heaps*, and multiple *views* of data types that obscure a direct comparison. We claim that at the very least our presentation is new because it is much simpler; we encourage the reader to compare the two.

As already observed by Hinze, McCreight's [19] priority search trees support range queries more efficiently than our trees. However, we can support the same range queries as Hinze efficiently, but that is outside the scope of the paper.

4 Undirected Graphs

There seems to be no single best way of how to represent undirected graphs in Isabelle/HOL. One variant is to represent an undirected graph as a symmetric relation, i.e., an entity of type $('v \times 'v)$ *set*. The advantage is that many existing theory of symmetric relations can be reused. However, edges in a symmetric relation are pairs, i.e., they are naturally directed: For $u{\neq}v$, we have $(u,v){\neq}(v,u)$, although, when interpreted as undirected edges, the two should be identified. The same issue transfers to derived concepts like paths in between two nodes.

Another option is to use undirected pairs or doubleton sets to represent an edge. The advantage is that HOL's equality on edges matches the natural equality. However, one cannot re-use the well-established theory of relations then, and has to develop many basic concepts from scratch.

For this formalization, we use a hybrid approach, which tries to combine the advantages of both, and being lightweight at the same time: A graph is represented as a symmetric relation, and only when equality on edges is required, they are converted to doubleton sets on the spot.

We start by defining the type of an undirected graph to be a finite, symmetric, and irreflexive relation together with a set of nodes, which must cover the domain of the relation:

$$\textbf{typedef}\ 'v\ \mathit{ugraph} = \{\ (V{::}'v\ \mathit{set}\ ,\ E).\ E \subseteq V{\times}V \wedge \mathit{finite}\ V \wedge \mathit{sym}\ E \wedge \mathit{irrefl}\ E\ \}$$

Next, we define accessor functions to obtain the nodes and edges of a graph:

$$\mathit{nodes}{::}'v\ \mathit{ugraph} \Rightarrow\ 'v\ \mathit{set} \qquad \mathit{edges}{::}'v\ \mathit{ugraph} \Rightarrow ('v \times 'v)\ \mathit{set}$$

We also define functions to construct a graph from its nodes and edges, to insert an edge into a graph, and to restrict the edges of a graph:

$$\begin{array}{l} \mathit{graph}{::}'v\ \mathit{set} \Rightarrow ('v \times 'v)\ \mathit{set} \Rightarrow\ 'v\ \mathit{ugraph} \\ \mathit{ins\_edge}{::}'v \times 'v \Rightarrow\ 'v\ \mathit{ugraph} \Rightarrow\ 'v\ \mathit{ugraph} \\ \mathit{restrict\_edges}{::}'v\ \mathit{ugraph} \Rightarrow ('v \times 'v)\ \mathit{set} \Rightarrow\ 'v\ \mathit{ugraph} \end{array}$$

Note that *graph* forms the symmetric closure of the edges, ignores reflexive edges, and adds missing nodes:

$$\begin{array}{l} \mathit{finite}\ V \wedge \mathit{finite}\ E \longrightarrow \\ \mathit{nodes}\ (\mathit{graph}\ V\ E) = V \cup \mathit{fst}\ `\ E \cup \mathit{snd}\ `\ E \wedge \mathit{edges}\ (\mathit{graph}\ V\ E) = E \cup E^{-1} - \mathit{Id} \end{array}$$

Similarly, *ins_edge* also inserts the nodes of the edge, and *restrict_edges* removes all edges not in the symmetric closure of the given set.

A *path* is a list of (directed) edges between two nodes:

$$\textit{path}\ g\ u\ []\ v = (u = v)$$
$$\textit{path}\ g\ u\ (e \# \textit{ps})\ w = (\exists\, v.\ e = (u,\ v) \wedge e \in \textit{edges}\ g \wedge \textit{path}\ g\ v\ \textit{ps}\ w)$$

As the edge relation is symmetric, every path induces a reversed path:

$$\textit{path}\ g\ u\ (\textit{revp}\ p)\ v = \textit{path}\ g\ v\ p\ u \qquad \text{where } \textit{revp}\ p = \textit{rev}\ (\textit{map}\ (\lambda(u,\ v).\ (v,\ u))\ p)$$

Obviously, existence of a path between two nodes is equivalent to these nodes being in the reflexive transitive closure of the edge relation:

$$(\exists\, p.\ \textit{path}\ g\ u\ p\ v) = ((u,\ v) \in (\textit{edges}\ g)^*) \qquad \text{where } \_^* \text{ is reflexive transitive closure}$$

We call a graph *connected*, if there exists path between all its nodes:

$$\textit{connected}\ g = (\textit{nodes}\ g \times \textit{nodes}\ g \subseteq (\textit{edges}\ g)^*)$$

A *simple path* does not contain any edge twice. Here, we need to consider undirected edges. Thus, we define a function $\textit{uedge}{::}'a \times {'a} \Rightarrow {'a}\ \textit{set}$ to map an edge to a doubleton set.

$$\textit{simple}\ p = \textit{distinct}\ (\textit{map}\ \textit{uedge}\ p) \qquad \text{where } \textit{uedge} = (\lambda(a,\ b).\ \{a,\ b\})$$

A *cycle* is a simple, non-empty path with the same start and end node. We define a predicate for *cycle-free* graphs:

$$\textit{cycle\_free}\ g = (\nexists\, p\ u.\ p \neq [] \wedge u \in \textit{nodes}\ g \wedge \textit{simple}\ p \wedge \textit{path}\ g\ u\ p\ u)$$

A *tree* is a connected and cycle free graph:

$$\textit{tree}\ g = (\textit{connected}\ g \wedge \textit{cycle\_free}\ g)$$

A *spanning tree* of a graph is a tree with the same nodes and a subset of the edges:

$$\textit{is\_spanning\_tree}\ G\ T = (\textit{tree}\ T \wedge \textit{nodes}\ T = \textit{nodes}\ G \wedge \textit{edges}\ T \subseteq \textit{edges}\ G)$$

Every connected graph has a spanning tree:

$$\textit{connected}\ g \longrightarrow (\exists\, t.\ \textit{is\_spanning\_tree}\ g\ t)$$

which is proved by removing edges on cycles until the graph is cycle free.

We model *weighted graphs* as graphs with a function $w{::}'v\ \textit{set} \Rightarrow \textit{nat}$ from (doubleton) sets to natural numbers[1]. Note that we do not need to restrict the domain of the weight function to be doubleton sets of valid nodes – the values for invalid nodes or sets will just be ignored. We then define the *weight* of a graph as the sum of the weights of all its edges:

$$\textit{weight}\ w\ g = \textit{sum}\ w\ (\textit{uedge}\ `\ \textit{edges}\ g)$$

A *minimum spanning tree* (MST) of a graph g is a spanning tree with minimal weight:

$$\textit{is\_MST}\ w\ g\ t =$$
$$(\textit{is\_spanning\_tree}\ g\ t \wedge (\forall\, t'.\ \textit{is\_spanning\_tree}\ g\ t' \longrightarrow \textit{weight}\ w\ t \le \textit{weight}\ w\ t'))$$

Obviously, each connected graph has a minimum spanning tree:

$$\textit{connected}\ g \longrightarrow (\exists\, t.\ \textit{is\_MST}\ w\ g\ t)$$

[1] For simplicity of presentation, we restrict weights to be natural numbers. Lifting this restriction is straightforward.

4.1 Related Work

There is a plethora of approaches to modelling graphs. Most formalizations focus on *directed* graphs. A typical example is Noschinski [24] (who should be consulted for a more detailed review of related work): he models undirected graphs as symmetric (bidirectional) directed graphs. Abstractly, this is also what Chou [5] does, who was the first to formalize undirected graphs. In both approaches, there is an explicit type of edges. Our approach is at the minimalist end: we avoid a separate edge type but use pairs of nodes. This means that we can use pattern matching on pairs in addition to projection functions but it also means that we cannot have multi-edges, something we don't need in our applications.

4.2 An Interface for Undirected Graphs

We have defined a representation of undirected weighted graphs in Isabelle/HOL. The next step towards an implementation is to specify an interface for the operations on undirected weighted graphs. We fix an implementation type $'g$, and an invariant $\mathit{invar}{::}'g \Rightarrow \mathit{bool}$, as well as two abstraction functions, one for the graph, and one for the weights:

$$\alpha g{::}'g \Rightarrow {}'v\ \mathit{ugraph} \qquad \alpha w{::}'g \Rightarrow {}'v\ \mathit{set} \Rightarrow \mathit{nat}$$

We specify operations to get the adjacent edges of a node, to create an empty graph, and to add an edge to a graph (implicitly adding the endpoints as nodes).

$$\mathit{adj}{::}'g \Rightarrow {}'v \Rightarrow ({}'v \times \mathit{nat})\ \mathit{list} \qquad \mathit{empty}{::}'g \qquad \mathit{add\_edge}{::}'v \times {}'v \Rightarrow \mathit{nat} \Rightarrow {}'g \Rightarrow {}'g$$

Note that these are exactly the operations required for our purpose of formalizing Prim's algorithm. More operations can easily be added. The specifications for the operations are:

$$\begin{aligned}&\mathit{invar}\ g \longrightarrow\\&\mathit{set}\ (\mathit{adj}\ g\ u) = \{(v,\ d) \mid (u,\ v) \in \mathit{edges}\ (\alpha g\ g) \wedge \alpha w\ g\ \{u,\ v\} = d\}\end{aligned}$$

$$\mathit{invar}\ \mathit{empty} \wedge \alpha g\ \mathit{empty} = \mathit{graph}\ \emptyset\ \emptyset \wedge \alpha w\ \mathit{empty} = (\lambda\_.\ 0)$$

$$\begin{aligned}&\mathit{invar}\ g \wedge (u,\ v) \notin \mathit{edges}\ (\alpha g\ g) \wedge u \neq v \longrightarrow\\&\mathit{invar}\ (\mathit{add\_edge}\ (u,\ v)\ d\ g) \wedge\\&\alpha g\ (\mathit{add\_edge}\ (u,\ v)\ d\ g) = \mathit{ins\_edge}\ (u,\ v)\ (\alpha g\ g) \wedge\\&\alpha w\ (\mathit{add\_edge}\ (u,\ v)\ d\ g) = (\alpha w\ g)(\{u,\ v\} := d)\end{aligned}$$

That is *adj g u* returns a list of pairs of nodes and weights, corresponding to the adjacent edges of node u, *empty* creates an empty graph, and *add_edge* inserts a (new) edge.

Note that designing interfaces often involves a trade-off between usability and implementability. We now motivate some of our design decisions:

- We leave the order of the adjacency list unspecified and allow duplicates. This introduces nondeterminism, and thus makes using the interface more complex. However, an (abstractly) fixed order on the adjacency list can only be implemented when the node type is linearly ordered, and even then, it incurs unnecessary overhead due to sorting.
- The node passed to *adj* needs not be a node of the graph (The returned list is empty for non-nodes). This makes the interface easier to use, as there is one precondition less to prove. Moreover, the implementation is straightforward.
- The weight function of the empty graph is fixed to $\lambda\_.\ 0$. This makes the specification deterministic, and thus simpler to use, and can be easily implemented.

4.3 Parsing Graphs from Lists

Based on the graph interface, we develop an algorithm to create a graph from a list of weighted edges. The elements of the list have the form $((u,\ v),\ d)$, describing an edge between u and v with weight d:

$$\textit{from\_list}\ l = \textit{foldr}\ (\lambda(e,\ d).\ \textit{add\_edge}\ e\ d)\ l\ \textit{empty}$$

We show that, for a valid list l, a graph implementation gi will be created that satisfies its invariant, and whose abstraction (g,w) contains exactly the nodes, edges, and weights contained in the list:

$$\begin{array}{l} \textit{G\_valid\_wgraph\_repr}\ l \longrightarrow \\ (\mathsf{let}\ gi = \textit{from\_list}\ l;\ g = \alpha g\ gi;\ w = \alpha w\ gi \\ \ \ \mathsf{in}\ \textit{invar}\ gi \wedge \textit{nodes}\ g = \bigcup\ \{\{u,\ v\}\ \mid \exists\, d.\ ((u,\ v),\ d) \in \textit{set}\ l\} \wedge \\ \qquad \textit{edges}\ g = \bigcup\ \{\{(u,\ v),\ (v,\ u)\}\ \mid \exists\, d.\ ((u,\ v),\ d) \in \textit{set}\ l\} \wedge \\ \qquad (\forall\,((u,\ v),\ d){\in}\textit{set}\ l.\ w\ \{u,\ v\} = d)) \end{array}$$

Here, a list is valid if no edge is specified twice and there are no reflexive edges:

$$\begin{array}{l} \textit{G\_valid\_wgraph\_repr}\ l = \\ ((\forall\,((u,\ v),\ d){\in}\textit{set}\ l.\ u \neq v) \wedge \textit{distinct}\ (\textit{map}\ (\lambda((u,\ v),\ d).\ \{u,\ v\})\ l)) \end{array}$$

5 Verifying Prim's Algorithm

Prim's algorithm [26] is a classical algorithm to find a minimum spanning tree of an undirected graph. In this section we describe our formalization of Prim's algorithm, roughly following the presentation of Cormen et al. [6].

Our approach features stepwise refinement. We start by a generic MST algorithm (Section 5.1) that covers both Prim's and Kruskal's algorithms. It maintains a subgraph A of an MST. Initially, A contains no edges and only the root node. In each iteration, the algorithm adds a new edge to A, maintaining the property that A is a subgraph of an MST. In a next refinement step, we only add edges that are adjacent to the current A, thus maintaining the invariant that A is always a tree (Section 5.2). Next, we show how to use a priority queue to efficiently determine a next edge to be added (Section 5.3), and implement the necessary update of the priority queue using a foreach-loop (Section 5.4). Finally we parameterize our algorithm over ADTs for graphs, maps, and priority queues (Section 5.5), instantiate these with actual data structures, and extract executable ML code (Section 5.6).

The advantage of this stepwise refinement approach is that the proof obligations of each step are mostly independent from the other steps. This modularization greatly helps to keep the proof manageable. Moreover, the steps also correspond to a natural split of the ideas behind Prim's algorithm: The same structuring is also done in the presentation of Cormen et al. [6], though not as detailed as ours.

5.1 Generic MST Algorithm

For the rest of this section, $g{::}'v\ \textit{ugraph}$ will be an undirected graph, $r \in \textit{nodes}\ g$ will be the root node identifying the connected component of the graph for which we want to compute the minimum spanning tree, and $w{::}'v\ \textit{set} \Rightarrow \textit{nat}$ will be a weight function.

Once we have fixed a root node, we can define the reachable part of the graph[2]:

$rg = \mathit{ins\_node}\ r\ (\mathit{restrict\_nodes}\ g\ ((\mathit{edges}\ g)^{*}\ ``\ \{r\}))$

Cormen et al. [6] describe Prim's algorithm as an instance of a more generic algorithm, which maintains a subgraph A of a minimum spanning tree. The graph is grown by repeatedly adding *safe edges*, i.e., edges that preserve the property of A being a subgraph of a minimum spanning tree.

$\mathit{is\_subset\_MST}\ w\ g\ A = (\exists\, t.\ \mathit{is\_MST}\ w\ g\ t \wedge A \subseteq \mathit{edges}\ t)$

Note that, like Cormen et al., we represent the current subgraph by a set of directed edges $A{::}('v \times 'v)\ \mathit{set}$.

The central idea of the generic algorithm provides a way to find a safe edge: A cut $(C,$ $\mathit{nodes}\ g - C)$ is a partitioning of the nodes. A subgraph *respects* a cut, if none of its edges cross the cut:

$\mathit{respects\_cut}\ A\ C = (A \subseteq C \times C \cup (-\ C) \times -\ C)$

An edge (u,v) is *light* w.r.t. a cut $(C,\ \mathit{nodes}\ g - C)$ if it *crosses* C (wlog. $u \in C \wedge v \notin C$), and its weight is minimal among all edges crossing C:

$\mathit{light\_edge}\ C\ u\ v =$
$(u \in C \wedge v \notin C \wedge (u,\ v) \in \mathit{edges}\ rg\ \wedge$
$(\forall\, (u',\ v') {\in} \mathit{edges}\ rg \cap C \times -\ C.\ w\ \{u,\ v\} \leq w\ \{u',\ v'\}))$

Given a cut that is respected by the current subgraph, light edges are safe:

$\mathit{is\_subset\_MST}\ w\ rg\ A \wedge \mathit{respects\_cut}\ A\ C \wedge \mathit{light\_edge}\ C\ u\ v \longrightarrow$
$\mathit{is\_subset\_MST}\ w\ rg\ (\{(v,\ u)\} \cup A)$

5.2 Prim's Algorithm

Prim's algorithm maintains a connected graph, i.e., A forms a tree. It starts with the singleton tree containing no edges and only the root node, and then repeatedly adds light edges connecting a node of the tree with a node not yet in the tree. Note that the nodes of the current tree can be defined from A as $S\ A = \{r\} \cup \mathit{fst}\ `\ A \cup \mathit{snd}\ `\ A$. Obviously, they form a cut respected by A: $\mathit{respects\_cut}\ A\ (S\ A)$.

Figure 1 shows the abstract algorithm in pseudocode: As long as the current nodes $S\ A$ are not closed under the edge relation, we pick a light edge (wrt. the cut $S\ A$), and add it to A. In order to prove this algorithm correct, we have to specify an invariant and a measure function, and show that the invariant holds initially, is preserved by a loop iteration, and implies that the result is a minimum spanning tree when the loop terminates. Moreover, we have to show that the measure decreases in every loop iteration.

As measure, we use the number of nodes that are *not* in $S\ A$:

$\mathit{T\_measure}1\ A = \mathit{card}\ (\mathit{nodes}\ rg - S\ A)$

The invariant states that A is a subgraph of a minimum spanning tree, and that all nodes of A are connected to the root node:

[2] Cormen at al. [6] assume that the graph is connected. Our setting is slightly more general. In particular, it saves us from checking a connectedness precondition.

```
$A := \{\}$
$\mathit{while}\ S\ A\ \mathit{not\ closed\ under\ edges}$
  $\mathit{choose\ edge}\ (u,v)\ \mathit{with}\ u \notin S\ A\ \mathit{and}\ v \in S\ A\ \mathit{such\ that}\ w\ \{u,v\}\ \mathit{is\ minimal}$
  $A := \{(u,v)\} \cup A$
```

Figure 1 Pseudocode of the abstract version of Prim's Algorithm.

$$\mathit{prim\_invar1}\ A = (\mathit{is\_subset\_MST}\ w\ rg\ A \wedge (\forall (u,\ v) \in A.\ (v,\ r) \in A^*))$$

The following theorems formalize invariant initialization, maintenance, and termination with the correct result for the abstract algorithm:

$$\mathit{prim\_invar1}\ \emptyset$$

$$\mathit{prim\_invar1}\ A \wedge \mathit{light\_edge}\ (S\ A)\ u\ v \longrightarrow$$
$$\mathit{prim\_invar1}\ (\{(v,\ u)\} \cup A) \wedge \mathit{T\_measure1}\ (\{(v,\ u)\} \cup A) < \mathit{T\_measure1}\ A$$

$$\mathit{prim\_invar1}\ A \wedge \mathit{edges}\ g \cap S\ A \times -\ S\ A = \emptyset \longrightarrow \mathit{is\_MST}\ w\ rg\ (\mathit{graph}\ \{r\}\ A)$$

Recall that *graph* forms the symmetric closure of the edges and adds missing nodes.

5.3 Using a Priority Queue

To efficiently find a next edge to be added, Prim's algorithm maintains a priority queue $Q{::}'v \Rightarrow \mathit{enat}$ and a predecessor map $\pi{::}'v \Rightarrow {}'v\ \mathit{option}$, where *enat* is the type of natural numbers with ∞ and $\mathit{enat} :: \mathit{nat} \Rightarrow \mathit{enat}$ is the canonical injection. A node u is *adjacent* to a set of nodes S, iff $u \notin S$ and there is an edge connecting u to some node in S.

For every node u that is adjacent to the current tree, $Q\ u$ stores the minimum weight of all edges connecting u to the tree. Moreover, $\pi\ u$ is the other endpoint of this edge. Additionally, we use π to store the edges of the current subtree itself: If $\pi\ u = \mathit{Some}\ v$, and $Q\ u = \infty$, i.e., the node u is already in the tree, then $(u,\ v)$ is an edge of the current subtree: $A\ Q\ \pi = \{(u,\ v) \mid \pi\ u = \mathit{Some}\ v \wedge Q\ u = \infty\}$.

Note that the implementation of Cormen et al. slightly differs from ours: Our priority queue only stores nodes that are *adjacent* to $S\ A$, while theirs stores *all* nodes not in $S\ A$. In their implementation, the priority queue has to be initialized with all (reachable) nodes of the graph, while we only need to initialize the queue for the root node. This simplifies the implementation as it saves an iteration over the graph's node.

A step of the algorithm extracts a node u from Q with minimum priority, and then updates the priorities for all adjacent nodes. The priority (and predecessor) of a node v' has to be updated if v' is adjacent to u, outside S, and the weight of the edge (u,v') is less than the weight currently stored in $Q\ v'$:

$$\mathit{upd\_cond}\ Q\ \pi\ u\ v' = ((v',\ u) \in \mathit{edges}\ g \wedge v' \notin S\ (A\ Q\ \pi) \wedge \mathit{enat}\ (w\ \{v',\ u\}) < Q\ v')$$

In this case, we update both, $Q\ v'$ and $\pi\ v'$:

$$\mathit{Qinter}\ Q\ \pi\ u\ v' = (\mathsf{if}\ \mathit{upd\_cond}\ Q\ \pi\ u\ v'\ \mathsf{then}\ \mathit{enat}\ (w\ \{v',\ u\})\ \mathsf{else}\ Q\ v')$$
$$Q'\ Q\ \pi\ u = (\mathit{Qinter}\ Q\ \pi\ u)(u := \infty)$$
$$\pi'\ Q\ \pi\ u\ v' = (\mathsf{if}\ \mathit{upd\_cond}\ Q\ \pi\ u\ v'\ \mathsf{then}\ \mathit{Some}\ u\ \mathsf{else}\ \pi\ v')$$

```
$Q := (\lambda\_.\ \infty)(r{:=}0);\ \pi := (\lambda\_.\ \mathit{None})$
$\mathit{while}\ Q \neq (\lambda\_.\ \infty)$
  $\mathit{pick}\ u\ \mathit{such\ that}\ Q\ u\ \mathit{is\ minimal}$
  $Q := Q'\ Q\ \pi\ u;\ \pi := \pi'\ Q\ \pi\ u$
  $Q\ u := \infty$
```

Figure 2 Pseudocode of Prim's Algorithm using a Priority Queue.

Note that we define Q' in two steps: Qinter is the priority queue where adjacent nodes have been updated, but the node u has not yet been removed. This definition is motivated by our implementation, which first iterates over the adjacent nodes, and then removes u from Q[3]. The refined algorithm is displayed in Figure 2.

Note that the refined algorithm starts with a priority queue that only contains the root node r. Intuitively, this corresponds to a tree that contains no nodes at all, not even the root node. In particular, it does not correspond to the initial state of the abstract algorithm. Only after the first loop iteration, the priority queue is initialized for the nodes adjacent to r. Thus, the refined state after the first iteration corresponds to the abstract initial state. We accommodate for this discrepancy in the invariant and variant of the refined loop:

$$\begin{aligned}&\mathit{prim\_invar2}\ Q\ \pi = (\mathit{prim\_invar2\_init}\ Q\ \pi \lor \mathit{prim\_invar2\_ctd}\ Q\ \pi)\\ &\mathit{T\_measure2}\ Q\ \pi = (\mathsf{if}\ Q\ r = \infty\ \mathsf{then}\ \mathit{T\_measure1}\ (A\ Q\ \pi)\ \mathsf{else}\ \mathit{card}\ (\mathit{nodes}\ \mathit{rg}))\end{aligned}$$

Here, $\mathit{prim\_invar2\_init}$ states that Q and π are in their initial states, and $\mathit{prim\_invar2\_ctd}$ states that the abstract invariant holds for the abstracted state $A\ Q\ \pi$, and, additionally, some consistency properties on Q and π:

▶ **Definition 1.** *Let (Q, π) be the refined state of the algorithm. Moreover, let A be the corresponding abstract state, S be the nodes of the current subtree, and $cE = \mathit{edges}\ \mathit{rg} \cap (-S) \times S$ be the set of edges crossing S. Then, $\mathit{prim\_invar2\_ctd}\ Q\ \pi$ states that*

1. *the abstract invariant holds: $\mathit{prim\_invar1}\ w\ g\ r\ A$*
2. *the root node has no predecessor and is not in Q: $\pi\ r = \mathit{None} \land Q\ r = \infty$*
3. *the outside node of any crossing edge is in Q: $\forall (u, v){\in}cE.\ Q\ u \neq \infty$*
4. *π encodes actual edges with target nodes in S:*
$$\forall\, u\ v.\ \pi\ u = \mathit{Some}\ v \longrightarrow v \in S \land (u, v) \in \mathit{edges}\ \mathit{rg}$$
5. *$Q\ u$ stores the weight of the corresponding edge in π, and this is the minimum weight of all crossing edges from u:*
$$\begin{aligned}&\forall\, u\ d.\ Q\ u = \mathit{enat}\ d \longrightarrow\\ &\qquad(\exists\, v.\ \pi\ u = \mathit{Some}\ v \land d = w\ \{u, v\} \land (\forall\, v'.\ (u, v') \in cE \longrightarrow d \leq w\ \{u, v'\}))\end{aligned}$$

Note that the first and second part of the invariant mutually exclude each other:

[3] Although non-standard, we chose this implementation because it slightly simplifies the proofs: When further refining the update, we can simply assume that the loop invariant holds. In the standard implementation, which removes u before the update, we would have to define an assertion that describes the state after the removal.

$\mathit{prim\_invar2\_init}\ Q\ \pi \longrightarrow \neg\ \mathit{prim\_invar2\_ctd}\ Q\ \pi$ Proof: Consider value of $Q\ r$.

Thus, the following lemmas imply correctness of the refined algorithm:

$$\mathit{prim\_invar2\_init}\ \mathit{initQ}\ \mathit{init}\pi$$

$$\begin{aligned}&\mathit{prim\_invar2\_init}\ Q\ \pi \wedge Q\ u = \mathit{enat}\ d \longrightarrow\\ &\mathit{prim\_invar2\_ctd}\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) \wedge\\ &\mathit{T\_measure2}\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) < \mathit{T\_measure2}\ Q\ \pi\end{aligned}$$

$$\begin{aligned}&\mathit{prim\_invar2\_ctd}\ Q\ \pi \wedge Q\ u = \mathit{enat}\ d \wedge (\forall v.\ \mathit{enat}\ d \le Q\ v) \longrightarrow\\ &\mathit{prim\_invar2\_ctd}\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) \wedge\\ &\mathit{T\_measure2}\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) < \mathit{T\_measure2}\ Q\ \pi\end{aligned}$$

$$\begin{aligned}&\mathit{prim\_invar2\_ctd}\ Q\ \pi \wedge Q = (\lambda\_.\ \infty) \longrightarrow\\ &\mathit{is\_MST}\ w\ \mathit{rg}\ (\mathit{graph}\ \{r\}\ \{(u,\ v) \mid \pi\ u = \mathit{Some}\ v\})\end{aligned}$$

5.4 Inner Foreach Loop

As a next step towards an efficiently executable implementation, we implement Q' and π' by iterating over the nodes adjacent to u. We assume that $\mathit{adjs}::('v \times \mathit{nat})\ \mathit{list}$ is the adjacency list of node u, and define:

```
$\mathit{foreach}\ u\ \mathit{adjs}\ (Q,\ \pi) =$
$\mathit{foldr}$
 $(\lambda(v,\ d)\ (Q,\ \pi).$
     $\mathsf{if}\ v \ne r \wedge (\pi\ v = \mathit{None} \vee Q\ v \ne \infty) \wedge \mathit{enat}\ d < Q\ v$
     $\mathsf{then}\ (Q(v := \mathit{enat}\ d),\ \pi(v \mapsto u))\ \mathsf{else}\ (Q,\ \pi))$
 $\mathit{adjs}\ (Q,\ \pi)$
```

where $f(x{\mapsto}y)$ is short for $f(x{:=}\mathit{Some}\ y)$. This updates Q and π only for adjacent nodes, with a smaller associated weight than that currently stored in Q. We show that this implementation computes the correct result:

$$\begin{aligned}&\mathit{set}\ \mathit{adjs} = \{(v,\ d) \mid (u,\ v) \in \mathit{edges}\ g \wedge w\ \{u,\ v\} = d\} \longrightarrow\\ &\mathit{foreach}\ u\ \mathit{adjs}\ (Q,\ \pi) = (\mathit{Qinter}\ Q\ \pi\ u,\ \pi'\ Q\ \pi\ u)\end{aligned}$$

In order to express Q and π after some but not all adjacent nodes have been processed, we need to generalize the statement accordingly. We define

$$\begin{aligned}&\mathit{Qigen}\ Q\ \pi\ u\ \mathit{adjs}\ v = (\mathsf{if}\ v \notin \mathit{fst}\ `\ \mathit{set}\ \mathit{adjs}\ \mathsf{then}\ Q\ v\ \mathsf{else}\ \mathit{Qinter}\ Q\ \pi\ u\ v)\\ &\pi'\mathit{gen}\ Q\ \pi\ u\ \mathit{adjs}\ v = (\mathsf{if}\ v \notin \mathit{fst}\ `\ \mathit{set}\ \mathit{adjs}\ \mathsf{then}\ \pi\ v\ \mathsf{else}\ \pi'\ Q\ \pi\ u\ v)\end{aligned}$$

and prove

$$\begin{aligned}&\mathit{set}\ \mathit{adjs} \subseteq \{(v,\ d) \mid (u,\ v) \in \mathit{edges}\ g \wedge w\ \{u,\ v\} = d\} \longrightarrow\\ &\mathit{foreach}\ u\ \mathit{adjs}\ (Q,\ \pi) = (\mathit{Qigen}\ Q\ \pi\ u\ \mathit{adjs},\ \pi'\mathit{gen}\ Q\ \pi\ u\ \mathit{adjs})\end{aligned}$$

by induction on the adjacency list adjs.

5.5 Data Structures

The next step towards an executable algorithm is to implement the graph, priority queue, and predecessor map by actual data structures. We do this in a two-step approach: First, we implement the algorithm parameterized over the interfaces of graphs, maps, and priority maps, and, in a second step, we instantiate these interfaces to actual data structures. This approach has two advantages: First, it is easy to exchange the used data structures by simply exchanging the instantiation. Second, not knowing the actual data structures when proving the implementation correct prevents accidental breaking of the interface and looking into data structure details. Note that this actually happens in practice, e.g., due to "forgotten" simplifier setup, or automated tools like sledgehammer, which do not know about interfaces.

For Prim's algorithm, we fix the interfaces of an undirected weighted graph (cf. Section 4.2), a map, and a priority map (cf. Section 3). The interface functions are prefixed with G, M, and Q, respectively, and the implementation types are $'g$, $'m$, and $'q$:

$$\begin{array}{lll}
G\_\alpha w :: 'g \Rightarrow 'v\ set \Rightarrow nat & G\_\alpha g :: 'g \Rightarrow 'v\ ugraph & G\_invar :: 'g \Rightarrow bool \\
G\_adj :: 'g \Rightarrow 'v \Rightarrow ('v \times nat)\ list & G\_empty :: 'g & \\
G\_add\_edge :: 'v \times 'v \Rightarrow nat \Rightarrow 'g \Rightarrow 'g & &
\end{array}$$

$$\begin{array}{lll}
M\_lookup :: 'm \Rightarrow 'v \Rightarrow 'v\ option & M\_invar :: 'm \Rightarrow bool & M\_empty :: 'm \\
M\_update :: 'v \Rightarrow 'v \Rightarrow 'm \Rightarrow 'm & M\_delete :: 'v \Rightarrow 'm \Rightarrow 'm &
\end{array}$$

$$\begin{array}{lll}
Q\_lookup :: 'q \Rightarrow 'v \Rightarrow nat\ option & Q\_invar :: 'q \Rightarrow bool & Q\_empty :: 'q \\
Q\_update :: 'v \Rightarrow nat \Rightarrow 'q \Rightarrow 'q & Q\_delete :: 'v \Rightarrow 'q \Rightarrow 'q & Q\_is\_empty :: 'q \Rightarrow bool \\
Q\_getmin :: 'q \Rightarrow 'v \times nat & &
\end{array}$$

For the rest of this section, we also fix a graph $g :: 'g$ and root node $r :: 'v$.

At this point of the formalization, we can actually define Prim's algorithm as a functional program. On the previous abstraction levels, this was not possible because functional programs in Isabelle/HOL must be deterministic. However, when, e.g., extracting a minimum element from a priority queue, we cannot define any tie-breaking in terms of the abstract representation $Q :: 'v \Rightarrow enat$, as the actual tie-breaking will depend on the data structure that is used. The same holds for the order in which the foreach loop iterates over the list of adjacent nodes. Figure 3 shows our implementation of the algorithm. It uses the *while* combinator, which obeys the following recursion equation:

$$while\ b\ c\ s = (\mathsf{if}\ b\ s\ \mathsf{then}\ while\ b\ c\ (c\ s)\ \mathsf{else}\ s)$$

In HOL, tail-recursive functions can always be defined, regardless of termination [2].

Like for the other abstraction levels, we define an invariant and a measure function:

$$\begin{array}{l}
prim\_invar\_impl\ Qi\ \pi i = \\
(Q\_invar\ Qi \wedge M\_invar\ \pi i \wedge prim\_invar2\ (Q\_\alpha\ Qi)\ (M\_lookup\ \pi i)) \\
T\_measure\_impl = (\lambda(Qi, \pi i).\ T\_measure2\ (Q\_\alpha\ Qi)\ (M\_lookup\ \pi i))
\end{array}$$

where Q_α abstracts the priority map to the type $'v \Rightarrow enat$, mapping *None* to ∞, and *M_lookup* abstracts the predecessor map to the type $'v \Rightarrow 'v\ option$.

Again, we show invariant initialization, maintenance, and termination with the correct result:

$$prim\_invar\_impl\ (Q\_update\ r\ 0\ Q\_empty)\ M\_empty$$

$$prim\_invar\_impl\ Qi\ \pi i \wedge \neg\ Q\_is\_empty\ Qi \wedge Q\_getmin\ Qi = (u,\ d)\ \wedge$$

$$\begin{aligned}
&\mathit{foreach\_impl}\ Qi\ \pi i\ u\ (G\_adj\ g\ u) = (Qi',\ \pi i') \longrightarrow\\
&\mathit{prim\_invar\_impl}\ (Q\_delete\ u\ Qi')\ \pi i' \wedge\\
&T\_measure\_impl\ (Q\_delete\ u\ Qi',\ \pi i') < T\_measure\_impl\ (Qi,\ \pi i)
\end{aligned}$$

$$\begin{aligned}
&Q\_is\_empty\ Q \wedge \mathit{prim\_invar\_impl}\ Q\ \pi \longrightarrow\\
&M\_invar\ \pi \wedge is\_MST\ (G\_\alpha w\ g)\ rg\ (graph\ \{r\}\ \{(u,\ v) \mid M\_lookup\ \pi\ u = Some\ v\})
\end{aligned}$$

Here, *foreach_impl* stands for the inner foreach loop:

$$\mathit{foreach\_impl}\ Qi\ \pi i\ u\ adjs = foldr\ (\mathit{foreach\_impl\_body}\ u)\ adjs\ (Qi,\ \pi i)$$

We show its correctness separately:

$$\begin{aligned}
&\mathit{foreach\_impl}\ Qi\ \pi i\ u\ (G\_adj\ g\ u) = (Qi',\ \pi i') \wedge \mathit{prim\_invar\_impl}\ Qi\ \pi i \longrightarrow\\
&Q\_invar\ Qi' \wedge M\_invar\ \pi i' \wedge Q\_\alpha\ Qi' = Qinter\ (Q\_\alpha\ Qi)\ (M\_lookup\ \pi i)\ u \wedge\\
&M\_lookup\ \pi i' = \pi'\ (Q\_\alpha\ Qi)\ (M\_lookup\ \pi i)\ u
\end{aligned}$$

This is proved by first showing that the abstract foreach loop *foreach* can simulate the concrete one, and then using the already proved correctness of the abstract loop.

Finally, we show correctness of the whole algorithm, i.e., that the returned predecessor map satisfies its invariant, and encodes a minimum spanning tree of the reachable part of the graph:

$$\begin{aligned}
&invar\_MST\ prim\_impl \wedge\\
&is\_MST\ (G\_\alpha w\ g)\ (component\_of\ (G\_\alpha g\ g)\ r)\\
&\ (graph\ \{r\}\ \{(u,\ v) \mid M\_lookup\ prim\_impl\ u = Some\ v\})
\end{aligned}$$

The proof is straightforward, using the standard invariant proof rule for while loops:

$$\begin{aligned}
&[\![P\ s;\ \bigwedge s.\ P\ s \wedge b\ s \longrightarrow P\ (c\ s);\ \bigwedge s.\ P\ s \wedge \neg\ b\ s \longrightarrow Q\ s;\ wf\ r;\\
&\ \bigwedge s.\ P\ s \wedge b\ s \longrightarrow (c\ s,\ s) \in r]\!]\\
&\Longrightarrow Q\ (while\ b\ c\ s)
\end{aligned}$$

5.6 Executable Code

Using Isabelle's locale mechanism, it is straightforward to instantiate the algorithm *prim_impl* to actual data structures implementing the interfaces. We do so by using red-black trees for both, the priority map and predecessor map. The graph is implemented by red-black trees, mapping nodes to their adjacency lists.

Finally, we combine the list parser *from_list* with *prim_impl*.

$$\begin{aligned}
&prim\_list\_impl\ l\ r =\\
&(\mathsf{if}\ G\_valid\_wgraph\_repr\ l\ \mathsf{then}\ Some\ (prim\_impl\ (G\_from\_list\ l)\ r)\ \mathsf{else}\ None)
\end{aligned}$$

We return *None* if the input list is not valid, otherwise we return a minimum spanning tree:

$$\begin{aligned}
&\mathsf{case}\ prim\_list\_impl\ l\ r\ \mathsf{of}\ None \Rightarrow \neg\ G\_valid\_wgraph\_repr\ l\\
&\mid Some\ \pi i \Rightarrow\\
&\quad \mathsf{let}\ g = \alpha g\ (from\_list\ l);\ w = \alpha w\ (from\_list\ l);\ rg = component\_of\ g\ r;\\
&\qquad t = graph\ \{r\}\ \{(u,\ v) \mid lookup\ \pi i\ u = Some\ v\}\\
&\quad \mathsf{in}\ G\_valid\_wgraph\_repr\ l \wedge invar\ \pi i \wedge is\_MST\ w\ rg\ t
\end{aligned}$$

Isabelle's code generator [11] can generate a functional program in various different target languages (SML, OCaml, Haskell, Scala) from *prim_list_impl*.

```
prim_impl = (let
  — Initialization
  (Q,π) = (Q_update r 0 Q_empty, M_empty);
  — Outer loop: Iterate until Q is empty
  (Q, π) =
  while (λ(Q, π). ¬ Q_is_empty Q) (λ(Q, π). let
    (u, _) = Q_getmin Q;
    — Inner loop: Update for adjacent nodes
    (Q, π) =
    foldr ((λ(v, d) (Q, π). let
        qv = Q_lookup Q v;
        πv = M_lookup π v
      in
        if v≠r ∧ (qv≠None ∨ πv=None) ∧ enat d < enat_of_option qv
        then (Q_update v d Q, M_update v u π) else (Q, π))
    ) (G_adj g u) (Q, π);
    Q = Q_delete u Q
    in (Q, π)) (Q, π)
  in π)
```

Figure 3 Implementation of Prim's algorithm, parameterized over a graph, map, and priority map interface.

5.7 Discussion and Related Work

We have used a stepwise refinement approach, from an abstract generic MST algorithm, over Prim's algorithm, to its implementation with a priority queue, and finally the realization of the priority queue with a concrete data structure and the extraction of executable code.

The abstract versions of the algorithm are inherently nondeterministic, which prevents their straightforward formalization in Isabelle/HOL. Instead, we manually came up with verification conditions (invariant maintenance) for the abstract level, and refined them towards the concrete level until we could use them to prove correct the concrete implementation. We expect that our approach of manual verification condition generation against informal algorithm sketches will not scale to more complex algorithms. To this end, the Isabelle Refinement Framework [18] provides a more scalable, though less lightweight, approach to stepwise refinement in Isabelle/HOL.

We are aware of two previous formal verifications of Prim's algorithm, but both of them ignore our focus, efficient data structures, and stop short of executable code. Abrial *et al.* [1] perform a stepwise refinement using the B event-based method. Guttmann [10] uses Isabelle/HOL to verify a version of Prim's algorithm in an extension of relation algebra.

6 Dijkstra's Algorithm

Dijkstra's algorithm [7] is a classical algorithm to determine the shortest paths from a root node to all other nodes in a weighted directed graph. Although it solves a different problem, and works on a different type of graphs, its structure is very similar to Prim's algorithm.

In particular, like Prim's algorithm, it has a simple loop structure and can be efficiently implemented by a priority queue. This makes Dijkstra's algorithm another good example to illustrate the main points of this proof pearl: Functional implementations of algorithms that use priority queues with a decrease-key operation.

A directed graph is represented by a weight function $w{::}'v \times {'v} \Rightarrow \mathit{enat}$. The edge relation is $\mathit{edges} = \{(u,\ v) \mid w\ (u,\ v) \neq \infty\}$.

Note that this formalization differs from our formalization of undirected graphs, in that we do not model an explicit node set, nor do we encode finiteness into the type. The modeling of an explicit node set has proved useful when formalizing the concept of trees[4], which is not required for Dijkstra's algorithm. As finiteness is the only additional property that we require, we traded the overhead of defining a new type for the overhead of maintaining finiteness as an explicit assumption.

6.1 Abstract Algorithm

Again, our formalization of Dijkstra's algorithm follows the presentation of Cormen et al. [6]. However, for the sake of simplicity, our algorithm does not compute actual shortest paths, but only their weights.

For the rest of this section, we fix a weighted directed graph w and a source node s. We define $\delta\ u\ v$ to be the minimum distance from node u to node v.

Abstractly, Dijkstra's algorithm keeps track of a set of finished nodes $S{::}'v\ \mathit{set}$, and an estimate of the shortest path weights $D{::}'v \Rightarrow \mathit{enat}$. The invariant $\mathit{D\_invar}\ D\ S$ states that

1. $D\ u$ is an upper bound of the minimum distance between s and u:

   $$\delta\ s\ u \leq D\ u$$

2. $D\ u$ is precise if u is finished:

   $$u \in S \longrightarrow D\ u = \delta\ s\ u$$

3. $D\ u$ is consistent with the distances induced by paths that end with an edge from a node $v{\in}S$:

   $$v \in S \longrightarrow D\ u \leq \delta\ s\ v + w\ (v,\ u)$$

4. The start node is finished, unless in the initial state

   $$s \in S \lor D = (\lambda\_.\ \infty)(s := 0) \land S = \emptyset$$

The main idea of the algorithm is that the least estimate $D\ u$ among all unfinished nodes $u{\notin}S$ is already precise:

$$u \notin S \land (\forall v.\ v \notin S \longrightarrow D\ u \leq D\ v) \longrightarrow D\ u = \delta\ s\ u$$

Thus, adding an unfinished node $u{\notin}S$ with minimal $D\ u$ to the finished set S, and updating the estimates of all successor nodes to accommodate for paths over u, will preserve the invariant. We iterate this until all nodes are finished, and thus all estimates are precise.

[4] For example, connecting two trees on disjoint nodes by a single edge yields a tree again. Without an explicit node set, the formulation of this lemma requires tedious special cases for singleton trees.

6.2 Refined Algorithm

Like for Prim's algorithm, a priority map $Q::'v \Rightarrow \mathit{nat\ option}$ from unfinished nodes to estimates is used to efficiently obtain a node with minimum estimate. Moreover, we use another map $V::'v \Rightarrow \mathit{nat\ option}$ to map finished nodes to their minimum distances from the source. The relation between a refined state (Q, V) and an abstract state (D,S) is defined as:

$$\begin{array}{l} \mathit{coupling}\ Q\ V\ D\ S = \\ (D = \mathit{enat\_of\_option} \circ V \mathbin{++} Q \land S = \mathit{dom}\ V \land \mathit{dom}\ V \cap \mathit{dom}\ Q = \emptyset) \end{array}$$

where $(++)$ joins two maps, and *enat_of_option* maps *None* to ∞. The refined loop invariant states that the refined state is related to an abstract state that satisfies its invariant:

$$\mathit{D\_invar}'\ Q\ V = (\exists\, D\ S.\ \mathit{coupling}\ Q\ V\ D\ S \land \mathit{D\_invar}\ D\ S)$$

The rest of the formalization proceeds analogously to the formalization of Prim's algorithm: We implement the update of the successor nodes by iteration over a successor list, refine the algorithm to use the interfaces of directed graphs, priority maps and maps, and finally instantiate it with concrete, red-black-tree based implementations. Combining the implementation with a *from_list* function for directed graphs, we get the executable function $\mathit{dijkstra\_list}::('v \times 'v \times \mathit{nat})\ \mathit{list} \Rightarrow 'v \Rightarrow ('v \times \mathit{nat},\ \mathit{color})\ \mathit{tree\ option}$ and the theorem

$$\begin{array}{l} \mathsf{case}\ \mathit{dijkstra\_list}\ l\ s\ \mathsf{of}\ \mathit{None} \Rightarrow \neg\ \mathit{valid\_graph\_rep}\ l \\ \mid \mathit{Some}\ D \Rightarrow \\ \quad \mathit{valid\_graph\_rep}\ l \land \mathit{invar}\ D \land \\ \quad (\forall\, u\ d.\ (\mathit{lookup}\ D\ u = \mathit{Some}\ d) = (\delta\ (\mathit{wgraph\_of\_list}\ l)\ s\ u = \mathit{enat}\ d)) \end{array}$$

Note that the distance δ is also parameterized over the graph here.

6.3 Related Work

Dijkstra's algorithm seems to be a standard benchmark for formal verification tools. One of the authors [23] has already verified Dijkstra's algorithm (including computation of shortest paths) using a functional priority queue based on Finger Trees [13], and later [15] amended the formalization to use an imperative heap data structure. Filliâtre [8] provides a verification in Why3 [9], Böhme *et al.* [3] provide one in Boogie, and Charguéraud [4] uses characteristic formulae to verify a Caml implementation. However, all of these do not verify priority queue data structures and only compute the distances, instead of actual shortest paths.

The treatment of priority queues differs in the above formalizations. Nordhoff and Lammich [23] use finger trees which support decrease-key. Charguéraud [4], however, writes:

> This implementation uses a priority queue that does not support the decrease-key operation. Using such a queue makes the proofs slightly more involved, because the invariants need to account for the fact that the queue may contain superseded values.

It appears that he uses the following trick that works for Dijkstra and Prim. Instead of decreasing the priority of some key k to p one adds the new pair (k,p) to the priority queue. If one also maintains a set of keys that have been extracted from the priority queue (which Dijkstra and Prim do anyway), one can simply ignore pairs (k,p) returned by *getmin*, if k has been extracted before. This trick requires that the same key is not inserted again after it has been extracted.

7 Conclusion

We presented priority search trees, a simple, purely functional and efficient data structure that combines search trees and priority queues, including an operation for modifying the priority associated with a key (aka "decrease-key"). We are only aware of considerably more complicated purely functional data structures with decrease-key, and the only one that has been formally verified is based on finger trees [13, 27, 22].

Based on priority search trees we gave the first verified executable implementation of Prim's algorithm. In particular we included the level of efficient data structures which had been ignored before. Therefore we show the details of the stepwise refinement and verification. We have also verified Dijkstra's algorithm in the same manner, but because of the ubiquity of this algorithm as a verification benchmark we merely sketched the proof.

References

1 Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal Derivation of Spanning Trees Algorithms. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pages 457–476. Springer, 2003. doi:10.1007/3-540-44880-2_27.

2 Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.

3 Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie — An Interactive Prover for the Boogie Program-Verifier. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 150–166, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

4 Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034828.

5 Ching-Tsun Chou. A Formal Theory of Undirected Graphs in Higher-Order Logic. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop, Valletta, Malta, September 19-22, 1994, Proceedings*, volume 859 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 1994. doi:10.1007/3-540-58450-1_40.

6 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2009.

7 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi:10.1007/BF01386390.

8 Jean-Christophe Filliâtre. Dijkstra's shortest path algorithm. From the Toccata gallery, http://toccata.lri.fr/gallery/dijkstra.en.html.

9 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, volume 7792. Springer, March 2013. URL: https://hal.inria.fr/hal-00789533.

10 Walter Guttmann. Relation-Algebraic Verification of Prim's Minimum Spanning Tree Algorithm. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 51–68, 2016. doi:10.1007/978-3-319-46750-4_4.

11 Florian Haftmann and Tobias Nipkow. Code Generation via Higher-Order Rewrite Systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

12 Ralf Hinze. A Simple Implementation Technique for Priority Search Queues. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 110–121. ACM, 2001. `doi:10.1145/507635.507650`.

13 Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

14 `http://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/RBT.html`.

15 Peter Lammich. Refinement based verification of imperative data structures. In Jeremy Avigad and Adam Chlipala, editors, *CPP 2016*, pages 27–36. ACM, 2016.

16 Peter Lammich and Tobias Nipkow. Priority Search Trees. *Archive of Formal Proofs*, 2019. Formal proof development. URL: `http://isa-afp.org/entries/Priority_Search_Trees.html`.

17 Peter Lammich and Tobias Nipkow. Purely Functional, Simple, and Efficient Implementation of Prim and Dijkstra. *Archive of Formal Proofs*, 2019. , Formal proof development. URL: `http://isa-afp.org/entries/Prim_Dijkstra_Simple.html`.

18 Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

19 Edward M. McCreight. Priority Search Trees. *SIAM J. Comput.*, 14(2):257–276, 1985. `doi:10.1137/0214021`.

20 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: `http://concrete-semantics.org`.

21 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

22 Benedikt Nordhoff, Stefan Körner, and Peter Lammich. Finger Trees. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *Archive of Formal Proofs*. `http://afp.sf.net/entries/Tree-Automata.shtml`, October 2010. Formal proof development.

23 Benedikt Nordhoff and Peter Lammich. Dijkstra's Shortest Path Algorithm. *Archive of Formal Proofs*, January 2012. , Formal proof development. URL: `http://isa-afp.org/entries/Dijkstra_Shortest_Path.html`.

24 Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015. `doi:10.1007/s11786-014-0183-z`.

25 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

26 R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, November 1957. `doi:10.1002/j.1538-7305.1957.tb01515.x`.

27 Matthieu Sozeau. Program-ing Finger Trees in Coq. *SIGPLAN Not.*, 42(9):13–24, October 2007. `doi:10.1145/1291220.1291156`.