Purely Functional, Simple, and Efficient Implementation of Prim and Dijkstra

Proof Pearl

Peter Lammich   Tobias Nipkow

September 9, 2019

Overview

Priority Search Trees

- simple, purely functional, support *decrease_key*
- easily added to any verified BST data structure

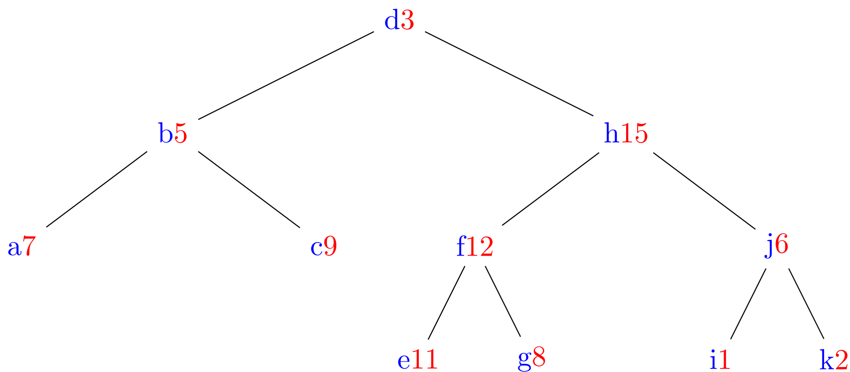
Lightweight verification of graph algorithms

- Prim (minimum spanning tree), Dijkstra (shortest Paths)
- purely functional, executable, efficient
- no heavy frameworks used

Essential proof technique: refinement

Priority Search Trees

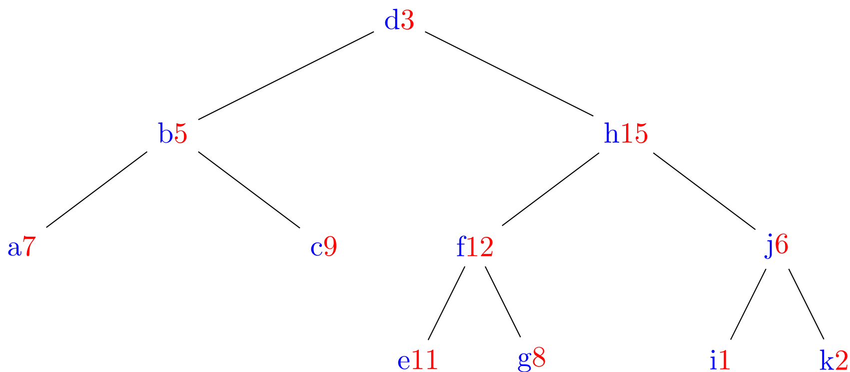
Binary Search Tree: Nodes contain key and priority



Priority Search Trees

Binary Search Tree: Nodes contain key and priority

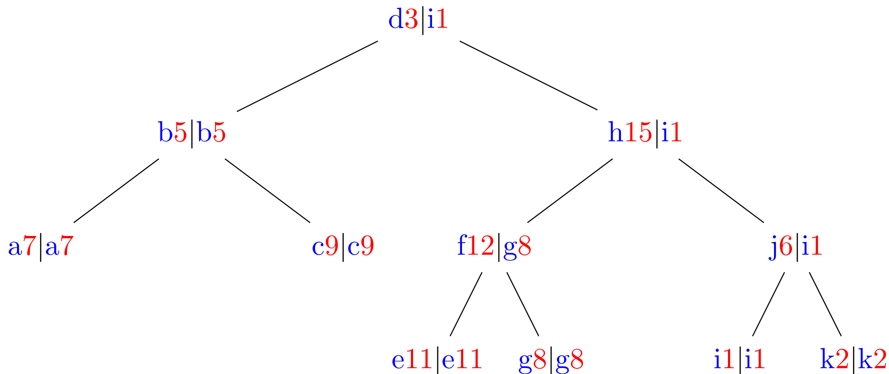
Idea: Annotate nodes with minimum priority kp in subtree.



Priority Search Trees

Binary Search Tree: Nodes contain key and priority

Idea: Annotate nodes with minimum priority kp in subtree.



Adapting Existing BST Implementation

Datatype

Old: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, \mathit{color}, \mathit{tree} \rangle$

New: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, k \times p, \mathit{color}, \mathit{tree} \rangle$

Adapting Existing BST Implementation

Datatype

Old: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, \mathit{color}, \mathit{tree} \rangle$

New: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, k \times p, \mathit{color}, \mathit{tree} \rangle$

Constructors

Old: $\langle l, kp, c, r \rangle$

New: $\mathit{mkNode}\ l\ kp\ c\ r$ where

$\mathit{mkNode}\ l\ kp\ c\ r = \langle l, kp, kp', c, r \rangle$
$kp' = \mathit{Min}\ (\{kp\} \cup \mathit{annot}\ l \cup \mathit{annot}\ r)$
$\mathit{annot}\ \mathit{Leaf} = \{\} \quad \mid \quad \mathit{annot}\ \langle _, _, kp', _, _ \rangle = kp'$

Adapting Existing BST Implementation

Datatype

Old: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, \mathit{color}, \mathit{tree} \rangle$

New: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, k \times p, \mathit{color}, \mathit{tree} \rangle$

Constructors

Old: $\langle l, kp, c, r \rangle$

New: $\mathit{mkNode}\ l\ kp\ c\ r$ where

$\mathit{mkNode}\ l\ kp\ c\ r = \langle l, kp, kp', c, r \rangle$

$kp' = \mathit{Min}\ (\{kp\} \cup \mathit{annot}\ l \cup \mathit{annot}\ r)$

$\mathit{annot}\ \mathit{Leaf} = \{\} \quad \mid \quad \mathit{annot}\ \langle _, _, kp', _, _ \rangle = kp'$

Destructors

Old: $\mathit{case}\ x\ \mathit{of}\ \langle l, kp, c, r \rangle \Rightarrow \ldots$

New: $\mathit{case}\ x\ \mathit{of}\ \langle l, kp, _, c, r \rangle \Rightarrow \ldots$

Adapting Existing BST Implementation

Datatype

Old: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, \mathit{color}, \mathit{tree} \rangle$

New: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, k \times p, \mathit{color}, \mathit{tree} \rangle$

Constructors

Old: $\langle l, kp, c, r \rangle$

New: $\mathit{mkNode}\ l\ kp\ c\ r$ where

$$\mathit{mkNode}\ l\ kp\ c\ r = \langle l, kp, kp', c, r \rangle$$
$$kp' = \mathit{Min}\ (\{kp\} \cup \mathit{annot}\ l \cup \mathit{annot}\ r)$$
$$\mathit{annot}\ \mathit{Leaf} = \{\} \quad \mid \quad \mathit{annot}\ \langle _, _, kp', _, _ \rangle = kp'$$

Destructors

Old: $\mathit{case}\ x\ \mathit{of}\ \langle l, kp, c, r \rangle \Rightarrow \ldots$

New: $\mathit{case}\ x\ \mathit{of}\ \langle l, kp, _, c, r \rangle \Rightarrow \ldots$

Maintenance overhead: $O(\mathit{t_compare})$ (typically $O(1)$)

Adapting Existing BST Implementation

Datatype

Old: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, \mathit{color}, \mathit{tree} \rangle$

New: $\mathit{tree} = \mathit{Leaf} \mid \langle \mathit{tree}, k \times p, k \times p, \mathit{color}, \mathit{tree} \rangle$

Constructors

Old: $\langle l, kp, c, r \rangle$

New: $\mathit{mkNode}\ l\ kp\ c\ r$ where

$\mathit{mkNode}\ l\ kp\ c\ r = \langle l, kp, kp', c, r \rangle$
$kp' = \mathit{Min}\ (\{kp\} \cup \mathit{annot}\ l \cup \mathit{annot}\ r)$
$\mathit{annot}\ \mathit{Leaf} = \{\} \quad \mid \quad \mathit{annot}\ \langle _, _, kp', _, _ \rangle = kp'$

Destructors

Old: $\mathit{case}\ x\ \mathit{of}\ \langle l, kp, c, r \rangle \Rightarrow \ldots$

New: $\mathit{case}\ x\ \mathit{of}\ \langle l, kp, _, c, r \rangle \Rightarrow \ldots$

Maintenance overhead: $O(t_compare)$ (typically $O(1)$)

Adaptation of existing proofs: straightforward

Stepwise Refinement Approach

- Level 1 **Abstract MST algorithm**
- Level 2 Implementation with priority queue
- Level 3 Parameterize over interfaces
- Level 4 Instantiate with data structures
- Level 5 Generate ML code

Setting the Scene

Fix g, w, r

$nodes\ g :: V\ set$
$edges\ g \subseteq V \times V$ irreflexive, symmetric
$w\ (u,v) :: nat_\infty$ weight of edge between u and v
$r \in nodes\ g$ root node

Definitions

$$is_MST\ w\ g\ t$$
$$= is_spanning_tree\ g\ t$$
$$\wedge\ (\forall\ t'.\ is_spanning_tree\ g\ t' \longrightarrow weight\ w\ t \leq weight\ w\ t')$$

$$is_spanning_tree\ g\ t = tree\ t \wedge nodes\ t = nodes\ g \wedge edges\ t \subseteq edges\ g$$

...

Abstract MST Algorithm

Grow tree A by adding *light edges* until spanning tree reached.

Abstract MST Algorithm

Grow tree A by adding *light edges* until spanning tree reached.

$A := \{\}$
while $\exists$ u v. *light_edge* $(S\ A)$ u v
  choose u v *with* *light_edge* $(S\ A)$ u v
  $A := \{(u,v)\} \cup A$

return A

where $S\ A = \{r\} \cup fst`A \cup snd`A$
and *light_edge* $C\ u\ v$ = (u,v) is minimal weight edge crossing C

Abstract MST Algorithm

Grow tree A by adding *light edges* until spanning tree reached.

A := {}
while $\exists$ *u v. light_edge* (*S A*) *u v*
  choose u v with light_edge (*S A*) *u v*
  A := {(*u*,*v*)} $\cup$ *A*

return A

where $S\ A = \{r\} \cup fst`A \cup snd`A$
and *light_edge C u v* = (*u*,*v*) is minimal weight edge crossing *C*

Invariant: $I_1\ A = A$ is subtree of an MST

VCs: invariant init, maintenance, yields MST on termination

Abstract MST Algorithm

Grow tree A by adding *light edges* until spanning tree reached.

$A := \{\}$
while $\exists$ *u v. light_edge* (*S A*) *u v*
  choose u v with light_edge (*S A*) *u v*
  $A := \{(u,v)\} \cup A$

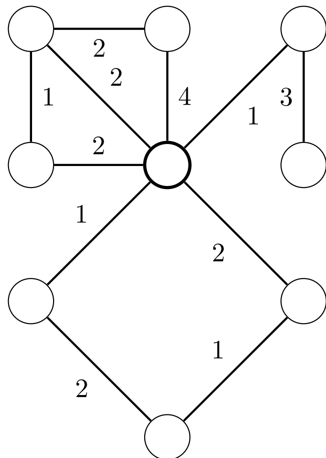
return A

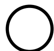
where $S\ A = \{r\} \cup fst`A \cup snd`A$
and *light_edge C u v* = (*u*,*v*) is minimal weight edge crossing *C*

Invariant: $I_1\ A = A$ is subtree of an MST

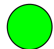
VCs: invariant init, maintenance, yields MST on termination
  in paper: termination proof

Example (Abstract Algorithm)





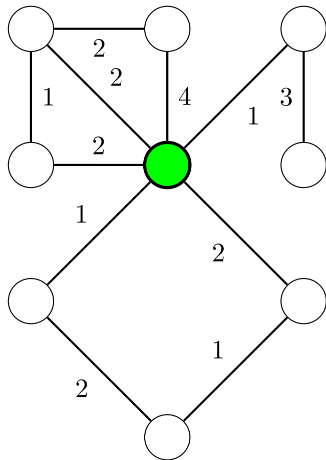
root node

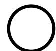


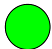
node in S A

edge in A

Example (Abstract Algorithm)

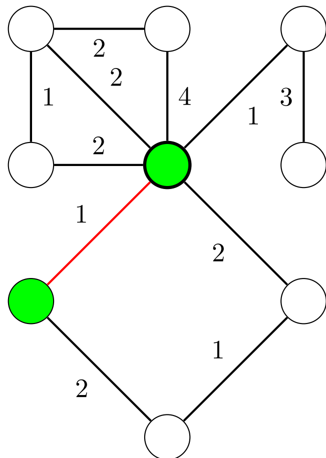


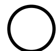

root node

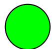

node in S A

edge in A

Example (Abstract Algorithm)

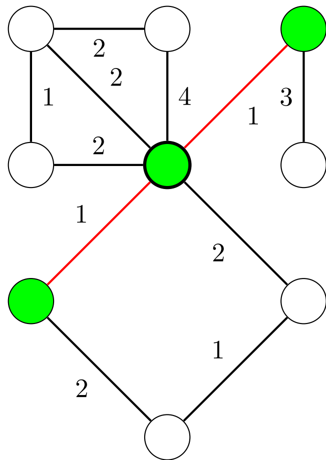


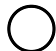

root node

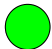

node in S A

edge in A

Example (Abstract Algorithm)

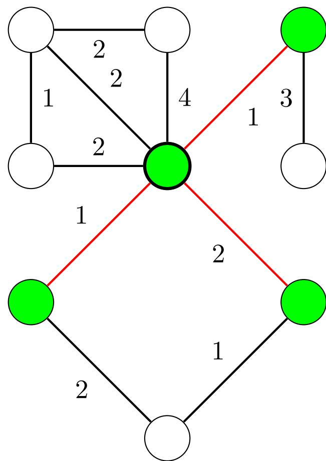


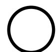
 root node

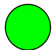
 node in S A

edge in A

Example (Abstract Algorithm)

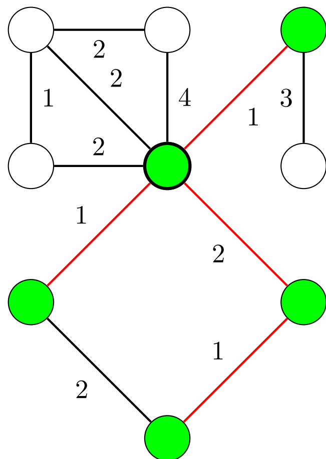


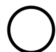

root node

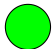

node in S A

edge in A

Example (Abstract Algorithm)

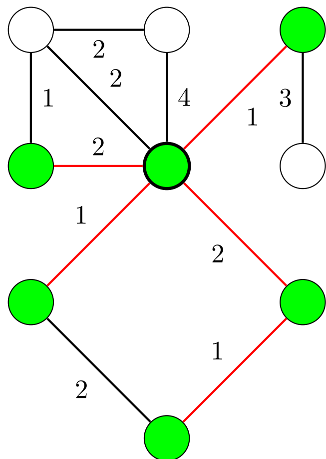


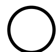
 root node

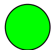
 node in S A

edge in A

Example (Abstract Algorithm)

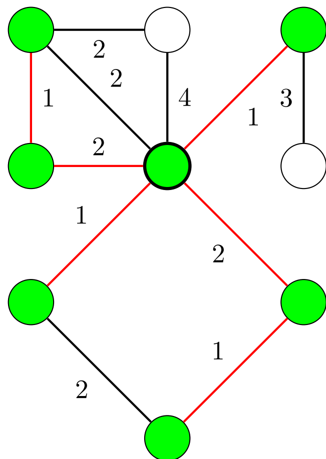


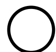

root node

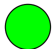

node in S A

edge in A

Example (Abstract Algorithm)



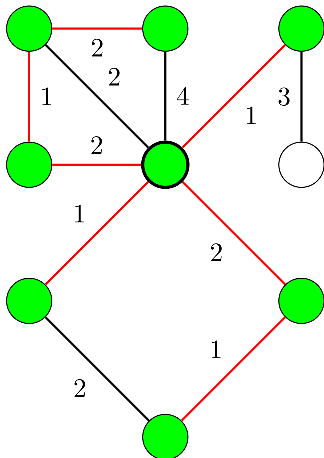
 root node

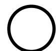
 node in S A

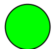
edge in A

Example (Abstract Algorithm)

Level 1

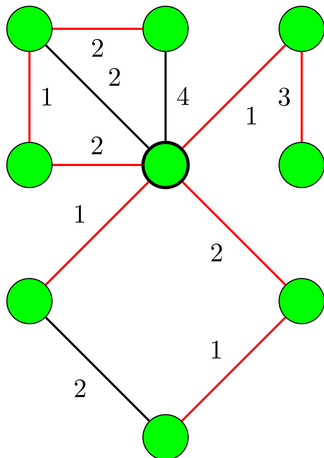


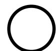
 root node

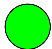
 node in S A

edge in A

Example (Abstract Algorithm)



 root node

 node in S A

edge in A

Stepwise Refinement Approach

- Level 1 Abstract MST algorithm
- Level 2 **Implementation with priority queue**
- Level 3 Parameterize over interfaces
- Level 4 Instantiate with data structures
- Level 5 Generate ML code

Prim's Algorithm: Use Priority Queue

$Q\ u = \mathit{Some}\ d$: d is minimal weight that connects u to current tree

$\pi\ u = \mathit{Some}\ v$: (u,v) is minimal connecting edge

If u not in Q: (u,v) is edge of current tree

Prim's Algorithm: Use Priority Queue

$Q\ u = Some\ d$: d is minimal weight that connects u to current tree

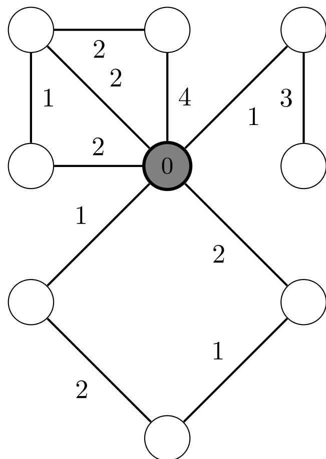
$\pi\ u = Some\ v$: (u,v) is minimal connecting edge

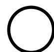
If u not in Q: (u,v) is edge of current tree

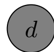
Abstraction: $A\ Q\ \pi = \{(u,\ v).\ \pi\ u = Some\ v \wedge\ Q\ u = \infty\}$

Corresponds to A in abstract algorithm

Example (Prim's Algorithm)

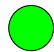



root node

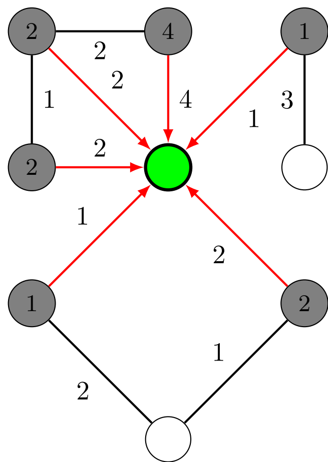


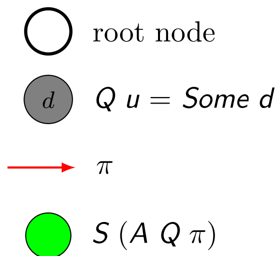
$Q\ u = Some\ d$

π

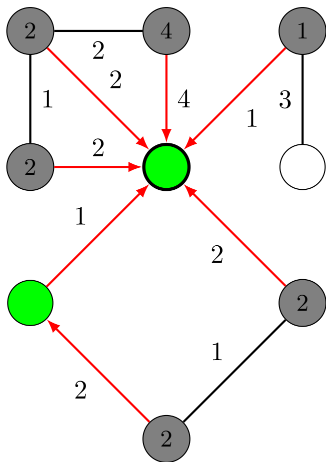

$S\ (A\ Q\ \pi)$

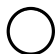
Example (Prim's Algorithm)

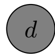




Example (Prim's Algorithm)

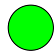



root node

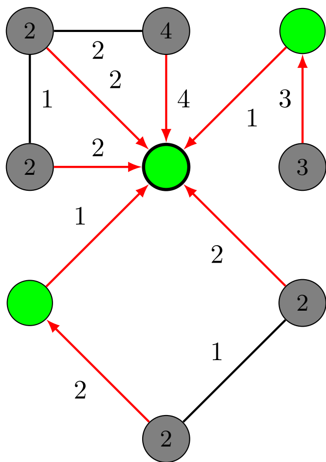


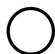
$Q\ u = Some\ d$

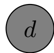
π


$S\ (A\ Q\ \pi)$

Example (Prim's Algorithm)

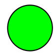



root node

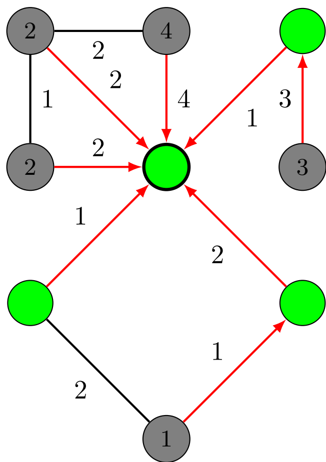


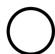
$Q\ u = Some\ d$

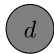
π


$S\ (A\ Q\ \pi)$

Example (Prim's Algorithm)

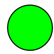



root node

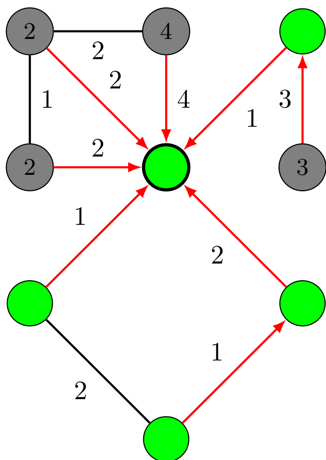


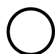
$Q\ u = Some\ d$

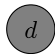
π


$S\ (A\ Q\ \pi)$

Example (Prim's Algorithm)

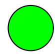



root node

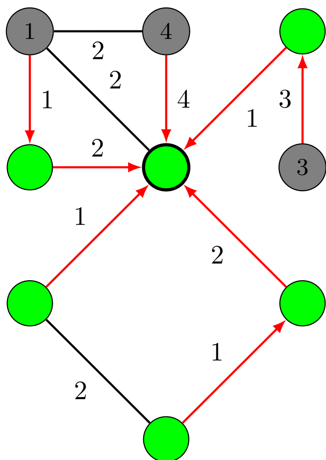


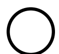
$Q\ u = Some\ d$

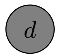
π


$S\ (A\ Q\ \pi)$

Example (Prim's Algorithm)

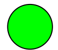



root node

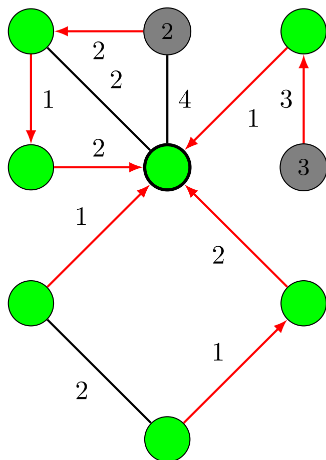


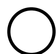
$Q\ u = Some\ d$

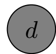
π


$S\ (A\ Q\ \pi)$

Example (Prim's Algorithm)

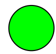



root node

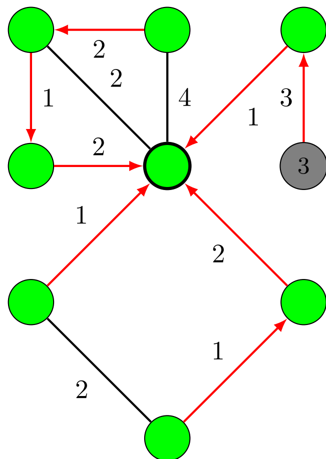


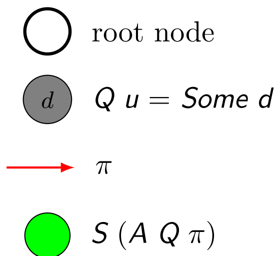
$Q\ u = Some\ d$

π

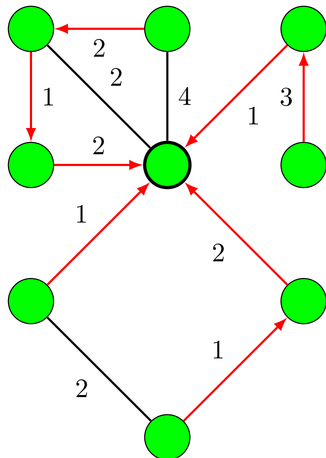

$S\ (A\ Q\ \pi)$

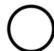
Example (Prim's Algorithm)

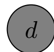




Example (Prim's Algorithm)

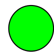



root node



$Q\ u = Some\ d$

π


$S\ (A\ Q\ \pi)$

Proving Implementation Correct

Abstraction: $A\ Q\ \pi = \{(u, v).\ \pi\ u = \mathit{Some}\ v \wedge Q\ u = \infty\}$

Proving Implementation Correct

Abstraction: $A\ Q\ \pi = \{(u, v).\ \pi\ u = Some\ v \wedge Q\ u = \infty\}$

Invariant: $I_2\ Q\ \pi = I_1\ (A\ Q\ \pi) \wedge$ consistency of Q, π

Proving Implementation Correct

Abstraction: $A\ Q\ \pi = \{(u, v).\ \pi\ u = \mathit{Some}\ v \wedge Q\ u = \infty\}$

Invariant: $I_2\ Q\ \pi = I_1\ (A\ Q\ \pi) \wedge$ consistency of Q, π

Proving VCs

- consider effect of operations on abstraction
- use VCs from abstract algorithm
- prove consistency separately

Proving Implementation Correct

Abstraction: $A\ Q\ \pi = \{(u,\ v).\ \pi\ u = Some\ v \wedge Q\ u = \infty\}$

Invariant: $I_2\ Q\ \pi = I_1\ (A\ Q\ \pi) \wedge$ consistency of Q, π

Proving VCs

- consider effect of operations on abstraction
- use VCs from abstract algorithm
- prove consistency separately

Example (invariant maintenance)

show that loop iteration adds light edge to $A\ Q\ \pi$

$$\begin{array}{l} I_2\ Q\ \pi \wedge Q\ u\ minimal \Longrightarrow \\ \quad \exists\, v.\ \pi\ u = Some\ v \wedge light_edge\ (S\ (A\ Q\ \pi))\ v\ u \\ \qquad \wedge\ A\ Q'\ \pi' = \{(u{,}v)\} \cup A\ Q\ \pi \end{array}$$

use VCs from abstract algorithm to get $I_1\ (A\ Q'\ \pi')$

Initialization Round

First iteration of loop: initialize π for root node.

Does not correspond to abstract loop.

Initialization Round

First iteration of loop: initialize π for root node.

Does not correspond to abstract loop.

Complete invariant $I_2\ Q\ \pi =$

$$\text{in first iteration } (\pi = (\lambda_.\ \mathit{None}) \land Q = [r \mapsto 0])$$
$$\lor\ I_1\ (A\ Q\ \pi) \land \text{consistency of } Q, \pi$$

Stepwise Refinement Approach

- Level 1 Abstract MST algorithm
- Level 2 Implementation with priority queue
- Level 3 **Parameterize over interfaces**
- Level 4 Instantiate with data structures
- Level 5 Generate ML code

Priority Map Interface

Invariant and abstraction function

$\mathit{invar} :: {'m} \Rightarrow \mathit{bool}$ and $\alpha :: {'m} \Rightarrow {'a} \Rightarrow {'b}{::}\mathit{linorder\ option}$

Priority Map Interface

Invariant and abstraction function

$invar :: {'m} \Rightarrow bool$ and $\alpha :: {'m} \Rightarrow {'a} \Rightarrow {'b}::linorder\ option$

Operations

$empty :: {'m}$
$update :: {'a} \Rightarrow {'b} \Rightarrow {'m} \Rightarrow {'m}$ — subsumes *decrease_key*
$delete :: {'a} \Rightarrow {'m} \Rightarrow {'m}$
$is_empty :: {'m} \Rightarrow bool$
$lookup :: {'m} \Rightarrow {'a} \Rightarrow {'b}\ option$
$getmin :: {'m} \Rightarrow {'a} \times {'b}$

Priority Map Interface

Invariant and abstraction function

$invar :: {}'m \Rightarrow bool$ and $\alpha :: {}'m \Rightarrow {}'a \Rightarrow {}'b{::}linorder\ option$

Operations

$empty :: {}'m$
$update :: {}'a \Rightarrow {}'b \Rightarrow {}'m \Rightarrow {}'m$ — subsumes *decrease_key*
$delete :: {}'a \Rightarrow {}'m \Rightarrow {}'m$
$is_empty :: {}'m \Rightarrow bool$
$lookup :: {}'m \Rightarrow {}'a \Rightarrow {}'b\ option$
$getmin :: {}'m \Rightarrow {}'a \times {}'b$

Properties

$invar\ m \Longrightarrow \alpha\ (update\ a\ b\ m) = (\alpha\ m)(a \mapsto b)$
$getmin\ m = (k,\ p) \wedge invar\ m \wedge \alpha\ m \neq Map.empty$
$\quad \Longrightarrow \alpha\ m\ k = Some\ p \wedge (\forall\ p' {\in} ran\ (\alpha\ m).\ p \leq p')$
...

Parameterized Algorithm

```
prim_impl = (let
  (Q, π) = (Q_update r 0 Q_empty, M_empty);
  (Q, π) =
  while (λ(Q, π). ¬ Q_is_empty Q) (λ(Q, π). let
    (u, _) = Q_getmin Q;
    (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) ∧ 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)
```

G_f, *Q_f*, *M_f*: interface operations for graph, PM, map

Correctness Proof

$$I_3\ Qi\ \pi i = I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i) \wedge Q_invar\ Qi \wedge M_invar\ \pi i$$

Correctness Proof

$I_3\ Qi\ \pi i = I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i) \wedge Q_invar\ Qi \wedge M_invar\ \pi i$

Proving $I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i)$

1. use interface properties, e.g.
   $M_\alpha\ (M_update\ v\ u\ \pi) = (M_\alpha\ \pi)(v{:=}Some\ u)$
2. then use lemmas from Level 2

Correctness Proof

$I_3\ Qi\ \pi i = I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i) \wedge Q_invar\ Qi \wedge M_invar\ \pi i$

Proving $I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i)$

1. use interface properties, e.g.
   $M_\alpha\ (M_update\ v\ u\ \pi) = (M_\alpha\ \pi)(v{:=}Some\ u)$
2. then use lemmas from Level 2

Proving data structure invariants (e.g. $M_invar\ (M_update\ v\ u\ \pi)$)

straightforward by interface properties

Correctness Proof

$I_3\ Qi\ \pi i = I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i) \wedge Q_invar\ Qi \wedge M_invar\ \pi i$

Proving $I_2\ (Q_\alpha\ Qi)\ (M_\alpha\ \pi i)$

1. use interface properties, e.g.
   $M_\alpha\ (M_update\ v\ u\ \pi) = (M_\alpha\ \pi)(v{:=}Some\ u)$
2. then use lemmas from Level 2

Proving data structure invariants (e.g. $M_invar\ (M_update\ v\ u\ \pi)$)

straightforward by interface properties

Final correctness theorem

$M_invar\ prim_impl$
$\wedge\ is_MST\ (G_\alpha w\ g)\ g'\ (MST_of\ (M_\alpha\ prim_impl))$

where $g' = component_of\ (G_\alpha\ g)\ r$
and $MST_of\ \pi = graph\ \{r\}\ \{(u, v)\ | u\ v.\ \pi\ u = Some\ v\}$

Stepwise Refinement Approach

Level 1 Abstract MST algorithm

Level 2 Implementation with priority queue

Level 3 Parameterize over interfaces

Level 4 **Instantiate with data structures**

Level 5 **Generate ML code**

Instantiation and Code Generation

Instantiate interfaces with actual data structures

- adjacency maps, priority search trees, red-black trees
- locales: instantiated correctness theorem for free

Instantiation and Code Generation

Instantiate interfaces with actual data structures

- adjacency maps, priority search trees, red-black trees
- locales: instantiated correctness theorem for free

Use code generator to export Standard-ML code for *prim_impl*

Conclusions

Priority Search Trees

- simple, purely functional, efficient
- support decrease-key
- implemented (and verified) easily on top of existing BST

Verification of Prim's Algorithm

- lightweight refinement approach
- down to executable functional code

In the paper

- interface for undirected graphs
- Dijkstra's algorithm