

Formalizing the Edmonds-Karp Algorithm

Peter Lammich and S. Reza Sefidgar

May 14, 2024

Abstract

We present a formalization of the Edmonds-Karp algorithm for computing the maximum flow in a network. Our formal proof closely follows a standard textbook proof, and is accessible even without being an expert in Isabelle/HOL— the interactive theorem prover used for the formalization. We use stepwise refinement to refine a generic formulation of the Ford-Fulkerson method to Edmonds-Karp algorithm, and formally prove its complexity bound of $O(VE^2)$.

Further refinement yields a verified implementation, whose execution time compares well to an unverified reference implementation in Java.

This entry is based on our ITP-2016 paper with the same title.

Contents

1	Introduction	3
2	The Ford-Fulkerson Method	3
2.1	Algorithm	3
2.2	Partial Correctness	4
2.3	Algorithm without Assertions	5
3	Edmonds-Karp Algorithm	6
3.1	Complexity and Termination Analysis	6
3.2	Algorithm	9
3.2.1	Total Correctness	10
3.2.2	Complexity Analysis	11
4	Breadth First Search	12
4.1	Algorithm	13
4.2	Correctness Proof	15
4.3	Extraction of Result Path	18
4.4	Inserting inner Loop and Successor Function	19
4.5	Imperative Implementation	22
5	Implementation of the Edmonds-Karp Algorithm	23
5.1	Refinement to Residual Graph	23
5.1.1	Refinement of Operations	23
5.2	Implementation of Bottleneck Computation and Augmentation	25
5.3	Refinement to use BFS	28
5.4	Implementing the Successor Function for BFS	29
5.5	Adding Tabulation of Input	30
5.6	Imperative Implementation	31
5.6.1	Implementation of Adjacency Map by Array	32
5.6.2	Implementation of Capacity Matrix by Array	33
5.6.3	Representing Result Flow as Residual Graph	34
5.6.4	Implementation of Functions	34
5.7	Correctness Theorem for Implementation	37
6	Combination with Network Checker	38
6.1	Adding Statistic Counters	38
6.2	Combined Algorithm	39
6.3	Usage Example: Computing Maxflow Value	39
7	Conclusion	42
7.1	Related Work	44
7.2	Future Work	45

1 Introduction

Computing the maximum flow of a network is an important problem in graph theory. Many other problems, like maximum-bipartite-matching, edge-disjoint-paths, circulation-demand, as well as various scheduling and resource allocating problems can be reduced to it. The Ford-Fulkerson method [8] describes a class of algorithms to solve the maximum flow problem. An important instance is the Edmonds-Karp algorithm [7], which was one of the first algorithms to solve the maximum flow problem in polynomial time for the general case of networks with real valued capacities.

In our paper [16], we present a formal verification of the Edmonds-Karp algorithm and its polynomial complexity bound. The formalization is conducted entirely in the Isabelle/HOL proof assistant [21]. This entry contains the complete formalization. Stepwise refinement techniques [25, 1, 2] allow us to elegantly structure our verification into an abstract proof of the Ford-Fulkerson method, its instantiation to the Edmonds-Karp algorithm, and finally an efficient implementation. The abstract parts of our verification closely follow the textbook presentation of Cormen et al. [5]. We have used the Isar [24] proof language to develop human-readable proofs that are accessible even to non-Isabelle experts.

While there exists another formalization of the Ford-Fulkerson method in Mizar [18], we are, to the best of our knowledge, the first that verify a polynomial maximum flow algorithm, prove the polynomial complexity bound, or provide a verified executable implementation. Moreover, this entry is a case study on elegantly formalizing algorithms.

2 The Ford-Fulkerson Method

```
theory FordFulkerson-Algo
imports
  Flow-Networks.Ford-Fulkerson
  Flow-Networks.Refine-Add-Fofu
begin
```

In this theory, we formalize the abstract Ford-Fulkerson method, which is independent of how an augmenting path is chosen

```
context Network
begin
```

2.1 Algorithm

We abstractly specify the procedure for finding an augmenting path: Assuming a valid flow, the procedure must return an augmenting path iff there exists one.

definition *find-augmenting-spec* $f \equiv do \{$
 assert (*NFlow* $c\ s\ t\ f$);
 select $p.$ *NPreflow.isAugmentingPath* $c\ s\ t\ f\ p$
 $\}$

Moreover, we specify augmentation of a flow along a path

definition (in *NFlow*) *augment-with-path* $p \equiv augment\ (augmentingFlow\ p)$

We also specify the loop invariant, and annotate it to the loop.

abbreviation *fofu-invar* $\equiv \lambda(f, brk).$
 NFlow $c\ s\ t\ f$
 $\wedge (brk \longrightarrow (\forall p. \neg NPreflow.isAugmentingPath\ c\ s\ t\ f\ p))$

Finally, we obtain the Ford-Fulkerson algorithm. Note that we annotate some assertions to ease later refinement

definition *fofu* $\equiv do \{$
 let $f_0 = (\lambda-. 0);$

 $(f, -) \leftarrow while^{fofu-invar}$
 $(\lambda(f, brk). \neg brk)$
 $(\lambda(f, -). do \{$
 $p \leftarrow find-augmenting-spec\ f;$
 case p *of*
 $None \Rightarrow return\ (f, True)$
 $| Some\ p \Rightarrow do \{$
 assert ($p \neq []$);
 assert (*NPreflow.isAugmentingPath* $c\ s\ t\ f\ p$);
 let $f = NFlow.augment-with-path\ c\ f\ p;$
 assert (*NFlow* $c\ s\ t\ f$);
 return ($f, False$)
 $\}$
 $\}$
 $(f_0, False);$
 assert (*NFlow* $c\ s\ t\ f$);
 return f
 $\}$

2.2 Partial Correctness

Correctness of the algorithm is a consequence from the Ford-Fulkerson theorem. We need a few straightforward auxiliary lemmas, though:

The zero flow is a valid flow

lemma *zero-flow*: *NFlow* $c\ s\ t\ (\lambda-. 0)$
 $\langle proof \rangle$

Augmentation preserves the flow property

lemma (in *NFlow*) *augment-pres-nflow*:
assumes *AUG*: *isAugmentingPath p*
shows *NFlow c s t (augment (augmentingFlow p))*
 ⟨*proof*⟩

Augmenting paths cannot be empty

lemma (in *NFlow*) *augmenting-path-not-empty*:
 $\neg \text{isAugmentingPath } []$
 ⟨*proof*⟩

Finally, we can use the verification condition generator to show correctness

theorem *fofu-partial-correct*: $\text{fofu} \leq (\text{spec } f. \text{isMaxFlow } f)$
 ⟨*proof*⟩

2.3 Algorithm without Assertions

For presentation purposes, we extract a version of the algorithm without assertions, and using a bit more concise notation

context begin

private abbreviation (*input*) *augment*
 $\equiv \text{NFlow.augment-with-path}$
private abbreviation (*input*) *is-augmenting-path f p*
 $\equiv \text{NPreflow.isAugmentingPath } c \ s \ t \ f \ p$

definition *ford-fulkerson-method* $\equiv \text{do } \{$
 $\text{let } f_0 = (\lambda(u,v). 0);$

 $(f, brk) \leftarrow \text{while } (\lambda(f, brk). \neg brk)$
 $(\lambda(f, brk). \text{do } \{$
 $\text{p} \leftarrow \text{select } p. \text{is-augmenting-path } f \ p;$
 $\text{case } p \text{ of}$
 $\text{None} \Rightarrow \text{return } (f, \text{True})$
 $\mid \text{Some } p \Rightarrow \text{return } (\text{augment } c \ f \ p, \text{False})$
 $\})$
 $(f_0, \text{False});$
 $\text{return } f$
 $\}$

end — Anonymous context

end — Network

theorem (in *Network*) *ford-fulkerson-method* $\leq (\text{spec } f. \text{isMaxFlow } f)$

⟨*proof*⟩

end — Theory

3 Edmonds-Karp Algorithm

theory *EdmondsKarp-Termination-Abstract* **imports**
Flow-Networks.Ford-Fulkerson
begin

lemma *mlex-fst-decrI*:
fixes $a\ a'\ b\ b'\ N :: \text{nat}$
assumes $a < a'$
assumes $b < N$ $b' < N$
shows $a * N + b < a' * N + b'$
 $\langle \text{proof} \rangle$

lemma (**in** *NFlow*) *augmenting-path-imp-shortest*:
 $\text{isAugmentingPath } p \implies \exists p. \text{Graph.isShortestPath } cf\ s\ p\ t$
 $\langle \text{proof} \rangle$

lemma (**in** *NFlow*) *shortest-is-augmenting*:
 $\text{Graph.isShortestPath } cf\ s\ p\ t \implies \text{isAugmentingPath } p$
 $\langle \text{proof} \rangle$

3.1 Complexity and Termination Analysis

In this section, we show that the loop iterations of the Edmonds-Karp algorithm are bounded by $O(VE)$.

The basic idea of the proof is, that a path that takes an edge reverse to an edge on some shortest path cannot be a shortest path itself.

As augmentation flips at least one edge, this yields a termination argument: After augmentation, either the minimum distance between source and target increases, or it remains the same, but the number of edges that lay on a shortest path decreases. As the minimum distance is bounded by V , we get termination within $O(VE)$ loop iterations.

context *Graph* **begin**

The basic idea is expressed in the following lemma, which, however, is not general enough to be applied for the correctness proof, where we flip more than one edge simultaneously.

lemma *isShortestPath-flip-edge*:
assumes $\text{isShortestPath } s\ p\ t$ $(u,v) \in \text{set } p$
assumes $\text{isPath } s\ p'\ t$ $(v,u) \in \text{set } p'$
shows $\text{length } p' \geq \text{length } p + 2$
 $\langle \text{proof} \rangle$

To be used for the analysis of augmentation, we have to generalize the lemma to simultaneous flipping of edges:

lemma *isShortestPath-flip-edges*:
assumes $\text{Graph.E } c' \supseteq E - \text{edges}$ $\text{Graph.E } c' \subseteq E \cup (\text{prod.swap } \text{'edges})$

assumes SP : $isShortestPath\ s\ p\ t$ **and** $EDGES-SS$: $edges \subseteq set\ p$
assumes P' : $Graph.isPath\ c'\ s\ p'\ t$ $prod.swap\ edges \cap set\ p' \neq \{\}$
shows $length\ p + 2 \leq length\ p'$
 $\langle proof \rangle$

end — Graph

We outsource the more specific lemmas to their own locale, to prevent name space pollution

locale $ek-analysis-defs = Graph +$
fixes $s\ t :: node$

locale $ek-analysis = ek-analysis-defs + Finite-Graph$
begin

definition (**in** $ek-analysis-defs$)
 $spEdges \equiv \{e. \exists p. e \in set\ p \wedge isShortestPath\ s\ p\ t\}$

lemma $spEdges-ss-E$: $spEdges \subseteq E$
 $\langle proof \rangle$

lemma $finite-spEdges[simp, intro]$: $finite\ (spEdges)$
 $\langle proof \rangle$

definition (**in** $ek-analysis-defs$) $uE \equiv E \cup E^{-1}$

lemma $finite-uE[simp, intro]$: $finite\ uE$
 $\langle proof \rangle$

lemma $E-ss-uE$: $E \subseteq uE$
 $\langle proof \rangle$

lemma $card-spEdges-le$:
shows $card\ spEdges \leq card\ uE$
 $\langle proof \rangle$

lemma $card-spEdges-less$:
shows $card\ spEdges < card\ uE + 1$
 $\langle proof \rangle$

definition (**in** $ek-analysis-defs$) $ekMeasure \equiv$
if (*connected* $s\ t$) *then*
 $(card\ V - min-dist\ s\ t) * (card\ uE + 1) + (card\ (spEdges))$
else 0

lemma $measure-decr$:
assumes SV : $s \in V$
assumes SP : $isShortestPath\ s\ p\ t$

assumes *SP-EDGES*: $edges \subseteq set\ p$
assumes *Ebounds*:
 $Graph.E\ c' \supseteq E - edges \cup prod.swap'edges$
 $Graph.E\ c' \subseteq E \cup prod.swap'edges$
shows *ek-analysis-defs.ekMeasure* $c' s\ t \leq ekMeasure$
and $edges - Graph.E\ c' \neq \{\}$
 $\implies ek-analysis-defs.ekMeasure\ c' s\ t < ekMeasure$
<proof>

end — Analysis locale

As a first step to the analysis setup, we characterize the effect of augmentation on the residual graph

context *Graph*
begin

definition *augment-cf edges cap* $\equiv \lambda e.$
if $e \in edges$ *then* $c\ e - cap$
else if $prod.swap\ e \in edges$ *then* $c\ e + cap$
else $c\ e$

lemma *augment-cf-empty[simp]*: $augment-cf\ \{\}\ cap = c$
<proof>

lemma *augment-cf-ss-V*: $\llbracket edges \subseteq E \rrbracket \implies Graph.V\ (augment-cf\ edges\ cap) \subseteq V$
<proof>

lemma *augment-saturate*:
fixes $edges\ e$
defines $c' \equiv augment-cf\ edges\ (c\ e)$
assumes *EIE*: $e \in edges$
shows $e \notin Graph.E\ c'$
<proof>

lemma *augment-cf-split*:
assumes $edges1 \cap edges2 = \{\}$ $edges1^{-1} \cap edges2 = \{\}$
shows $Graph.augment-cf\ c\ (edges1 \cup edges2)\ cap$
 $= Graph.augment-cf\ (Graph.augment-cf\ c\ edges1\ cap)\ edges2\ cap$
<proof>

end — Graph

context *NFlow* **begin**

lemma *augmenting-edge-no-swap*: $isAugmentingPath\ p \implies set\ p \cap (set\ p)^{-1} = \{\}$
<proof>

lemma *aug-flows-finite[simp, intro!]*:

finite {*cf* *e* | *e*. *e* ∈ *set p*}
 ⟨*proof*⟩

lemma *aug-flows-finite'*[*simp, intro!*]:
finite {*cf* (*u,v*) | *u v*. (*u,v*) ∈ *set p*}
 ⟨*proof*⟩

lemma *augment-alt*:
assumes *AUG*: *isAugmentingPath p*
defines *f'* ≡ *augment* (*augmentingFlow p*)
defines *cf'* ≡ *residualGraph c f'*
shows *cf'* = *Graph.augment-cf cf* (*set p*) (*resCap p*)
 ⟨*proof*⟩

lemma *augmenting-path-contains-resCap*:
assumes *isAugmentingPath p*
obtains *e* **where** *e* ∈ *set p* *cf e* = *resCap p*
 ⟨*proof*⟩

Finally, we show the main theorem used for termination and complexity analysis: Augmentation with a shortest path decreases the measure function.

theorem *shortest-path-decr-ek-measure*:
fixes *p*
assumes *SP*: *Graph.isShortestPath cf s p t*
defines *f'* ≡ *augment* (*augmentingFlow p*)
defines *cf'* ≡ *residualGraph c f'*
shows *ek-analysis-defs.ekMeasure cf' s t* < *ek-analysis-defs.ekMeasure cf s t*
 ⟨*proof*⟩

end — Network with flow

end
theory *EdmondsKarp-Algo*
imports *EdmondsKarp-Termination-Abstract FordFulkerson-Algo*
begin

In this theory, we formalize an abstract version of Edmonds-Karp algorithm, which we obtain by refining the Ford-Fulkerson algorithm to always use shortest augmenting paths.

Then, we show that the algorithm always terminates within $O(VE)$ iterations.

3.2 Algorithm

context *Network*
begin

First, we specify the refined procedure for finding augmenting paths

definition *find-shortest-augmenting-spec* $f \equiv \text{assert } (NFlow\ c\ s\ t\ f) \gg$
(select p. Graph.isShortestPath (residualGraph c f) s p t)

We show that our refined procedure is actually a refinement

thm *SELECT-refine*

lemma *find-shortest-augmenting-refine[refine]*:

$(f',f) \in Id \implies \text{find-shortest-augmenting-spec } f' \leq \Downarrow (Id)\ \text{option-rel } (\text{find-augmenting-spec } f)$
<proof>

Next, we specify the Edmonds-Karp algorithm. Our first specification still uses partial correctness, termination will be proved afterwards.

definition *edka-partial* $\equiv \text{do } \{$

let f = (λ -. 0);

$(f,-) \leftarrow \text{while}^{\text{fofu-invar}}$

$(\lambda(f,brk). \neg brk)$

$(\lambda(f,-). \text{do } \{$

p $\leftarrow \text{find-shortest-augmenting-spec } f;$

case p of

None $\Rightarrow \text{return } (f, True)$

| *Some p* $\Rightarrow \text{do } \{$

assert $(p \neq []);$

assert $(NPreflow.isAugmentingPath\ c\ s\ t\ f\ p);$

assert $(Graph.isShortestPath\ (residualGraph\ c\ f)\ s\ p\ t);$

let f = NFlow.augment-with-path c f p;

assert $(NFlow\ c\ s\ t\ f);$

return $(f, False)$

$\}$

$\})$

$(f, False);$

assert $(NFlow\ c\ s\ t\ f);$

return f

$\}$

lemma *edka-partial-refine[refine]*: *edka-partial* $\leq \Downarrow Id\ \text{fofu}$

<proof>

end — Network

3.2.1 Total Correctness

context *Network begin*

We specify the total correct version of Edmonds-Karp algorithm.

definition *edka* $\equiv \text{do } \{$

let f = (λ -. 0);

```

(f,-) ← whileTfofu-invar
  (λ(f,brk). ¬brk)
  (λ(f,-). do {
    p ← find-shortest-augmenting-spec f;
    case p of
      None ⇒ return (f,True)
    | Some p ⇒ do {
      assert (p≠[]);
      assert (NPreflow.isAugmentingPath c s t f p);
      assert (Graph.isShortestPath (residualGraph c f) s p t);
      let f = NFlow.augment-with-path c f p;
      assert (NFlow c s t f);
      return (f, False)
    }
  })
  (f,False);
assert (NFlow c s t f);
return f
}

```

Based on the measure function, it is easy to obtain a well-founded relation that proves termination of the loop in the Edmonds-Karp algorithm:

definition *edka-wf-rel* ≡ *inv-image*
 (*less-than-bool* <*lex*> *measure* (λcf. *ek-analysis-defs.ekMeasure* cf s t))
 (λ(f,brk). (¬brk, *residualGraph* c f))

lemma *edka-wf-rel-wf*[*simp*, *intro!*]: *wf edka-wf-rel*
 ⟨*proof*⟩

The following theorem states that the total correct version of Edmonds-Karp algorithm refines the partial correct one.

theorem *edka-refine*[*refine*]: *edka* ≤ \Downarrow *Id edka-partial*
 ⟨*proof*⟩

3.2.2 Complexity Analysis

For the complexity analysis, we additionally show that the measure function is bounded by $O(VE)$. Note that our absolute bound is not as precise as possible, but clearly $O(VE)$.

lemma *ekMeasure-upper-bound*:
ek-analysis-defs.ekMeasure (*residualGraph* c (λ-. 0)) s t
 < 2 * *card* V * *card* E + *card* V
 ⟨*proof*⟩

Finally, we present a version of the Edmonds-Karp algorithm which is instrumented with a loop counter, and asserts that there are less than $2|V||E| + |V| = O(|V||E|)$ iterations.

Note that we only count the non-breaking loop iterations.

The refinement is achieved by a refinement relation, coupling the instrumented loop state with the uninstrumented one

definition *edkac-rel* $\equiv \{((f, brk, itc), (f, brk)) \mid f \text{ brk } itc.$
 $itc + ek\text{-analysis-defs.ekMeasure (residualGraph } c \text{ f) } s \text{ t}$
 $< 2 * card \ V * card \ E + card \ V$
 $\}$

definition *edka-complexity* $\equiv do \{$
 $let \ f = (\lambda -. 0);$
 $(f, -, itc) \leftarrow while_T$
 $(\lambda(f, brk, -). \neg brk)$
 $(\lambda(f, -, itc). do \{$
 $\ p \leftarrow find\text{-shortest-augmenting-spec } f;$
 $\ case \ p \ of$
 $\ \ \ None \Rightarrow return \ (f, True, itc)$
 $\ \ \ | \ Some \ p \Rightarrow do \{$
 $\ \ \ \ \ let \ f = NFlow.augment\text{-with-path } c \ f \ p;$
 $\ \ \ \ \ return \ (f, False, itc + 1)$
 $\ \ \ \ \ }$
 $\ \ \ }$
 $\ \ \ })$
 $(f, False, 0);$
 $assert \ (itc < 2 * card \ V * card \ E + card \ V);$
 $return \ f$
 $\}$

lemma *edka-complexity-refine*: *edka-complexity* $\leq \Downarrow Id \ edka$
 $\langle proof \rangle$

We show that this algorithm never fails, and computes a maximum flow.

theorem *edka-complexity* $\leq (spec \ f. isMaxFlow \ f)$
 $\langle proof \rangle$

end — Network
end — Theory

4 Breadth First Search

theory *Augmenting-Path-BFS*
imports
 $\ Flow\text{-Networks.Refine-Add-Fofu}$
 $\ Flow\text{-Networks.Graph-Impl}$
begin

In this theory, we present a verified breadth-first search with an efficient imperative implementation. It is parametric in the successor function.

4.1 Algorithm

locale *pre-bfs-invar* = *Graph* +
fixes *src dst* :: *node*
begin

abbreviation *ndist v* \equiv *min-dist src v*

definition *Vd* :: *nat* \Rightarrow *node set*

where

$\bigwedge d. Vd\ d \equiv \{v. \text{connected } src\ v \wedge ndist\ v = d\}$

lemma *Vd-disj*: $\bigwedge d\ d'. d \neq d' \implies Vd\ d \cap Vd\ d' = \{\}$
 $\langle proof \rangle$

lemma *src-Vd0[simp]*: $Vd\ 0 = \{src\}$
 $\langle proof \rangle$

lemma *in-Vd-conv*: $v \in Vd\ d \iff \text{connected } src\ v \wedge ndist\ v = d$
 $\langle proof \rangle$

lemma *Vd-succ*:

assumes $u \in Vd\ d$

assumes $(u, v) \in E$

assumes $\forall i \leq d. v \notin Vd\ i$

shows $v \in Vd\ (Suc\ d)$

$\langle proof \rangle$

end

locale *valid-PRED* = *pre-bfs-invar* +

fixes *PRED* :: *node* \rightarrow *node*

assumes *SRC-IN-V[simp]*: $src \in V$

assumes *FIN-V[simp, intro!]*: *finite V*

assumes *PRED-src[simp]*: $PRED\ src = Some\ src$

assumes *PRED-dist*: $\llbracket v \neq src; PRED\ v = Some\ u \rrbracket \implies ndist\ v = Suc\ (ndist\ u)$

assumes *PRED-E*: $\llbracket v \neq src; PRED\ v = Some\ u \rrbracket \implies (u, v) \in E$

assumes *PRED-closed*: $\llbracket PRED\ v = Some\ u \rrbracket \implies u \in dom\ PRED$

begin

lemma *FIN-E[simp, intro!]*: *finite E* $\langle proof \rangle$

lemma *FIN-succ[simp, intro!]*: *finite* $(E''\{u\})$
 $\langle proof \rangle$

end

locale *nf-invar'* = *valid-PRED c src dst PRED* **for** *c src dst*

and *PRED* :: *node* \rightarrow *node*

and *C N* :: *node set*

and *d* :: *nat*

+

assumes *VIS-eg*: $dom\ PRED = N \cup \{u. \exists i \leq d. u \in Vd\ i\}$

assumes C -ss: $C \subseteq \text{Vd } d$
assumes N -eq: $N = \text{Vd } (d+1) \cap E''(\text{Vd } d - C)$

assumes dst -ne-VIS: $dst \notin \text{dom } PRED$

locale nf -invar = nf -invar' +
assumes $empty$ -assm: $C = \{\}$ \implies $N = \{\}$

locale f -invar = $valid$ -PRED c src dst PRED **for** c src dst
and PRED :: $node \rightarrow node$
and d :: nat
+
assumes dst -found: $dst \in \text{dom } PRED \cap \text{Vd } d$

context *Graph* **begin**

abbreviation $outer$ -loop-invar src $dst \equiv \lambda(f, PRED, C, N, d).$
 $(f \rightarrow f\text{-invar } c \text{ src } dst \text{ PRED } d) \wedge$
 $(\neg f \rightarrow nf\text{-invar } c \text{ src } dst \text{ PRED } C \text{ N } d)$

abbreviation $assn1$ src $dst \equiv \lambda(f, PRED, C, N, d).$
 $\neg f \wedge nf\text{-invar}' \text{ } c \text{ src } dst \text{ PRED } C \text{ N } d$

definition add -succ-spec dst $succ$ v PRED $N \equiv \text{ASSERT } (N \subseteq \text{dom } PRED) \gg$
 $\text{SPEC } (\lambda(f, PRED', N').$
case f of
 $False \Rightarrow dst \notin succ - \text{dom } PRED$
 $\wedge PRED' = \text{map-mmupd } PRED (succ - \text{dom } PRED) \text{ } v$
 $\wedge N' = N \cup (succ - \text{dom } PRED)$
| $True \Rightarrow dst \in succ - \text{dom } PRED$
 $\wedge PRED \subseteq_m PRED'$
 $\wedge PRED' \subseteq_m \text{map-mmupd } PRED (succ - \text{dom } PRED) \text{ } v$
 $\wedge dst \in \text{dom } PRED'$
 $)$

definition pre -bfs :: $node \Rightarrow node \Rightarrow (nat \times (node \rightarrow node)) \text{ option } nres$
where pre -bfs src $dst \equiv \text{do } \{$
 $(f, PRED, -, -, d) \leftarrow \text{WHILEIT } (outer\text{-loop-invar } src \text{ } dst)$
 $(\lambda(f, PRED, C, N, d). f = False \wedge C \neq \{\})$
 $(\lambda(f, PRED, C, N, d). \text{do } \{$
 $v \leftarrow \text{SPEC } (\lambda v. v \in C); \text{let } C = C - \{v\};$
 $\text{ASSERT } (v \in V);$
 $\text{let } succ = (E''\{v\});$
 $\text{ASSERT } (finite \text{ } succ);$
 $(f, PRED, N) \leftarrow add\text{-succ-spec } dst \text{ } succ \text{ } v \text{ } PRED \text{ } N;$
 $\text{if } f \text{ then}$
 $\text{RETURN } (f, PRED, C, N, d+1)$
 $\text{else do } \{$

```

    ASSERT (assn1 src dst (f,PRED,C,N,d));
    if (C={}) then do {
      let C=N;
      let N={};
      let d=d+1;
      RETURN (f,PRED,C,N,d)
    } else RETURN (f,PRED,C,N,d)
  }
}
(False,[src→src],[src],{ },0::nat);
if f then RETURN (Some (d, PRED)) else RETURN None
}

```

4.2 Correctness Proof

lemma (in *nf-invar'*) *ndist-C[simp]*: $\llbracket v \in C \rrbracket \implies \text{ndist } v = d$
 ⟨proof⟩

lemma (in *nf-invar*) *CVdI*: $\llbracket u \in C \rrbracket \implies u \in Vd \ d$
 ⟨proof⟩

lemma (in *nf-invar*) *inPREDD*:
 $\llbracket PRED \ v = \text{Some } u \rrbracket \implies v \in N \vee (\exists i \leq d. v \in Vd \ i)$
 ⟨proof⟩

lemma (in *nf-invar'*) *C-ss-VIS*: $\llbracket v \in C \rrbracket \implies v \in \text{dom } PRED$
 ⟨proof⟩

lemma (in *nf-invar*) *invar-succ-step*:
assumes $v \in C$
assumes $\text{dst} \notin E'\{v\} - \text{dom } PRED$
shows *nf-invar' c src dst*
 (*map-mmupd* PRED ($E'\{v\} - \text{dom } PRED$) v)
 ($C - \{v\}$)
 ($N \cup (E'\{v\} - \text{dom } PRED)$)
 d
 ⟨proof⟩

lemma *invar-init*: $\llbracket \text{src} \neq \text{dst}; \text{src} \in V; \text{finite } V \rrbracket$
 $\implies \text{nf-invar } c \ \text{src} \ \text{dst} \ [\text{src} \mapsto \text{src}] \ \{\text{src}\} \ \{\} \ 0$
 ⟨proof⟩

lemma (in *nf-invar*) *invar-exit*:
assumes $\text{dst} \in C$
shows *f-invar c src dst PRED d*
 ⟨proof⟩

lemma (in *nf-invar*) *invar-C-ss-V*: $u \in C \implies u \in V$
 ⟨proof⟩

lemma (in *nf-invar*) *invar-N-ss-Vis*: $u \in N \implies \exists v. PRED\ u = \text{Some } v$
 ⟨proof⟩

lemma (in *pre-bfs-invar*) *Vdsucinter-conv*[*simp*]:
 $\forall d (Suc\ d) \cap E \text{ “ } \forall d\ d = \forall d (Suc\ d)$
 ⟨proof⟩

lemma (in *nf-invar'*) *invar-shift*:
assumes [*simp*]: $C = \{\}$
shows *nf-invar* *c src dst PRED N* $\{\}$ (*Suc d*)
 ⟨proof⟩

lemma (in *nf-invar'*) *invar-restore*:
assumes [*simp*]: $C \neq \{\}$
shows *nf-invar* *c src dst PRED C N d*
 ⟨proof⟩

definition *bfs-spec src dst r* \equiv (
case r of *None* $\Rightarrow \neg$ *connected src dst*
 | *Some (d,PRED)* \Rightarrow *connected src dst*
 \wedge *min-dist src dst = d*
 \wedge *valid-PRED c src PRED*
 \wedge *dst* \in *dom PRED*)

lemma (in *f-invar*) *invar-found*:
shows *bfs-spec src dst (Some (d,PRED))*
 ⟨proof⟩

lemma (in *nf-invar*) *invar-not-found*:
assumes [*simp*]: $C = \{\}$
shows *bfs-spec src dst None*
 ⟨proof⟩

lemma *map-le-mp*: $\llbracket m \subseteq_m m' ; m\ k = \text{Some } v \rrbracket \implies m'\ k = \text{Some } v$
 ⟨proof⟩

lemma (in *nf-invar*) *dst-notin-Vdd*[*intro, simp*]: $i \leq d \implies \text{dst} \notin \forall d\ i$
 ⟨proof⟩

lemma (in *nf-invar*) *invar-exit'*:
assumes $u \in C \quad (u, \text{dst}) \in E \quad \text{dst} \in \text{dom } PRED'$
assumes *SS1*: $PRED \subseteq_m PRED'$
and *SS2*: $PRED' \subseteq_m \text{map-mmupd } PRED (E \text{ “ } \{u\} - \text{dom } PRED)$ *u*
shows *f-invar* *c src dst PRED'* (*Suc d*)
 ⟨proof⟩

definition *max-dist src* $\equiv \text{Max } (\text{min-dist } \text{src } V)$

definition *outer-loop-rel src* \equiv
inv-image (
 less-than-bool
 $\langle *lex* \rangle$ *greater-bounded* (*max-dist src* + 1)
 $\langle *lex* \rangle$ *finite-psubset*
 $(\lambda(f, PRED, C, N, d). (\neg f, d, C))$)

lemma *outer-loop-rel-wf*:
assumes *finite V*
shows *wf (outer-loop-rel src)*
 $\langle proof \rangle$

lemma (**in** *nf-invar*) *C-ne-max-dist*:
assumes $C \neq \{\}$
shows $d \leq \text{max-dist } src$
 $\langle proof \rangle$

lemma (**in** *nf-invar*) *Vd-ss-V*: $\forall d \subseteq V$
 $\langle proof \rangle$

lemma (**in** *nf-invar*) *finite-C[simp, intro!]*: *finite C*
 $\langle proof \rangle$

lemma (**in** *nf-invar*) *finite-succ*: *finite (E“{u})*
 $\langle proof \rangle$

theorem *pre-bfs-correct*:
assumes [*simp*]: $src \in V \quad src \neq dst$
assumes [*simp*]: *finite V*
shows $pre\text{-}bfs \ src \ dst \leq SPEC \ (bfs\text{-}spec \ src \ dst)$
 $\langle proof \rangle$

definition *bfs-core* $:: node \Rightarrow node \Rightarrow (nat \times (node \rightarrow node)) \ option \ nres$
where *bfs-core src dst* $\equiv do \{$
 $(f, P, -, -, d) \leftarrow while_T \ (\lambda(f, P, C, N, d). f = False \wedge C \neq \{\})$
 $(\lambda(f, P, C, N, d). do \{$
 $v \leftarrow spec \ v. v \in C; \text{let } C = C - \{v\};$
 $\text{let } succ = (E“\{v\});$
 $(f, P, N) \leftarrow add\text{-}succ\text{-}spec \ dst \ succ \ v \ P \ N;$
 if f **then**
 $\text{return } (f, P, C, N, d+1)$
 else $do \{$
 if $(C = \{\})$ **then** $do \{$
 $\text{let } C = N; \text{let } N = \{\}; \text{let } d = d+1;$
 $\text{return } (f, P, C, N, d)$
 $\} \text{ else } \text{return } (f, P, C, N, d)$
 $\} \}$

```

    }
  })
  (False, [src → src], {src}, {}, 0 :: nat);
  if f then return (Some (d, P)) else return None
}

```

theorem

```

assumes src ∈ V    src ≠ dst    finite V
shows bfs-core src dst ≤ (spec p. bfs-spec src dst p)
⟨proof⟩

```

4.3 Extraction of Result Path

```

definition extract-rpath src dst PRED ≡ do {
  (-, p) ← WHILEIT
  (λ(v, p).
    v ∈ dom PRED
  ∧ isPath v p dst
  ∧ distinct (pathVertices v p)
  ∧ (∀ v' ∈ set (pathVertices v p).
    pre-bfs-invar.ndist c src v ≤ pre-bfs-invar.ndist c src v')
  ∧ pre-bfs-invar.ndist c src v + length p
    = pre-bfs-invar.ndist c src dst)
  (λ(v, p). v ≠ src) (λ(v, p). do {
    ASSERT (v ∈ dom PRED);
    let u = the (PRED v);
    let p = (u, v) # p;
    let v = u;
    RETURN (v, p)
  }) (dst, []);
  RETURN p
}

```

end

context valid-PRED **begin**

```

lemma extract-rpath-correct:
assumes dst ∈ dom PRED
shows extract-rpath src dst PRED
  ≤ SPEC (λp. isSimplePath src p dst ∧ length p = ndist dst)
⟨proof⟩

```

end

context Graph **begin**

```

definition bfs src dst ≡ do {
  if src = dst then RETURN (Some [])
  else do {

```

```

    br ← pre-bfs src dst;
    case br of
      None ⇒ RETURN None
    | Some (d,PRED) ⇒ do {
      p ← extract-rpath src dst PRED;
      RETURN (Some p)
    }
  }
}

```

lemma *bfs-correct*:

assumes $src \in V$ *finite* V

shows $bfs\ src\ dst$

$\leq SPEC\ (\lambda$

$\quad None \Rightarrow \neg connected\ src\ dst$

$\quad | Some\ p \Rightarrow isShortestPath\ src\ p\ dst)$

$\langle proof \rangle$

end

context *Finite-Graph* **begin**

interpretation *Refine-Monadic-Syntax* $\langle proof \rangle$

theorem

assumes $src \in V$

shows $bfs\ src\ dst \leq (spec\ p.\ case\ p\ of$

$\quad None \Rightarrow \neg connected\ src\ dst$

$\quad | Some\ p \Rightarrow isShortestPath\ src\ p\ dst)$

$\langle proof \rangle$

end

4.4 Inserting inner Loop and Successor Function

context *Graph* **begin**

definition *inner-loop* $dst\ succ\ u\ PRED\ N \equiv FOREACHci$

$(\lambda it\ (f, PRED', N').$

$\quad PRED' = map-mmupd\ PRED\ ((succ - it) - dom\ PRED)\ u$

$\quad \wedge\ N' = N \cup ((succ - it) - dom\ PRED)$

$\quad \wedge\ f = (dst \in (succ - it) - dom\ PRED)$

$\quad)$

$(succ)$

$(\lambda(f, PRED, N). \neg f)$

$(\lambda v\ (f, PRED, N). do\ \{$

$\quad if\ v \in dom\ PRED\ then\ RETURN\ (f, PRED, N)$

$\quad else\ do\ \{$

$\quad \quad let\ PRED = PRED(v \mapsto u);$

$\quad \quad ASSERT\ (v \notin N);$

$\quad \quad let\ N = insert\ v\ N;$

```

    RETURN (v=dst,PRED,N)
  }
})
(False,PRED,N)

```

lemma *inner-loop-refine*[*refine*]:

```

assumes [simp]: finite succ
assumes [simplified, simp]:
  (succi,succ)∈Id (ui,u)∈Id (PREDi,PRED)∈Id (Ni,N)∈Id
shows inner-loop dst succi ui PREDi Ni
  ≤  $\Downarrow$ Id (add-succ-spec dst succ u PRED N)
<proof>

```

definition *inner-loop2 dst succl u PRED N* ≡ *nfoldli*

```

(succl) (λ(f,-,-). ¬f) (λv (f,PRED,N). do {
  if PRED v ≠ None then RETURN (f,PRED,N)
  else do {
    let PRED = PRED(v ↦ u);
    ASSERT (v∉N);
    let N = insert v N;
    RETURN ((v=dst),PRED,N)
  }
}) (False,PRED,N)

```

lemma *inner-loop2-refine*:

```

assumes SR: (succl, succ) ∈  $\langle$ Id $\rangle$ list-set-rel
shows inner-loop2 dst succl u PRED N ≤  $\Downarrow$ Id (inner-loop dst succ u PRED N)
<proof>

```

thm *conc-trans*[*OF inner-loop2-refine inner-loop-refine, no-vars*]

lemma *inner-loop2-correct*:

```

assumes (succl, succ) ∈  $\langle$ Id $\rangle$ list-set-rel

assumes [simplified, simp]:
  (dsti,dst)∈Id (ui, u) ∈ Id (PREDi, PRED) ∈ Id (Ni, N) ∈ Id
shows inner-loop2 dsti succl ui PREDi Ni
  ≤  $\Downarrow$  Id (add-succ-spec dst succ u PRED N)
<proof>

```

type-synonym *bfs-state* = *bool* × (*node* → *node*) × *node set* × *node set* × *nat*

context

fixes *succ* :: *node* ⇒ *node list nres*

begin

definition *init-state* :: *node* \Rightarrow *bfs-state nres*

where

init-state src \equiv *RETURN* (*False*, [*src* \rightarrow *src*], {*src*}, {}, 0 :: *nat*)

definition *pre-bfs2* :: *node* \Rightarrow *node* \Rightarrow (*nat* \times (*node* \rightarrow *node*)) *option nres*

where *pre-bfs2 src dst* \equiv *do* {

s \leftarrow *init-state src*;

(*f*, *PRED*, -, -, *d*) \leftarrow *WHILET* ($\lambda(f, PRED, C, N, d). f = \text{False} \wedge C \neq \{\}$)

($\lambda(f, PRED, C, N, d). \text{do}$ {

ASSERT (*C* \neq {});

v \leftarrow *op-set-pick* *C*; *let* *C* = *C* - {*v*};

ASSERT (*v* \in *V*);

sl \leftarrow *succ v*;

(*f*, *PRED*, *N*) \leftarrow *inner-loop2 dst sl v PRED N*;

if f then

RETURN (*f*, *PRED*, *C*, *N*, *d* + 1)

else do {

ASSERT (*assn1 src dst* (*f*, *PRED*, *C*, *N*, *d*));

if (*C* = {}) *then do* {

let *C* = *N*;

let *N* = {};

let *d* = *d* + 1;

RETURN (*f*, *PRED*, *C*, *N*, *d*)

} *else RETURN* (*f*, *PRED*, *C*, *N*, *d*)

}

}

;

if f then RETURN (*Some* (*d*, *PRED*)) *else RETURN* *None*

}

lemma *pre-bfs2-refine*:

assumes *succ-impl*: $\bigwedge ui u. \llbracket (ui, u) \in Id; u \in V \rrbracket$

\implies *succ ui* \leq *SPEC* ($\lambda l. (l, E^{\llbracket u \rrbracket}) \in \langle Id \rangle \text{list-set-rel}$)

shows *pre-bfs2 src dst* \leq $\Downarrow Id$ (*pre-bfs src dst*)

<proof>

end

definition *bfs2 succ src dst* \equiv *do* {

if src = *dst then*

RETURN (*Some* [])

else do {

br \leftarrow *pre-bfs2 succ src dst*;

case br of

None \Rightarrow *RETURN* *None*

| *Some* (*d*, *PRED*) \Rightarrow *do* {

p \leftarrow *extract-rpath src dst PRED*;

RETURN (*Some p*)

```

}
}
}

```

lemma *bfs2-refine*:

```

assumes succ-impl:  $\bigwedge ui u. \llbracket (ui, u) \in Id; u \in V \rrbracket$ 
 $\impl succ\ ui \leq SPEC (\lambda l. (l, E''\{u\}) \in \langle Id \rangle list\text{-set-rel})$ 
shows bfs2 succ src dst  $\leq \Downarrow Id (bfs\ src\ dst)$ 
 $\langle proof \rangle$ 

```

end

lemma *bfs2-refine-succ*:

```

assumes [refine]:  $\bigwedge ui u. \llbracket (ui, u) \in Id; u \in Graph.V\ c \rrbracket$ 
 $\impl succi\ ui \leq \Downarrow Id (succ\ u)$ 
assumes [simplified, simp]:  $(si, s) \in Id \quad (ti, t) \in Id \quad (ci, c) \in Id$ 
shows Graph.bfs2 ci succi si ti  $\leq \Downarrow Id (Graph.bfs2\ c\ succ\ s\ t)$ 
 $\langle proof \rangle$ 

```

4.5 Imperative Implementation

context *Impl-Succ begin*

```

definition op-bfs :: 'ga  $\Rightarrow$  node  $\Rightarrow$  node  $\Rightarrow$  path option nres
where [simp]: op-bfs c s t  $\equiv Graph.bfs2 (absG\ c) (succ\ c) s\ t$ 

```

lemma *pat-op-dfs*[*pat-rules*]:

```

 $Graph.bfs2\ (absG\ c)\ (succ\ c)\ s\ t \equiv UNPROTECT\ op-bfs\ c\ s\ t$   $\langle proof \rangle$ 

```

sepref-register *PR-CONST op-bfs*

```

:: 'ig  $\Rightarrow$  node  $\Rightarrow$  node  $\Rightarrow$  path option nres

```

type-synonym *ibfs-state*

```

= bool  $\times$  (node, node) i-map  $\times$  node set  $\times$  node set  $\times$  nat

```

sepref-register *Graph.init-state* :: node \Rightarrow *ibfs-state* nres

schematic-goal *init-state-impl*:

```

fixes src :: nat
notes [id-rules] =
  itypeI[Pure.of src TYPE(nat)]
shows hn-refine (hn-val nat-rel src src)
  ( $?c :: ?'c\ Heap$ )  $?T' ?R (Graph.init-state\ src)$ 
 $\langle proof \rangle$ 

```

concrete-definition (**in** $-$) *init-state-impl* **uses** *Impl-Succ.init-state-impl*

lemmas [*sepref-fr-rules*] = *init-state-impl.refine[OF this-loc, to-hfref]*

schematic-goal *bfs-impl*:

```

notes [sepref-opt-simps] = heap-WHILET-def

```

```

fixes s t :: nat
notes [id-rules] =
  itypeI[Pure.of s TYPE(nat)]
  itypeI[Pure.of t TYPE(nat)]
  itypeI[Pure.of c TYPE('ig)]
  — Declare parameters to operation identification
shows hn-refine (
  hn-ctxt (isG) c ci
  * hn-val nat-rel s si
  * hn-val nat-rel t ti) (?c::?'c Heap) ?Γ' ?R (PR-CONST op-bfs c s t)
  ⟨proof⟩

concrete-definition (in —) bfs-impl uses Impl-Succ.bfs-impl
  — Extract generated implementation into constant
prepare-code-thms (in —) bfs-impl-def

lemmas bfs-impl-fr-rule = bfs-impl.refine[OF this-loc,to-hfref]

end

export-code bfs-impl checking SML-imp

end

```

5 Implementation of the Edmonds-Karp Algorithm

```

theory EdmondsKarp-Impl
imports
  EdmondsKarp-Algo
  Augmenting-Path-BFS
  Refine-Imperative-HOL.IICF
begin

```

We now implement the Edmonds-Karp algorithm. Note that, during the implementation, we explicitly write down the whole refined algorithm several times. As refinement is modular, most of these copies could be avoided—we inserted them deliberately for documentation purposes.

5.1 Refinement to Residual Graph

As a first step towards implementation, we refine the algorithm to work directly on residual graphs. For this, we first have to establish a relation between flows in a network and residual graphs.

5.1.1 Refinement of Operations

```

context Network

```

begin

We define the relation between residual graphs and flows

definition *cfi-rel* \equiv *br flow-of-cf* (*RGraph c s t*)

It can also be characterized the other way round, i.e., mapping flows to residual graphs:

lemma *cfi-rel-alt*: *cfi-rel* = $\{(cf, f). cf = residualGraph\ c\ f \wedge NFlow\ c\ s\ t\ f\}$
 $\langle proof \rangle$

Initially, the residual graph for the zero flow equals the original network

lemma *residualGraph-zero-flow*: *residualGraph c* ($\lambda-. 0$) = *c*
 $\langle proof \rangle$

lemma *flow-of-c*: *flow-of-cf c* = ($\lambda-. 0$)
 $\langle proof \rangle$

The residual capacity is naturally defined on residual graphs

definition *resCap-cf cf p* \equiv *Min* $\{cf\ e \mid e. e \in set\ p\}$

lemma (*in NFlow*) *resCap-cf-refine*: *resCap-cf cf p* = *resCap p*
 $\langle proof \rangle$

Augmentation can be done by *Graph.augment-cf*.

lemma (*in NFlow*) *augment-cf-refine-aux*:

assumes *AUG*: *isAugmentingPath p*

shows *residualGraph c* (*augment* (*augmentingFlow p*)) (*u, v*) = (

if (*u, v*) \in *set p* *then* (*residualGraph c f* (*u, v*) - *resCap p*)

else if (*v, u*) \in *set p* *then* (*residualGraph c f* (*u, v*) + *resCap p*)

else *residualGraph c f* (*u, v*)

$\langle proof \rangle$

lemma *augment-cf-refine*:

assumes *R*: (*cf, f*) \in *cfi-rel*

assumes *AUG*: *NPreFlow.isAugmentingPath c s t f p*

shows (*Graph.augment-cf cf* (*set p*) (*resCap-cf cf p*),

NFlow.augment-with-path c f p) \in *cfi-rel*

$\langle proof \rangle$

We rephrase the specification of shortest augmenting path to take a residual graph as parameter

definition *find-shortest-augmenting-spec-cf cf* \equiv

assert (*RGraph c s t cf*) \gg

SPEC (λ

None \Rightarrow \neg *Graph.connected cf s t*

\mid *Some p* \Rightarrow *Graph.isShortestPath cf s p t*)

lemma (*in RGraph*) *find-shortest-augmenting-spec-cf-refine*:

find-shortest-augmenting-spec-cf cf

\leq *find-shortest-augmenting-spec* (*flow-of-cf* *cf*)
 ⟨*proof*⟩

This leads to the following refined algorithm

definition *edka2* \equiv *do* {
let *cf* = *c*;

 (*cf*,-) \leftarrow *while*_T
 (λ (*cf*,*brk*). \neg *brk*)
 (λ (*cf*,-). *do* {
 assert (*RGraph* *c s t cf*);
 p \leftarrow *find-shortest-augmenting-spec-cf* *cf*;
 case *p* *of*
 None \Rightarrow *return* (*cf*, *True*)
 | *Some* *p* \Rightarrow *do* {
 assert (*p* \neq []);
 assert (*Graph.isShortestPath* *cf s p t*);
 let *cf* = *Graph.augment-cf* *cf* (*set* *p*) (*resCap-cf* *cf* *p*);
 assert (*RGraph* *c s t cf*);
 return (*cf*, *False*)
 }
 }
 (*cf*, *False*);
assert (*RGraph* *c s t cf*);
let *f* = *flow-of-cf* *cf*;
return *f*
 }

lemma *edka2-refine*: *edka2* \leq \Downarrow *Id* *edka*
 ⟨*proof*⟩

5.2 Implementation of Bottleneck Computation and Augmentation

We will access the capacities in the residual graph only by a get-operation, which asserts that the edges are valid

abbreviation (*input*) *valid-edge* $::$ *edge* \Rightarrow *bool* **where**
valid-edge \equiv λ (*u,v*). *u* \in *V* \wedge *v* \in *V*

definition *cf-get*
 $::$ *'capacity graph* \Rightarrow *edge* \Rightarrow *'capacity nres*
where *cf-get* *cf e* \equiv *ASSERT* (*valid-edge e*) \gg *RETURN* (*cf e*)

definition *cf-set*
 $::$ *'capacity graph* \Rightarrow *edge* \Rightarrow *'capacity* \Rightarrow *'capacity graph nres*
where *cf-set* *cf e cap* \equiv *ASSERT* (*valid-edge e*) \gg *RETURN* (*cf*(*e:=cap*))

definition *resCap-cf-impl* $::$ *'capacity graph* \Rightarrow *path* \Rightarrow *'capacity nres*
where *resCap-cf-impl* *cf p* \equiv

```

case p of
  [] ⇒ RETURN (0::'capacity)
| (e#p) ⇒ do {
  cap ← cf-get cf e;
  ASSERT (distinct p);
  nfoldli
  p (λ-. True)
  (λe cap. do {
    cape ← cf-get cf e;
    RETURN (min cape cap)
  })
  cap
}

```

lemma (in *RGraph*) *resCap-cf-impl-refine*:
assumes *AUG*: *cf.isSimplePath s p t*
shows *resCap-cf-impl cf p ≤ SPEC (λr. r = resCap-cf cf p)*
<proof>

definition (in *Graph*)
augment-edge e cap ≡ (c(
 e := c e - cap,
 prod.swap e := c (prod.swap e) + cap))

lemma (in *Graph*) *augment-cf-inductive*:
fixes *e cap*
defines *c' ≡ augment-edge e cap*
assumes *P*: *isSimplePath s (e#p) t*
shows *augment-cf (insert e (set p)) cap = Graph.augment-cf c' (set p) cap*
and $\exists s'. \text{Graph.isSimplePath } c' s' p t$
<proof>

definition *augment-edge-impl cf e cap* ≡ do {
 v ← cf-get cf e; cf ← cf-set cf e (v-cap);
 let e = prod.swap e;
 v ← cf-get cf e; cf ← cf-set cf e (v+cap);
 RETURN cf
}

lemma *augment-edge-impl-refine*:
assumes *valid-edge e* $\forall u. e \neq (u, u)$
shows *augment-edge-impl cf e cap*
 $\leq (\text{spec } r. r = \text{Graph.augment-edge } cf e cap)$
<proof>

definition *augment-cf-impl*
 $:: 'capacity \text{ graph} \Rightarrow \text{path} \Rightarrow 'capacity \Rightarrow 'capacity \text{ graph } nres$
where

```

augment-cf-impl cf p x ≡ do {
  (recT D. λ
    ( [], cf) ⇒ return cf
  | (e#p, cf) ⇒ do {
      cf ← augment-edge-impl cf e x;
      D (p, cf)
    }
  ) (p, cf)
}

```

Deriving the corresponding recursion equations

lemma *augment-cf-impl-simps[simp]*:
augment-cf-impl cf [] x = return cf
augment-cf-impl cf (e#p) x = do {
cf ← augment-edge-impl cf e x;
augment-cf-impl cf p x}
 ⟨proof⟩

lemma *augment-cf-impl-aux*:
assumes $\forall e \in \text{set } p. \text{ valid-edge } e$
assumes $\exists s. \text{ Graph.isSimplePath } cf \ s \ p \ t$
shows *augment-cf-impl cf p x ≤ RETURN (Graph.augment-cf cf (set p) x)*
 ⟨proof⟩

lemma (in *RGraph*) *augment-cf-impl-refine*:
assumes *Graph.isSimplePath cf s p t*
shows *augment-cf-impl cf p x ≤ RETURN (Graph.augment-cf cf (set p) x)*
 ⟨proof⟩

Finally, we arrive at the algorithm where augmentation is implemented algorithmically:

definition *edka3* ≡ do {
 let cf = c;

 (cf, -) ← while_T
 (λ(cf, brk). ¬brk)
 (λ(cf, -). do {
 assert (RGraph c s t cf);
 p ← find-shortest-augmenting-spec-cf cf;
 case p of
 None ⇒ return (cf, True)
 | Some p ⇒ do {
 assert (p ≠ []);
 assert (Graph.isShortestPath cf s p t);
 bn ← resCap-cf-impl cf p;
 cf ← augment-cf-impl cf p bn;
 assert (RGraph c s t cf);
 return (cf, False)
 }
 }
}

```

    })
    (cf, False);
    assert (RGraph c s t cf);
    let f = flow-of-cf cf;
    return f
  }

```

lemma *edka3-refine*: $edka3 \leq \Downarrow Id\ edka2$
 ⟨proof⟩

5.3 Refinement to use BFS

We refine the Edmonds-Karp algorithm to use breadth first search (BFS)

```

definition edka4 ≡ do {
  let cf = c;

  (cf, -) ← whileT
    (λ(cf, brk). ¬brk)
    (λ(cf, -). do {
      assert (RGraph c s t cf);
      p ← Graph.bfs cf s t;
      case p of
        None ⇒ return (cf, True)
      | Some p ⇒ do {
          assert (p ≠ []);
          assert (Graph.isShortestPath cf s p t);
          bn ← resCap-cf-impl cf p;
          cf ← augment-cf-impl cf p bn;
          assert (RGraph c s t cf);
          return (cf, False)
        }
      })
    (cf, False);
  assert (RGraph c s t cf);
  let f = flow-of-cf cf;
  return f
}

```

A shortest path can be obtained by BFS

lemma *bfs-refines-shortest-augmenting-spec*:
 $Graph.bfs\ cf\ s\ t \leq find-shortest-augmenting-spec-cf\ cf$
 ⟨proof⟩

lemma *edka4-refine*: $edka4 \leq \Downarrow Id\ edka3$
 ⟨proof⟩

5.4 Implementing the Successor Function for BFS

We implement the successor function in two steps. The first step shows how to obtain the successor function by filtering the list of adjacent nodes. This step contains the idea of the implementation. The second step is purely technical, and makes explicit the recursion of the filter function as a recursion combinator in the monad. This is required for the Sepref tool.

Note: We use *filter-rev* here, as it is tail-recursive, and we are not interested in the order of successors.

definition *rg-succ am cf u* \equiv
filter-rev ($\lambda v. cf (u,v) > 0$) (*am u*)

lemma (in *RGraph*) *rg-succ-ref1*: $\llbracket is-adj-map\ am \rrbracket$
 $\implies (rg-succ\ am\ cf\ u, Graph.E\ cf\ \{u\}) \in \langle Id \rangle list-set-rel$
 $\langle proof \rangle$

definition *ps-get-op* :: $- \Rightarrow node \Rightarrow node\ list\ nres$
where *ps-get-op am u* $\equiv assert (u \in V) \gg return (am\ u)$

definition *monadic-filter-rev-aux*
 :: $'a\ list \Rightarrow ('a \Rightarrow bool\ nres) \Rightarrow 'a\ list \Rightarrow 'a\ list\ nres$

where

monadic-filter-rev-aux a P l $\equiv (rec_T\ D. (\lambda(l,a). case\ l\ of$
 $\quad [] \Rightarrow return\ a$
 $\quad | (v\#\!l) \Rightarrow do\ \{$
 $\quad\quad c \leftarrow P\ v;$
 $\quad\quad let\ a = (if\ c\ then\ v\#\!a\ else\ a);$
 $\quad\quad D\ (l,a)$
 $\quad\ \})$
 $\quad))\ (l,a)$

lemma *monadic-filter-rev-aux-rule*:

assumes $\bigwedge x. x \in set\ l \implies P\ x \leq SPEC\ (\lambda r. r = Q\ x)$
shows *monadic-filter-rev-aux a P l* $\leq SPEC\ (\lambda r. r = filter-rev-aux\ a\ Q\ l)$
 $\langle proof \rangle$

definition *monadic-filter-rev* = *monadic-filter-rev-aux* []

lemma *monadic-filter-rev-rule*:

assumes $\bigwedge x. x \in set\ l \implies P\ x \leq (spec\ r. r = Q\ x)$
shows *monadic-filter-rev P l* $\leq (spec\ r. r = filter-rev\ Q\ l)$
 $\langle proof \rangle$

definition *rg-succ2 am cf u* $\equiv do\ \{$

$\quad l \leftarrow ps-get-op\ am\ u;$
 $\quad monadic-filter-rev\ (\lambda v. do\ \{$
 $\quad\quad x \leftarrow cf-get\ cf\ (u,v);$
 $\quad\quad return\ (x > 0)$
 $\quad\ \})$

```

    }) l
  }

```

lemma (in *RGraph*) *rg-succ-ref2*:
assumes *PS*: *is-adj-map am* **and** *V*: $u \in V$
shows *rg-succ2 am cf u* \leq *return (rg-succ am cf u)*
<proof>

lemma (in *RGraph*) *rg-succ-ref*:
assumes *A*: *is-adj-map am*
assumes *B*: $u \in V$
shows *rg-succ2 am cf u* \leq *SPEC* ($\lambda l. (l, cf.E\{\!|u\!\}) \in \langle Id \rangle list-set-rel$)
<proof>

5.5 Adding Tabulation of Input

Next, we add functions that will be refined to tabulate the input of the algorithm, i.e., the network's capacity matrix and adjacency map, into efficient representations. The capacity matrix is tabulated to give the initial residual graph, and the adjacency map is tabulated for faster access.

Note, on the abstract level, the tabulation functions are just identity, and merely serve as marker constants for implementation.

definition *init-cf* :: '*capacity graph nres*
 — Initialization of residual graph from network
where *init-cf* \equiv *RETURN c*
definition *init-ps* :: (*node* \Rightarrow *node list*) \Rightarrow -
 — Initialization of adjacency map
where *init-ps am* \equiv *ASSERT (is-adj-map am)* \gg *RETURN am*

definition *compute-rflow* :: '*capacity graph* \Rightarrow '*capacity flow nres*
 — Extraction of result flow from residual graph
where
compute-rflow cf \equiv *ASSERT (RGraph c s t cf)* \gg *RETURN (flow-of-cf cf)*

definition *bfs2-op am cf* \equiv *Graph.bfs2 cf (rg-succ2 am cf) s t*

We split the algorithm into a tabulation function, and the running of the actual algorithm:

definition *edka5-tabulate am* \equiv *do* {
cf \leftarrow *init-cf*;
am \leftarrow *init-ps am*;
return (cf, am)
}

definition *edka5-run cf am* \equiv *do* {
(cf, -) \leftarrow *while_T*
 ($\lambda(cf, brk). \neg brk$)
 ($\lambda(cf, -). do$ {

```

    assert (RGraph c s t cf);
    p ← bfs2-op am cf;
    case p of
    None ⇒ return (cf, True)
  | Some p ⇒ do {
    assert (p ≠ []);
    assert (Graph.isShortestPath cf s p t);
    bn ← resCap-cf-impl cf p;
    cf ← augment-cf-impl cf p bn;
    assert (RGraph c s t cf);
    return (cf, False)
  }
  })
  (cf, False);
  f ← compute-rflow cf;
  return f
}

definition edka5 am ≡ do {
  (cf, am) ← edka5-tabulate am;
  edka5-run cf am
}

lemma edka5-refine: [is-adj-map am] ⇒ edka5 am ≤ ↓Id edka4
  ⟨proof⟩

end

```

5.6 Imperative Implementation

In this section we provide an efficient imperative implementation, using the Sepref tool. It is mostly technical, setting up the mappings from abstract to concrete data structures, and then refining the algorithm, function by function.

This is also the point where we have to choose the implementation of capacities. Up to here, they have been a polymorphic type with a typeclass constraint of being a linearly ordered integral domain. Here, we switch to *capacity-impl* (*capacity-impl*).

locale *Network-Impl* = *Network* c s t **for** c :: *capacity-impl graph* **and** s t

Moreover, we assume that the nodes are natural numbers less than some number N , which will become an additional parameter of our algorithm.

locale *Edka-Impl* = *Network-Impl* +
fixes N :: nat
assumes V-ss: $V \subseteq \{0..<N\}$
begin
lemma *this-loc*: *Edka-Impl* c s t N ⟨proof⟩

lemma *E-ss*: $E \subseteq \{0..<N\} \times \{0..<N\}$ *<proof>*

lemma *mtx-nonzero-iff[simp]*: $mtx\text{-nonzero } c = E$ *<proof>*

lemma *mtx-nonzeroN*: $mtx\text{-nonzero } c \subseteq \{0..<N\} \times \{0..<N\}$ *<proof>*

lemma *[simp]*: $v \in V \implies v < N$ *<proof>*

Declare some variables to Sepref.

lemmas *[id-rules]* =
itypeI[*Pure.of N TYPE(nat)*]
itypeI[*Pure.of s TYPE(node)*]
itypeI[*Pure.of t TYPE(node)*]
itypeI[*Pure.of c TYPE(capacity-impl graph)*]

Instruct Sepref to not refine these parameters. This is expressed by using identity as refinement relation.

lemmas *[sepref-import-param]* =
IdI[*of N*]
IdI[*of s*]
IdI[*of t*]

lemma *[sepref-fr-rules]*: $(uncurry0 \text{ (return } c), uncurry0 \text{ (return } c)) \in unit\text{-assn}^k$
 $\rightarrow_a \text{ pure (nat-rel} \times_r \text{ nat-rel} \rightarrow \text{ int-rel)}$
<proof>

5.6.1 Implementation of Adjacency Map by Array

definition *is-am am psi*
 $\equiv \exists_A l. psi \mapsto_a l$
 $* \uparrow(\text{length } l = N \wedge (\forall i < N. l!i = am \ i)$
 $\wedge (\forall i \geq N. am \ i = []))$

lemma *is-am-precise[safe-constraint-rules]*: *precise (is-am)*
<proof>

sepref-decl-intf *i-ps is nat* \Rightarrow *nat list*

definition (**in** $-$) *ps-get-imp psi u* \equiv *Array.nth psi u*

lemma *[def-pat-rules]*: $Network.ps\text{-get-op}\$c \equiv UNPROTECT \text{ ps-get-op}$ *<proof>*
sepref-register *PR-CONST ps-get-op* :: *i-ps* \Rightarrow *node* \Rightarrow *node list nres*

lemma *ps-get-op-refine[sepref-fr-rules]*:
 $(uncurry \text{ ps-get-imp}, uncurry \text{ (PR-CONST ps-get-op)})$
 $\in is\text{-am}^k *_a (\text{pure Id})^k \rightarrow_a \text{ list-assn (pure Id)}$
<proof>

lemma *is-pred-succ-no-node*: $\llbracket \text{is-adj-map } a; u \notin V \rrbracket \implies a \ u = []$
 $\langle \text{proof} \rangle$

lemma [*sepref-fr-rules*]: $(\text{Array.make } N, \text{PR-CONST } \text{init-ps})$
 $\in (\text{pure } \text{Id})^k \rightarrow_a \text{is-am}$
 $\langle \text{proof} \rangle$

lemma [*def-pat-rules*]: $\text{Network.init-ps}\$c \equiv \text{UNPROTECT } \text{init-ps}$ $\langle \text{proof} \rangle$
sepref-register *PR-CONST init-ps* :: $(\text{node} \Rightarrow \text{node list}) \Rightarrow \text{i-ps nres}$

5.6.2 Implementation of Capacity Matrix by Array

lemma [*def-pat-rules*]: $\text{Network.cf-get}\$c \equiv \text{UNPROTECT } \text{cf-get}$ $\langle \text{proof} \rangle$
lemma [*def-pat-rules*]: $\text{Network.cf-set}\$c \equiv \text{UNPROTECT } \text{cf-set}$ $\langle \text{proof} \rangle$

sepref-register

PR-CONST cf-get :: $\text{capacity-impl } \text{i-mtx} \Rightarrow \text{edge} \Rightarrow \text{capacity-impl nres}$

sepref-register

PR-CONST cf-set :: $\text{capacity-impl } \text{i-mtx} \Rightarrow \text{edge} \Rightarrow \text{capacity-impl}$
 $\Rightarrow \text{capacity-impl } \text{i-mtx nres}$

We have to link the matrix implementation, which encodes the bound, to the abstract assertion of the bound

sepref-definition *cf-get-impl is uncurry (PR-CONST cf-get)* :: $(\text{asmtx-assn } N \ \text{id-assn})^k *_{\alpha} (\text{prod-assn } \text{id-assn } \text{id-assn})^k \rightarrow_{\alpha} \text{id-assn}$
 $\langle \text{proof} \rangle$

lemmas [*sepref-fr-rules*] = *cf-get-impl.refine*

lemmas [*sepref-opt-simps*] = *cf-get-impl-def*

sepref-definition *cf-set-impl is uncurry2 (PR-CONST cf-set)* :: $(\text{asmtx-assn } N \ \text{id-assn})^d *_{\alpha} (\text{prod-assn } \text{id-assn } \text{id-assn})^k *_{\alpha} \text{id-assn}^k \rightarrow_{\alpha} \text{asmtx-assn } N \ \text{id-assn}$
 $\langle \text{proof} \rangle$

lemmas [*sepref-fr-rules*] = *cf-set-impl.refine*

lemmas [*sepref-opt-simps*] = *cf-set-impl-def*

sepref-thm *init-cf-impl is uncurry0 (PR-CONST init-cf)* :: $\text{unit-assn}^k \rightarrow_{\alpha} \text{asmtx-assn } N \ \text{id-assn}$
 $\langle \text{proof} \rangle$

concrete-definition (**in** $-$) *init-cf-impl uses Edka-Impl.init-cf-impl.refine-raw*
is $(\text{uncurry0 } ?f, -) \in -$

prepare-code-thms (**in** $-$) *init-cf-impl-def*

lemmas [*sepref-fr-rules*] = *init-cf-impl.refine[OF this-loc]*

lemma *amtx-cnv*: $\text{amtx-assn } N \ M \ \text{id-assn} = \text{ICF-Array-Matrix.is-amtx } N \ M$
 $\langle \text{proof} \rangle$

lemma [def-pat-rules]: $Network.init\text{-}cf\ \$c \equiv UNPROTECT\ init\text{-}cf$ $\langle proof \rangle$
sepref-register $PR\text{-}CONST\ init\text{-}cf :: capacity\text{-}impl\ i\text{-}mtx\ nres$

5.6.3 Representing Result Flow as Residual Graph

definition (in *Network-Impl*) $is\text{-}rflow\ N\ f\ cfi$
 $\equiv \exists\ _A\ cf.\ asmtx\text{-}assn\ N\ id\text{-}assn\ cf\ cfi * \uparrow(RGraph\ c\ s\ t\ cf \wedge f = flow\text{-}of\text{-}cf\ cf)$
lemma $is\text{-}rflow\text{-}precise[safe\text{-}constraint\text{-}rules]:\ precise\ (is\text{-}rflow\ N)$
 $\langle proof \rangle$

sepref-decl-intf $i\text{-}rflow\ is\ nat \times nat \Rightarrow int$

lemma [sepref-fr-rules]:
 $(\lambda cf.\ return\ cfi,\ PR\text{-}CONST\ compute\text{-}rflow) \in (asmtx\text{-}assn\ N\ id\text{-}assn)^d \rightarrow_a$
 $is\text{-}rflow\ N$
 $\langle proof \rangle$

lemma [def-pat-rules]:
 $Network.compute\text{-}rflow\ \$c\ \$s\ \$t \equiv UNPROTECT\ compute\text{-}rflow$ $\langle proof \rangle$
sepref-register
 $PR\text{-}CONST\ compute\text{-}rflow :: capacity\text{-}impl\ i\text{-}mtx \Rightarrow i\text{-}rflow\ nres$

5.6.4 Implementation of Functions

schematic-goal $rg\text{-}succ2\text{-}impl$:
fixes $am :: node \Rightarrow node\ list$ **and** $cf :: capacity\text{-}impl\ graph$
notes [id-rules] =
 $itypeI[Pure.of\ u\ TYPE(node)]$
 $itypeI[Pure.of\ am\ TYPE(i\text{-}ps)]$
 $itypeI[Pure.of\ cf\ TYPE(capacity\text{-}impl\ i\text{-}mtx)]$
notes [sepref-import-param] = $IdI[of\ N]$
notes [sepref-fr-rules] = $HOL\text{-}list\text{-}empty\text{-}hnr$
shows $hn\text{-}refine\ (hn\text{-}ctxt\ is\text{-}am\ am\ psi * hn\text{-}ctxt\ (asmtx\text{-}assn\ N\ id\text{-}assn)\ cf\ cfi$
 $*\ hn\text{-}val\ nat\text{-}rel\ u\ ui)\ (?c::?'c\ Heap)\ ?\Gamma\ ?R\ (rg\text{-}succ2\ am\ cf\ u)$
 $\langle proof \rangle$
concrete-definition (in $-$) $succ\text{-}imp$ **uses** $Edka\text{-}Impl.rg\text{-}succ2\text{-}impl$
prepare-code-thms (in $-$) $succ\text{-}imp\text{-}def$

lemma $succ\text{-}imp\text{-}refine[sepref\text{-}fr\text{-}rules]$:
 $(uncurry2\ (succ\text{-}imp\ N),\ uncurry2\ (PR\text{-}CONST\ rg\text{-}succ2))$
 $\in is\text{-}am^k * _a\ (asmtx\text{-}assn\ N\ id\text{-}assn)^k * _a\ (pure\ Id)^k \rightarrow_a\ list\text{-}assn\ (pure\ Id)$
 $\langle proof \rangle$

lemma [def-pat-rules]: $Network.rg\text{-}succ2\ \$c \equiv UNPROTECT\ rg\text{-}succ2$ $\langle proof \rangle$
sepref-register
 $PR\text{-}CONST\ rg\text{-}succ2 :: i\text{-}ps \Rightarrow capacity\text{-}impl\ i\text{-}mtx \Rightarrow node \Rightarrow node\ list\ nres$

lemma [sepref-import-param]: $(min, min) \in Id \rightarrow Id \rightarrow Id$ $\langle proof \rangle$

abbreviation $is-path \equiv list-assn (prod-assn (pure Id) (pure Id))$

schematic-goal $resCap-imp-impl$:

fixes $am :: node \Rightarrow node\ list$ **and** $cf :: capacity-impl\ graph$ **and** $p\ pi$

notes [id-rules] =

$itypeI[Pure.of\ p\ TYPE(edge\ list)]$

$itypeI[Pure.of\ cf\ TYPE(capacity-impl\ i-mtx)]$

notes [sepref-import-param] = $IdI[of\ N]$

shows $hn-refine$

$(hn-ctxt (asmx-assn\ N\ id-assn)\ cf\ cfi * hn-ctxt\ is-path\ p\ pi)$

$(?c::?'c\ Heap)\ ?\Gamma\ ?R$

$(resCap-cf-impl\ cf\ p)$

$\langle proof \rangle$

concrete-definition (in $-$) $resCap-imp$ **uses** $Edka-Impl.resCap-imp-impl$

prepare-code-thms (in $-$) $resCap-imp-def$

lemma $resCap-impl-refine[sepref-fr-rules]$:

$(uncurry (resCap-imp\ N), uncurry (PR-CONST\ resCap-cf-impl))$

$\in (asmx-assn\ N\ id-assn)^k *_{\alpha} (is-path)^k \rightarrow_{\alpha} (pure\ Id)$

$\langle proof \rangle$

lemma [def-pat-rules]:

$Network.resCap-cf-impl\$c \equiv UNPROTECT\ resCap-cf-impl$

$\langle proof \rangle$

sepref-register $PR-CONST\ resCap-cf-impl$

$:: capacity-impl\ i-mtx \Rightarrow path \Rightarrow capacity-impl\ nres$

sepref-thm $augment-imp$ **is** $uncurry2 (PR-CONST\ augment-cf-impl) :: ((asmx-assn\ N\ id-assn)^d *_{\alpha} (is-path)^k *_{\alpha} (pure\ Id)^k \rightarrow_{\alpha} asmx-assn\ N\ id-assn)$

$\langle proof \rangle$

concrete-definition (in $-$) $augment-imp$ **uses** $Edka-Impl.augment-imp.refine-raw$

is $(uncurry2\ ?f, -) \in -$

prepare-code-thms (in $-$) $augment-imp-def$

lemma $augment-impl-refine[sepref-fr-rules]$:

$(uncurry2 (augment-imp\ N), uncurry2 (PR-CONST\ augment-cf-impl))$

$\in (asmx-assn\ N\ id-assn)^d *_{\alpha} (is-path)^k *_{\alpha} (pure\ Id)^k \rightarrow_{\alpha} asmx-assn\ N$

$id-assn$

$\langle proof \rangle$

lemma [def-pat-rules]:

$Network.augment-cf-impl\$c \equiv UNPROTECT\ augment-cf-impl$

$\langle proof \rangle$

sepref-register $PR-CONST\ augment-cf-impl$

$:: capacity-impl\ i-mtx \Rightarrow path \Rightarrow capacity-impl \Rightarrow capacity-impl\ i-mtx\ nres$

sublocale *bfs*: *Impl-Succ*

snd
TYPE(*i-ps* × *capacity-impl i-mtx*)
PR-CONST ($\lambda(am, cf). rg-succ2\ am\ cf$)
prod-assn is-am (*asmtx-assn N id-assn*)
 $\lambda(am, cf). succ-imp\ N\ am\ cf$
<proof>

definition (**in** $-$) *bfsi' N s t psi cfi*
 $\equiv bfs-impl\ (\lambda(am, cf). succ-imp\ N\ am\ cf)\ (psi, cfi)\ s\ t$

lemma [*sepref-fr-rules*]:
(*uncurry* (*bfsi' N s t*), *uncurry* (*PR-CONST bfs2-op*))
 $\in is-am^k *_a (asmtx-assn\ N\ id-assn)^k \rightarrow_a option-assn\ is-path$
<proof>

lemma [*def-pat-rules*]: *Network.bfs2-op* $\equiv UNPROTECT\ bfs2-op$ *<proof>*
sepref-register *PR-CONST bfs2-op*
 $:: i-ps \Rightarrow capacity-impl\ i-mtx \Rightarrow path\ option\ nres$

schematic-goal *edka-imp-tabulate-impl*:
notes [*sepref-opt-simps*] = *heap-WHILET-def*
fixes *am* :: *node* \Rightarrow *node list* **and** *cf* :: *capacity-impl graph*
notes [*id-rules*] =
itypeI[*Pure.of am TYPE(node* \Rightarrow *node list)*]
notes [*sepref-import-param*] = *IdI*[*of am*]
shows *hn-refine* (*emp*) (?*c*::?'*c Heap*) ? Γ ?*R* (*edka5-tabulate am*)
<proof>

concrete-definition (**in** $-$) *edka-imp-tabulate*
uses *Edka-Impl.edka-imp-tabulate-impl*
prepare-code-thms (**in** $-$) *edka-imp-tabulate-def*

lemma *edka-imp-tabulate-refine*[*sepref-fr-rules*]:
(*edka-imp-tabulate c N*, *PR-CONST edka5-tabulate*)
 $\in (pure\ Id)^k \rightarrow_a prod-assn\ (asmtx-assn\ N\ id-assn)\ is-am$
<proof>

lemma [*def-pat-rules*]:
Network.edka5-tabulate $\equiv UNPROTECT\ edka5-tabulate$
<proof>
sepref-register *PR-CONST edka5-tabulate*
 $:: (node \Rightarrow node\ list) \Rightarrow (capacity-impl\ i-mtx \times i-ps)\ nres$

schematic-goal *edka-imp-run-impl*:
notes [*sepref-opt-simps*] = *heap-WHILET-def*
fixes *am* :: *node* \Rightarrow *node list* **and** *cf* :: *capacity-impl graph*

notes [*id-rules*] =
itypeI[*Pure.of cf TYPE(capacity-impl i-mtx)*]
itypeI[*Pure.of am TYPE(i-ps)*]
shows *hn-refine*
(*hn-ctxt (asmix-assn N id-assn) cf cfi * hn-ctxt is-am am psi*)
(*?c::?'c Heap*) *?Γ ?R*
(*edka5-run cf am*)
 \langle *proof* \rangle

concrete-definition (*in -*) *edka-imp-run* **uses** *Edka-Impl.edka-imp-run-impl*
prepare-code-thms (*in -*) *edka-imp-run-def*

thm *edka-imp-run-def*

lemma *edka-imp-run-refine*[*sepref-fr-rules*]:
(*uncurry (edka-imp-run s t N)*, *uncurry (PR-CONST edka5-run)*)
 \in (*asmix-assn N id-assn*)^{*d*} *_{*a*} (*is-am*)^{*k*} \rightarrow_a *is-rflow N*
 \langle *proof* \rangle

lemma [*def-pat-rules*]:
Network.edka5-run $\$c\$s\$t \equiv$ *UNPROTECT edka5-run*
 \langle *proof* \rangle

sepref-register *PR-CONST edka5-run*
 $::$ *capacity-impl i-mtx* \Rightarrow *i-ps* \Rightarrow *i-rflow nres*

schematic-goal *edka-imp-impl*:

notes [*sepref-opt-simps*] = *heap-WHILET-def*
fixes *am* $::$ *node* \Rightarrow *node list* **and** *cf* $::$ *capacity-impl graph*
notes [*id-rules*] =
itypeI[*Pure.of am TYPE(node* \Rightarrow *node list)*]
notes [*sepref-import-param*] = *IdI*[*of am*]
shows *hn-refine (emp) (?c::?'c Heap) ?Γ ?R (edka5 am)*
 \langle *proof* \rangle

concrete-definition (*in -*) *edka-imp* **uses** *Edka-Impl.edka-imp-impl*
prepare-code-thms (*in -*) *edka-imp-def*
lemmas *edka-imp-refine* = *edka-imp.refine*[*OF this-loc*]

thm *pat-rules TrueI def-pat-rules*

end

export-code *edka-imp checking SML-imp*

5.7 Correctness Theorem for Implementation

We combine all refinement steps to derive a correctness theorem for the implementation

```

context Network-Impl begin
  theorem edka-imp-correct:
    assumes VN: Graph.V c  $\subseteq$  {0.. $N$ }
    assumes ABS-PS: is-adj-map am
    shows
      <emp>
      edka-imp c s t N am
      < $\lambda fi. \exists Af. is-rflow N f fi * \uparrow(isMaxFlow f)$ >_t
    <proof>
  end
end

```

6 Combination with Network Checker

```

theory Edka-Checked-Impl
imports Flow-Networks.NetCheck EdmondsKarp-Impl
begin

```

In this theory, we combine the Edmonds-Karp implementation with the network checker.

6.1 Adding Statistic Counters

We first add some statistic counters, that we use for profiling

```

definition stat-outer-c :: unit Heap where stat-outer-c = return ()

```

```

lemma insert-stat-outer-c: m = stat-outer-c  $\gg$  m

```

```

  <proof>

```

```

definition stat-inner-c :: unit Heap where stat-inner-c = return ()

```

```

lemma insert-stat-inner-c: m = stat-inner-c  $\gg$  m

```

```

  <proof>

```

```

code-printing

```

```

code-module stat  $\rightarrow$  (SML)  $\langle$ 

```

```

  structure stat = struct

```

```

    val outer-c = ref 0;

```

```

    fun outer-c-incr () = (outer-c := !outer-c + 1; ())

```

```

    val inner-c = ref 0;

```

```

    fun inner-c-incr () = (inner-c := !inner-c + 1; ())

```

```

  end

```

```

 $\rangle$ 

```

```

| constant stat-outer-c  $\rightarrow$  (SML) stat.outer'-c'-incr

```

```

| constant stat-inner-c  $\rightarrow$  (SML) stat.inner'-c'-incr

```

```

schematic-goal [code]: edka-imp-run-0 s t N f brk = ?foo

```

```

  <proof>

```

```

thm bfs-impl.code

```

schematic-goal [code]: *bfs-impl-0 succ-impl ci ti x = ?foo*
 ⟨proof⟩

6.2 Combined Algorithm

definition *edmonds-karp el s t* \equiv *do* {
case prepareNet el s t of
None \Rightarrow *return None*
 | *Some (c,am,N)* \Rightarrow *do* {
f \leftarrow *edka-imp c s t N am* ;
return (Some (c,am,N,f))
 }
 }
 }

export-code *edmonds-karp checking SML*

lemma *network-is-impl: Network c s t \impl Network-Impl c s t* ⟨proof⟩

theorem *edmonds-karp-correct:*
 <emp> *edmonds-karp el s t* < λ
None \Rightarrow $\uparrow(\neg \text{ln-invar } el \vee \neg \text{Network } (\text{ln-}\alpha \text{ } el) \text{ } s \text{ } t)$
 | *Some (c,am,N,fi)* \Rightarrow
 $\exists_{Af}. \text{Network-Impl.is-rflow } c \text{ } s \text{ } t \text{ } N \text{ } f \text{ } fi$
 * $\uparrow(\text{ln-}\alpha \text{ } el = c \wedge \text{Graph.is-adj-map } c \text{ } am$
 $\wedge \text{Network.isMaxFlow } c \text{ } s \text{ } t \text{ } f$
 $\wedge \text{ln-invar } el \wedge \text{Network } c \text{ } s \text{ } t \wedge \text{Graph.V } c \subseteq \{0..<N\})$
 >_t
 ⟨proof⟩

context

begin

private definition *is-rflow* \equiv *Network-Impl.is-rflow* **theorem**

fixes *el* **defines** *c* \equiv *ln- α el*

shows

<emp>
edmonds-karp el s t
 < λ *None* \Rightarrow $\uparrow(\neg \text{ln-invar } el \vee \neg \text{Network } c \text{ } s \text{ } t)$
 | *Some (-, -, N, cf)* \Rightarrow
 $\uparrow(\text{ln-invar } el \wedge \text{Network } c \text{ } s \text{ } t \wedge \text{Graph.V } c \subseteq \{0..<N\})$
 * $(\exists_{Af}. \text{is-rflow } c \text{ } s \text{ } t \text{ } N \text{ } f \text{ } cf * \uparrow(\text{Network.isMaxFlow } c \text{ } s \text{ } t \text{ } f))$
 >_t ⟨proof⟩

end

6.3 Usage Example: Computing Maxflow Value

We implement a function to compute the value of the maximum flow.

lemma (in *Network*) *am-s-is-incoming:*

assumes *is-adj-map am*

shows $E'\{s\} = \text{set } (am \text{ } s)$

<proof>

context *RGraph* **begin**

lemma *val-by-adj-map*:

assumes *is-adj-map am*

shows $f.val = (\sum_{v \in set} (am\ s).\ c\ (s,v) - cf\ (s,v))$

<proof>

end

context *Network*

begin

definition *get-cap e* $\equiv c\ e$

definition (**in** $-$) *get-am* $:: (node \Rightarrow node\ list) \Rightarrow node \Rightarrow node\ list$

where *get-am am v* $\equiv am\ v$

definition *compute-flow-val am cf* $\equiv do\ \{$

let succs = get-am am s;

sum-impl

$(\lambda v. do\ \{$

let csv = get-cap (s,v);

cfsv \leftarrow cf-get cf (s,v);

return (csv - cfsv)

$\}) (set\ succs)$

$\}$

lemma (**in** *RGraph*) *compute-flow-val-correct*:

assumes *is-adj-map am*

shows *compute-flow-val am cf* $\leq (spec\ v. v = f.val)$

<proof>

For technical reasons (poor foreach-support of Sepref tool), we have to add another refinement step:

definition *compute-flow-val2 am cf* $\equiv (do\ \{$

let succs = get-am am s;

ifoldli succs $(\lambda-. True)$

$(\lambda x\ a. do\ \{$

b \leftarrow do\ \{

let csv = get-cap (s, x);

cfsv \leftarrow cf-get cf (s, x);

return (csv - cfsv)

$\};$

return (a + b)

$\})$

0

$\})$


```

lemma (in RGraph) compute-flow-val2-correct:
  assumes is-adj-map am
  shows compute-flow-val2 am cf ≤ (spec v. v = f.val)
  ⟨proof⟩

end

context Edka-Impl begin
  term is-am

  lemma [sepref-import-param]: (c, PR-CONST get-cap) ∈ Id ×r Id → Id
  ⟨proof⟩
  lemma [def-pat-rules]:
    Network.get-cap$c ≡ UNPROTECT get-cap ⟨proof⟩
  sepref-register
    PR-CONST get-cap :: node × node ⇒ capacity-impl

  lemma [sepref-import-param]: (get-am, get-am) ∈ Id → Id → ⟨Id⟩list-rel
  ⟨proof⟩

  schematic-goal compute-flow-val-imp:
    fixes am :: node ⇒ node list and cf :: capacity-impl graph
    notes [id-rules] =
      itypeI[Pure.of am TYPE(node ⇒ node list)]
      itypeI[Pure.of cf TYPE(capacity-impl i-mtx)]
    notes [sepref-import-param] = IdI[of N] IdI[of am]
    shows hn-refine
      (hn-ctxt (asmtx-assn N id-assn) cf cfi)
      (?c::?'d Heap) ? $\Gamma$  ? $R$  (compute-flow-val2 am cf)
    ⟨proof⟩
  concrete-definition (in -) compute-flow-val-imp for c s N am cfi
  uses Edka-Impl.compute-flow-val-imp
  prepare-code-thms (in -) compute-flow-val-imp-def
end

context Network-Impl begin

  lemma compute-flow-val-imp-correct-aux:
    assumes VN: Graph.V c ⊆ {0..N}
    assumes ABS-PS: is-adj-map am
    assumes RG: RGraph c s t cf
    shows
      ⟨asmtx-assn N id-assn cf cfi⟩
      compute-flow-val-imp c s N am cfi
      ⟨ $\lambda v. \text{asmtx-assn } N \text{ id-assn } cf \text{ cfi} * \uparrow(v = \text{Flow.val } c \text{ s } (\text{flow-of-cf } cf))$ ⟩t
    ⟨proof⟩

```

lemma *compute-flow-val-imp-correct*:
assumes VN : $Graph.V\ c \subseteq \{0..<N\}$
assumes $ABS-PS$: $Graph.is-adj-map\ c\ am$
shows
 $\langle is-rflow\ N\ f\ cfi \rangle$
 $compute-flow-val-imp\ c\ s\ N\ am\ cfi$
 $\langle \lambda v. is-rflow\ N\ f\ cfi * \uparrow(v = Flow.val\ c\ s\ f) \rangle_t$
 $\langle proof \rangle$

end

definition *edmonds-karp-val* $el\ s\ t \equiv do\ \{$
 $r \leftarrow edmonds-karp\ el\ s\ t;$
 $case\ r\ of$
 $None \Rightarrow return\ None$
 $| Some\ (c,am,N,cfi) \Rightarrow do\ \{$
 $v \leftarrow compute-flow-val-imp\ c\ s\ N\ am\ cfi;$
 $return\ (Some\ v)$
 $\}$
 $\}$

theorem *edmonds-karp-val-correct*:
 $\langle emp \rangle\ edmonds-karp-val\ el\ s\ t\ <\lambda$
 $None \Rightarrow \uparrow(\neg ln-invar\ el \vee \neg Network\ (ln-\alpha\ el)\ s\ t)$
 $| Some\ v \Rightarrow \uparrow(\exists f\ N.$
 $ln-invar\ el \wedge Network\ (ln-\alpha\ el)\ s\ t$
 $\wedge Graph.V\ (ln-\alpha\ el) \subseteq \{0..<N\}$
 $\wedge Network.isMaxFlow\ (ln-\alpha\ el)\ s\ t\ f$
 $\wedge v = Flow.val\ (ln-\alpha\ el)\ s\ f)$
 \rangle_t
 $\langle proof \rangle$

end

7 Conclusion

We have presented a verification of the Edmonds-Karp algorithm, using a stepwise refinement approach. Starting with a proof of the Ford-Fulkerson theorem, we have verified the generic Ford-Fulkerson method, specialized it to the Edmonds-Karp algorithm, and proved the upper bound $O(VE)$ for the number of outer loop iterations. We then conducted several refinement steps to derive an efficiently executable implementation of the algorithm, including a verified breadth first search algorithm to obtain shortest augmenting paths. Finally, we added a verified algorithm to check whether the

input is a valid network, and generated executable code in SML. The runtime of our verified implementation compares well to that of an unverified reference implementation in Java. Our formalization has combined several techniques to achieve an elegant and accessible formalization: Using the Isar proof language [24], we were able to provide a completely rigorous but still accessible proof of the Ford-Fulkerson theorem. The Isabelle Refinement Framework [17, 12] and the Sepref tool [14, 15] allowed us to present the Ford-Fulkerson method on a level of abstraction that closely resembles pseudocode presentations found in textbooks, and then formally link this presentation to an efficient implementation. Moreover, modularity of refinement allowed us to develop the breadth first search algorithm independently, and later link it to the main algorithm. The BFS algorithm can be reused as building block for other algorithms. The data structures are re-usable, too: although we had to implement the array representation of (capacity) matrices for this project, it will be added to the growing library of verified imperative data structures supported by the Sepref tool, such that it can be re-used for future formalizations. During this project, we have learned some lessons on verified algorithm development:

- It is important to keep the levels of abstraction strictly separated. For example, when implementing the capacity function with arrays, one needs to show that it is only applied to valid nodes. However, proving that, e.g., augmenting paths only contain valid nodes is hard at this low level. Instead, one can protect the application of the capacity function by an assertion— already on a high abstraction level where it can be easily discharged. On refinement, this assertion is passed down, and ultimately available for the implementation. Optimally, one wraps the function together with an assertion of its precondition into a new constant, which is then refined independently.
- Profiling has helped a lot in identifying candidates for optimization. For example, based on profiling data, we decided to delay a possible deforestation optimization on augmenting paths, and to first refine the algorithm to operate on residual graphs directly.
- “Efficiency bugs” are as easy to introduce as for unverified software. For example, out of convenience, we implemented the successor list computation by *filter*. Profiling then indicated a hot-spot on this function. As the order of successors does not matter, we invested a bit more work to make the computation tail recursive and gained a significant speed-up. Moreover, we realized only lately that we had accidentally implemented and verified matrices with column major ordering, which have a poor cache locality for our algorithm. Changing the order resulted in another significant speed-up.

We conclude with some statistics: The formalization consists of roughly 8000 lines of proof text, where the graph theory up to the Ford-Fulkerson algorithm requires 3000 lines. The abstract Edmonds-Karp algorithm and its complexity analysis contribute 800 lines, and its implementation (including BFS) another 1700 lines. The remaining lines are contributed by the network checker and some auxiliary theories. The development of the theories required roughly 3 man month, a significant amount of this time going into a first, purely functional version of the implementation, which was later dropped in favor of the faster imperative version.

7.1 Related Work

We are only aware of one other formalization of the Ford-Fulkerson method conducted in Mizar [20] by Lee. Unfortunately, there seems to be no publication on this formalization except [18], which provides a Mizar proof script without any additional comments except that it “defines and proves correctness of Ford/Fulkerson’s Maximum Network-Flow algorithm at the level of graph manipulations”. Moreover, in Lee et al. [19], which is about graph representation in Mizar, the formalization is shortly mentioned, and it is clarified that it does not provide any implementation or data structure formalization. As far as we understood the Mizar proof script, it formalizes an algorithm roughly equivalent to our abstract version of the Ford-Fulkerson method. Termination is only proved for integer valued capacities. Apart from our own work [13, 22], there are several other verifications of graph algorithms and their implementations, using different techniques and proof assistants. Noschinski [23] verifies a checker for (non-)planarity certificates using a bottom-up approach. Starting at a C implementation, the AutoCorres tool [10, 11] generates a monadic representation of the program in Isabelle. Further abstractions are applied to hide low-level details like pointer manipulations and fixed size integers. Finally, a verification condition generator is used to prove the abstracted program correct. Note that their approach takes the opposite direction than ours: While they start at a concrete version of the algorithm and use abstraction steps to eliminate implementation details, we start at an abstract version, and use concretization steps to introduce implementation details.

Charguéraud [4] also uses a bottom-up approach to verify imperative programs written in a subset of OCaml, amongst them a version of Dijkstra’s algorithm: A verification condition generator generates a *characteristic formula*, which reflects the semantics of the program in the logic of the Coq proof assistant [3].

7.2 Future Work

Future work includes the optimization of our implementation, and the formalization of more advanced maximum flow algorithms, like Dinic’s algorithm [6] or push-relabel algorithms [9]. We expect both formalizing the abstract theory and developing efficient implementations to be challenging but realistic tasks.

References

- [1] R.-J. Back. *On the correctness of refinement steps in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.
- [2] R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.
- [3] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq’Art The Calculus of Inductive Constructions*. Springer, 1st edition, 2010.
- [4] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430. ACM, 2011.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [6] Y. Dinitz. Theoretical computer science. chapter Dinitz’ Algorithm: The Original Version and Even’s Version, pages 218–240. Springer, 2006.
- [7] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.
- [8] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.
- [9] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4), Oct. 1988.
- [10] D. Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, CSE, UNSW, Sydney, Australia, mar 2015.
- [11] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *ITP*, pages 99–115. Springer, aug 2012.

- [12] P. Lammich. Refinement for monadic programs. In *Archive of Formal Proofs*. https://isa-afp.org/entries/Refine_Monadic.shtml, 2012. Formal proof development.
- [13] P. Lammich. Verified efficient implementation of Gabows strongly connected component algorithm. In *ITP*, volume 8558 of *LNCS*, pages 325–340. Springer, 2014.
- [14] P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.
- [15] P. Lammich. Refinement based verification of imperative data structures. In *CPP*, pages 27–36. ACM, 2016.
- [16] P. Lammich and S. R. Sefidgar. Formalizing the edmonds-karp algorithm. In *Interactive Theorem Proving*. Springer, 2016. to appear.
- [17] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft’s algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
- [18] G. Lee. Correctness of ford-fulkersons maximum flow algorithm1. *Formalized Mathematics*, 13(2):305–314, 2005.
- [19] G. Lee and P. Rudnicki. Alternative aggregates in mizar. In *Calculemus '07 / MKM '07*, pages 327–341. Springer, 2007.
- [20] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, page 2005, 2005.
- [21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [22] B. Nordhoff and P. Lammich. Formalization of Dijkstra’s algorithm. *Archive of Formal Proofs*, Jan. 2012. https://isa-afp.org/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.
- [23] L. Noschinski. *Formalizing Graph Theory and Planarity Certificates*. PhD thesis, Fakultät für Informatik, Technische Universität München, November 2015.
- [24] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184. Springer, 1999.
- [25] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), Apr. 1971.