Formalizing the Edmonds-Karp Algorithm

Peter Lammich and S. Reza Sefidgar

March 17, 2025

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

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* ≡ *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* ≡ *augment* (*augmentingFlow p*)

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

abbreviation *fofu-invar* ≡ λ(*f*,*brk*).
    NFlow c s t f
  ∧ (*brk* ⟶ (∀ *p*. ¬*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 ≡ do {
  let f₀ = (λ-. 0);

  (f,-) ← while^fofu-invar
    (λ(f,brk). ¬brk)
    (λ(f,-). do {
      p ← find-augmenting-spec f;
      case p of
        None ⇒ return (f,True)
      | Some p ⇒ do {
          assert (p≠[]);
          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)
        }
    })
    (f₀,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* (λ-. *0*)
  ⟨*proof*⟩

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*:
  ¬*isAugmentingPath* []
  ⟨*proof*⟩

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

theorem *fofu-partial-correct*: *fofu* ≤ (*spec f. 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*
  ≡ *NFlow.augment-with-path*
private abbreviation (*input*) *is-augmenting-path f p*
  ≡ *NPreflow.isAugmentingPath c s t f p*

```
definition ford-fulkerson-method ≡ do {
  let f₀ = (λ(u,v). 0);

  (f,brk) ← while (λ(f,brk). ¬brk)
    (λ(f,brk). do {
      p ← select p. is-augmenting-path f p;
      case p of
        None ⇒ return (f,True)
      | Some p ⇒ return (augment c f p, False)
    })
    (f₀,False);
  return f
}
```

end — Anonymous context
end — Network

theorem (**in** *Network*) *ford-fulkerson-method* ≤ (*spec f. isMaxFlow f*)

⟨*proof*⟩

end — Theory

3 Edmonds-Karp Algorithm

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

lemma *mlex-fst-decrI*:
fixes $a\ a'\ b\ b'\ N :: nat$
assumes $a{<}a'$
assumes $b{<}N \quad b'{<}N$
shows $a{*}N + b < a'{*}N + b'$
⟨*proof*⟩

lemma (**in** *NFlow*) *augmenting-path-imp-shortest*:
$isAugmentingPath\ p \Longrightarrow \exists\, p.\ Graph.isShortestPath\ cf\ s\ p\ t$
⟨*proof*⟩

lemma (**in** *NFlow*) *shortest-is-augmenting*:
$Graph.isShortestPath\ cf\ s\ p\ t \Longrightarrow isAugmentingPath\ p$
⟨*proof*⟩

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 $isShortestPath\ s\ p\ t \quad (u,v){\in}set\ p$
assumes $isPath\ s\ p'\ t \quad (v,u){\in}set\ p'$
shows $length\ p' \geq length\ p + 2$
⟨*proof*⟩

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

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

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'*
⟨*proof*⟩

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 ≡ {*e*. ∃ *p*. *e*∈*set p* ∧ *isShortestPath s p t*}

lemma *spEdges-ss-E*: *spEdges* ⊆ *E*
⟨*proof*⟩

lemma *finite-spEdges*[*simp*, *intro*]: *finite* (*spEdges*)
⟨*proof*⟩

definition (**in** *ek-analysis-defs*) *uE* ≡ *E* ∪ E^{-1}

lemma *finite-uE*[*simp*,*intro*]: *finite uE*
⟨*proof*⟩

lemma *E-ss-uE*: *E*⊆*uE*
⟨*proof*⟩

lemma *card-spEdges-le*:
shows *card spEdges* ≤ *card uE*
⟨*proof*⟩

lemma *card-spEdges-less*:
shows *card spEdges* < *card uE* + *1*
⟨*proof*⟩

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

lemma *measure-decr*:
assumes *SV*: *s*∈*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 $ek\text{-}analysis\text{-}defs.ekMeasure\ c'\ s\ t \leq ekMeasure$
and $edges - Graph.E\ c' \neq \{\}$
$\Longrightarrow ek\text{-}analysis\text{-}defs.ekMeasure\ c'\ s\ t < ekMeasure$
⟨*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 $augment\text{-}cf\ edges\ cap \equiv \lambda e.$
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*]: $augment\text{-}cf\ \{\}\ cap = c$
⟨*proof*⟩

lemma *augment-cf-ss-V*: $[\![edges \subseteq E]\!] \Longrightarrow Graph.V\ (augment\text{-}cf\ edges\ cap) \subseteq V$
⟨*proof*⟩

lemma *augment-saturate*:
fixes *edges e*
defines $c' \equiv augment\text{-}cf\ edges\ (c\ e)$
assumes *EIE*: $e \in edges$
shows $e \notin Graph.E\ c'$
⟨*proof*⟩

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

end — Graph

context *NFlow* **begin**

lemma *augmenting-edge-no-swap*: $isAugmentingPath\ p \Longrightarrow set\ p \cap (set\ p)^{-1} = \{\}$
⟨*proof*⟩

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

finite $\{cf\ e\ |e.\ e \in set\ p\}$
⟨*proof*⟩

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

lemma *augment-alt*:
assumes *AUG*: *isAugmentingPath p*
defines $f' \equiv$ *augment* (*augmentingFlow p*)
defines $cf' \equiv$ *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** $e \in set\ p$ $\quad cf\ e = resCap\ p$
⟨*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 $f' \equiv$ *augment* (*augmentingFlow p*)
defines $cf' \equiv$ *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* ≡ *assert* (*NFlow c s t f*) ≫
(*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*]:
$(f',f) \in Id \Longrightarrow$ *find-shortest-augmenting-spec* $f' \leq \Downarrow(\langle Id\rangle$ *option-rel*) (*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 ≡ do {
  let f = (λ-. 0);

  (f,-) ← while^fofu-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
}
```

lemma *edka-partial-refine*[*refine*]: *edka-partial* ≤ ⇓*Id fofu*
⟨*proof*⟩

end — Network

3.2.1 Total Correctness

context *Network* **begin**

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

```
definition edka ≡ do {
  let f = (λ-. 0);
```

```
  (f,-) ← while_T^fofu-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* (λ*cf*. *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* ≤ ⇓*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* $\equiv$ $\{((f,brk,itc), (f,brk)) \mid f\ brk\ itc.$
  $itc + ek\text{-}analysis\text{-}defs.ekMeasure\ (residualGraph\ c\ f)\ s\ t$
 $< 2 * card\ V * card\ E + card\ V$
$\}$

definition *edka-complexity* $\equiv$ *do* {
 let f = (λ-. *0*);

 $(f,\text{-},itc) \leftarrow while_T$
  $(\lambda(f,brk,\text{-}).\ \neg brk)$
  $(\lambda(f,\text{-},itc).$ *do* {
   $p \leftarrow$ *find-shortest-augmenting-spec f*;
   case p of
    None $\Rightarrow$ *return* (*f*,*True*,*itc*)
   | *Some p* $\Rightarrow$ *do* {
     let f = *NFlow.augment-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*
⟨*proof*⟩

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

theorem *edka-complexity* $\leq$ (*spec f*. *isMaxFlow f*)
⟨*proof*⟩

end — Network
end — Theory

4 Breadth First Search

theory *Augmenting-Path-BFS*
imports
 Flow-Networks.Refine-Add-Fofu
 Flow-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 $ndist\ v \equiv min\text{-}dist\ src\ v$

  definition $Vd :: nat \Rightarrow node\ set$
  where
    $\bigwedge d.\ Vd\ d \equiv \{v.\ connected\ src\ v \wedge ndist\ v = d\}$

  lemma $Vd\text{-}disj$: $\bigwedge d\ d'.\ d \neq d' \Longrightarrow Vd\ d \cap Vd\ d' = \{\}$
    ⟨*proof*⟩

  lemma $src\text{-}Vd0[simp]$: $Vd\ 0 = \{src\}$
    ⟨*proof*⟩

  lemma $in\text{-}Vd\text{-}conv$: $v \in Vd\ d \longleftrightarrow connected\ src\ v \wedge ndist\ v = d$
    ⟨*proof*⟩

  lemma $Vd\text{-}succ$:
    assumes $u \in Vd\ d$
    assumes $(u,v) \in E$
    assumes $\forall i \leq d.\ v \notin Vd\ i$
    shows $v \in Vd\ (Suc\ d)$
    ⟨*proof*⟩

end

locale *valid-PRED* = *pre-bfs-invar* +
  fixes $PRED :: node \rightharpoonup node$
  assumes $SRC\text{-}IN\text{-}V[simp]$: $src \in V$
  assumes $FIN\text{-}V[simp, intro!]$: $finite\ V$
  assumes $PRED\text{-}src[simp]$: $PRED\ src = Some\ src$
  assumes $PRED\text{-}dist$: $[\![v \neq src;\ PRED\ v = Some\ u]\!] \Longrightarrow ndist\ v = Suc\ (ndist\ u)$
  assumes $PRED\text{-}E$: $[\![v \neq src;\ PRED\ v = Some\ u]\!] \Longrightarrow (u,v) \in E$
  assumes $PRED\text{-}closed$: $[\![\ PRED\ v = Some\ u\ ]\!] \Longrightarrow u \in dom\ PRED$
begin
  lemma $FIN\text{-}E[simp, intro!]$: $finite\ E$ ⟨*proof*⟩
  lemma $FIN\text{-}succ[simp, intro!]$: $finite\ (E``\{u\})$
    ⟨*proof*⟩
end

locale $nf\text{-}invar' = valid\text{-}PRED\ c\ src\ dst\ PRED$ **for** $c\ src\ dst$
  and $PRED :: node \rightharpoonup node$
  and $C\ N :: node\ set$
  and $d :: nat$
  +
  assumes $VIS\text{-}eq$: $dom\ PRED = N \cup \{u.\ \exists i \leq d.\ u \in Vd\ i\}$

```
  assumes C-ss: C ⊆ Vd d
  assumes N-eq: N = Vd (d+1) ∩ E``(Vd d − C)

  assumes dst-ne-VIS: dst ∉ dom PRED

locale nf-invar = nf-invar' +
  assumes empty-assm: C={} ⟹ N={}

locale f-invar = valid-PRED c src dst PRED for c src dst
  and PRED :: node ⇀ node
  and d :: nat
  +
  assumes dst-found: dst ∈ dom PRED ∩ Vd d

context Graph begin

  abbreviation outer-loop-invar src dst ≡ λ(f,PRED,C,N,d).
    (f ⟶ f-invar c src dst PRED d) ∧
    (¬f ⟶ nf-invar c src dst PRED C N d)


  abbreviation assn1 src dst ≡ λ(f,PRED,C,N,d).
    ¬f ∧ nf-invar' c src dst PRED C N d

definition add-succ-spec dst succ v PRED N ≡ ASSERT (N ⊆ dom PRED) ≫
  SPEC (λ(f,PRED',N').
    case f of
      False ⇒ dst ∉ succ − dom PRED
        ∧ PRED' = map-mmupd PRED (succ − dom PRED) v
        ∧ N' = N ∪ (succ − dom PRED)
    | True ⇒ dst ∈ succ − dom PRED
      ∧ PRED ⊆m PRED'
      ∧ PRED' ⊆m map-mmupd PRED (succ − dom PRED) v
      ∧ dst∈dom PRED'
)

definition pre-bfs :: node ⇒ node ⇒ (nat × (node⇀node)) option nres
  where pre-bfs src dst ≡ do {
  (f,PRED,-,-,d) ← WHILEIT (outer-loop-invar src dst)
    (λ(f,PRED,C,N,d). f=False ∧ C≠{})
    (λ(f,PRED,C,N,d). do {
      v ← SPEC (λv. v∈C); let C = C−{v};
      ASSERT (v∈V);
      let succ = (E``{v});
      ASSERT (finite succ);
      (f,PRED,N) ← add-succ-spec dst succ 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)
    }
  })
  (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]: ⟦v∈C⟧ ⟹ ndist v = d
  ⟨proof⟩
lemma (in nf-invar) CVdI: ⟦u∈C⟧ ⟹ u∈Vd d
  ⟨proof⟩

lemma (in nf-invar) inPREDD:
  ⟦PRED v = Some u⟧ ⟹ v∈N ∨ (∃ i≤d. v∈Vd i)
  ⟨proof⟩

lemma (in nf-invar') C-ss-VIS: ⟦v∈C⟧ ⟹ v∈dom PRED
  ⟨proof⟩

lemma (in nf-invar) invar-succ-step:
  assumes v∈C
  assumes dst ∉ E``{v} − dom PRED
  shows nf-invar' c src dst
    (map-mmupd PRED (E``{v} − dom PRED) v)
    (C−{v})
    (N ∪ (E``{v} − dom PRED))
    d
⟨proof⟩

lemma invar-init: ⟦src≠dst; src∈V; finite V⟧
  ⟹ nf-invar c src dst [src ↦ src] {src} {} 0
  ⟨proof⟩

lemma (in nf-invar) invar-exit:
  assumes dst∈C
  shows f-invar c src dst PRED d
  ⟨proof⟩

lemma (in nf-invar) invar-C-ss-V: u∈C ⟹ u∈V
  ⟨proof⟩
```

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

lemma (**in** *pre-bfs-invar*) *Vdsucinter-conv*[*simp*]:
$Vd\ (Suc\ d) \cap E\ ``\ Vd\ d = Vd\ (Suc\ d)$
⟨*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* ≡ (
case r of None ⇒ ¬ *connected src dst*
| *Some* (*d*,*PRED*) ⇒ *connected src dst*
∧ *min-dist src dst* = *d*
∧ *valid-PRED c src PRED*
∧ *dst*∈*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*: $[\![m{\subseteq_m}m';\ m\ k = Some\ v]\!] \Longrightarrow m'\ k = Some\ v$
⟨*proof*⟩

lemma (**in** *nf-invar*) *dst-notin-Vdd*[*intro*, *simp*]: $i{\le}d \Longrightarrow dst{\notin}\,Vd\ i$
⟨*proof*⟩

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

definition *max-dist src* ≡ *Max* (*min-dist src*‘*V*)

definition *outer-loop-rel src* ≡
  inv-image (
      less-than-bool
      <*lex*> *greater-bounded* (*max-dist src* + *1*)
      <*lex*> *finite-psubset*)
    (λ(*f*,*PRED*,*C*,*N*,*d*). (¬*f*,*d*,*C*))
lemma *outer-loop-rel-wf*:
  assumes *finite V*
  shows *wf* (*outer-loop-rel src*)
  ⟨*proof*⟩

lemma (**in** *nf-invar*) *C-ne-max-dist*:
  assumes *C*≠{}
  shows *d* ≤ *max-dist src*
⟨*proof*⟩

lemma (**in** *nf-invar*) *Vd-ss-V*: *Vd d* ⊆ *V*
  ⟨*proof*⟩

lemma (**in** *nf-invar*) *finite-C*[*simp*, *intro*!]: *finite C*
  ⟨*proof*⟩

lemma (**in** *nf-invar*) *finite-succ*: *finite* (*E*‘‘{*u*})
  ⟨*proof*⟩

theorem *pre-bfs-correct*:
  assumes [*simp*]: *src*∈*V*   *src*≠*dst*
  assumes [*simp*]: *finite V*
  shows *pre-bfs src dst* ≤ *SPEC* (*bfs-spec src dst*)
  ⟨*proof*⟩

definition *bfs-core* :: *node* ⇒ *node* ⇒ (*nat* × (*node*⇀*node*)) *option nres*
  where *bfs-core src dst* ≡ *do* {
  (*f*,*P*,-,-,*d*) ← $while_T$ (λ(*f*,*P*,*C*,*N*,*d*). *f*=*False* ∧ *C*≠{})
    (λ(*f*,*P*,*C*,*N*,*d*). *do* {
      v ← *spec v*. *v*∈*C*; *let C* = *C*−{*v*};
      let succ = (*E*‘‘{*v*});
      (*f*,*P*,*N*) ← *add-succ-spec dst succ v P N*;
      if f then
        return (*f*,*P*,*C*,*N*,*d*+*1*)
      else do {
        if (*C*={}) *then do* {
          let C=*N*; *let N*={}; *let d*=*d*+*1*;
          return (*f*,*P*,*C*,*N*,*d*)
        } *else return* (*f*,*P*,*C*,*N*,*d*)

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

theorem
assumes $src \in V$ $src \neq dst$ *finite* V
shows *bfs-core src dst* $\leq$ (*spec p. bfs-spec src dst p*)
⟨*proof*⟩

4.3 Extraction of Result Path

definition *extract-rpath src dst PRED* $\equiv$ *do* {
(-,p) $\leftarrow$ *WHILEIT*
(λ(v,p).
$v \in$ *dom PRED*
$\wedge$ *isPath v p dst*
$\wedge$ *distinct* (*pathVertices v p*)
$\wedge$ ($\forall$ $v' \in$*set* (*pathVertices v p*).
pre-bfs-invar.ndist c src v $\leq$ *pre-bfs-invar.ndist c src* v')
$\wedge$ *pre-bfs-invar.ndist c src v* + *length p*
= *pre-bfs-invar.ndist c src dst*)
(λ(v,p). $v \neq src$) (λ(v,p). *do* {
ASSERT ($v \in$ *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 $dst \in$ *dom PRED*
shows *extract-rpath src dst PRED*
$\leq$ *SPEC* (λp. *isSimplePath src p dst* $\wedge$ *length p* = *ndist dst*)
⟨*proof*⟩

end

context *Graph* **begin**

definition *bfs src dst* $\equiv$ *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∈V   finite V
  shows bfs src dst
    ≤ SPEC (λ
      None ⇒ ¬connected src dst
    | Some p ⇒ isShortestPath src p dst)
  ⟨proof⟩
end


context Finite-Graph begin
  interpretation Refine-Monadic-Syntax ⟨proof⟩
  theorem
    assumes src∈V
    shows bfs src dst ≤ (spec p. case p of
        None ⇒ ¬connected src dst
      | Some p ⇒ isShortestPath src p dst)
    ⟨proof⟩

end
```

4.4 Inserting inner Loop and Successor Function

```
context Graph begin

definition inner-loop dst succ u PRED N ≡ FOREACHci
  (λit (f,PRED′,N′).
      PRED′ = map-mmupd PRED ((succ − it) − dom PRED) u
    ∧ N′ = N ∪ ((succ − it) − dom PRED)
    ∧ f = (dst∈(succ − it) − dom PRED)
  )
  (succ)
  (λ(f,PRED,N). ¬f)
  (λv (f,PRED,N). do {
    if v∈dom PRED 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-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*
≤ ⇓*Id* (*add-succ-spec dst succ u PRED N*)
⟨*proof*⟩

definition *inner-loop2 dst succl u PRED N* ≡ *nfoldli*
(*succl*) (λ(*f*,-,-). ¬*f*) (λ*v* (*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*)∈⟨*Id*⟩*list-set-rel*
shows *inner-loop2 dst succl u PRED N* ≤ ⇓*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*) ∈ ⟨*Id*⟩*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*
≤ ⇓ *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* ⇒ *bfs-state nres*
where
init-state src ≡ *RETURN* (*False*,[*src*↦*src*],{*src*},{},*0*::*nat*)

definition *pre-bfs2* :: *node* ⇒ *node* ⇒ (*nat* × (*node*⇀*node*)) *option nres*
where *pre-bfs2 src dst* ≡ *do* {

```
s ← init-state src;
(f,PRED,-,-,d) ← WHILET (λ(f,PRED,C,N,d). f=False ∧ C≠{})
  (λ(f,PRED,C,N,d). do {
    ASSERT (C≠{});
    v ← op-set-pick C; let C = C−{v};
    ASSERT (v∈V);
    sl ← succ v;
    (f,PRED,N) ← 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)
    }
  })
  s;
if f then RETURN (Some (d, PRED)) else RETURN None
}
```

lemma *pre-bfs2-refine*:
assumes *succ-impl*: $\bigwedge ui\ u.\ [\![(ui,u)\in Id;\ u\in V]\!]$
$\Longrightarrow$ *succ ui* ≤ *SPEC* (λ*l*. (*l*,*E*‘‘{*u*}) ∈ ⟨*Id*⟩*list-set-rel*)
shows *pre-bfs2 src dst* ≤⇓*Id* (*pre-bfs src dst*)
⟨*proof*⟩

end

definition *bfs2 succ src dst* ≡ *do* {

```
if src=dst then
  RETURN (Some [])
else do {
  br ← pre-bfs2 succ src dst;
  case br of
    None ⇒ RETURN None
  | Some (d,PRED) ⇒ do {
      p ← extract-rpath src dst PRED;
      RETURN (Some p)
```

}
}
}

lemma *bfs2-refine*:
assumes *succ-impl*: $\bigwedge ui\ u.\ [\![(ui,u)\in Id;\ u\in V]\!]$
$\Longrightarrow succ\ ui \leq SPEC\ (\lambda l.\ (l,E``\{u\}) \in \langle Id\rangle list\text{-}set\text{-}rel)$
shows $bfs2\ succ\ src\ dst \leq \Downarrow Id\ (bfs\ src\ dst)$
⟨*proof*⟩

end

lemma *bfs2-refine-succ*:
assumes [*refine*]: $\bigwedge ui\ u.\ [\![(ui,u)\in Id;\ u\in Graph.V\ c]\!]$
$\Longrightarrow succi\ ui \leq \Downarrow Id\ (succ\ u)$
assumes [*simplified*, *simp*]: $(si,s)\in Id \quad (ti,t)\in Id \quad (ci,c)\in Id$
shows $Graph.bfs2\ ci\ succi\ si\ ti \leq \Downarrow Id\ (Graph.bfs2\ c\ succ\ s\ t)$
⟨*proof*⟩

4.5 Imperative Implementation

context *Impl-Succ* **begin**
definition *op-bfs* :: $'ga \Rightarrow node \Rightarrow node \Rightarrow path\ option\ nres$
where [*simp*]: $op\text{-}bfs\ c\ s\ t \equiv Graph.bfs2\ (absG\ c)\ (succ\ c)\ s\ t$

lemma *pat-op-dfs*[*pat-rules*]:
$Graph.bfs2\$(absG\$c)\$(succ\$c)\$s\$t \equiv UNPROTECT\ op\text{-}bfs\$c\$s\$t$ ⟨*proof*⟩

sepref-register *PR-CONST op-bfs*
$:: 'ig \Rightarrow node \Rightarrow node \Rightarrow path\ option\ nres$

type-synonym *ibfs-state*
$= bool \times (node,node)\ i\text{-}map \times node\ set \times node\ set \times nat$

sepref-register $Graph.init\text{-}state :: node \Rightarrow ibfs\text{-}state\ nres$
schematic-goal *init-state-impl*:
fixes $src :: nat$
notes [*id-rules*] =
$itypeI[Pure.of\ src\ TYPE(nat)]$
shows $hn\text{-}refine\ (hn\text{-}val\ nat\text{-}rel\ src\ srci)$
$(?c::?'c\ Heap)\ ?\Gamma'\ ?R\ (Graph.init\text{-}state\ src)$
⟨*proof*⟩
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* ≡ *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* ∧ *NFlow c s t f*}
⟨*proof*⟩

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

lemma *residualGraph-zero-flow*: *residualGraph c* (λ-. *0*) = *c*
⟨*proof*⟩
lemma *flow-of-c*: *flow-of-cf c* = (λ-. *0*)
⟨*proof*⟩

The residual capacity is naturally defined on residual graphs

definition *resCap-cf cf p* ≡ *Min* {*cf e* | *e*. *e*∈*set p*}
lemma (**in** *NFlow*) *resCap-cf-refine*: *resCap-cf cf p* = *resCap p*
⟨*proof*⟩

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*)∈*set p then* (*residualGraph c f* (*u*,*v*) − *resCap p*)
else if (*v*,*u*)∈*set p then* (*residualGraph c f* (*u*,*v*) + *resCap p*)
else residualGraph c f (*u*,*v*))
⟨*proof*⟩

lemma *augment-cf-refine*:
assumes *R*: (*cf*,*f*)∈*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) ∈ *cfi-rel*
⟨*proof*⟩

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

definition *find-shortest-augmenting-spec-cf cf* ≡
assert (*RGraph c s t cf*) ≫
SPEC (λ
None ⇒ ¬*Graph.connected cf s t*
| *Some p* ⇒ *Graph.isShortestPath cf s p t*)

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

```
  ≤ find-shortest-augmenting-spec (flow-of-cf cf)
  ⟨proof⟩
```

This leads to the following refined algorithm

```
definition edka2 ≡ 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);
          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 ≤ ⇓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 ⇒ bool where
  valid-edge ≡ λ(u,v). u∈V ∧ v∈V

definition cf-get
  :: 'capacity graph ⇒ edge ⇒ 'capacity nres
  where cf-get cf e ≡ ASSERT (valid-edge e) ≫ RETURN (cf e)
definition cf-set
  :: 'capacity graph ⇒ edge ⇒ 'capacity ⇒ 'capacity graph nres
  where cf-set cf e cap ≡ ASSERT (valid-edge e) ≫ RETURN (cf(e:=cap))

definition resCap-cf-impl :: 'capacity graph ⇒ path ⇒ 'capacity nres
where resCap-cf-impl cf p ≡
```

```
  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* (λ*r*. *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 ∃ *s′*. *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*   ∀ *u*. *e*≠(*u*,*u*)
  shows *augment-edge-impl cf e cap*
      ≤ (*spec r*. *r* = *Graph.augment-edge cf e cap*)
  ⟨*proof*⟩

definition *augment-cf-impl*
  :: *′capacity graph* ⇒ *path* ⇒ *′capacity* ⇒ *′capacity graph nres*
  where

```
augment-cf-impl cf p x ≡ do {
  (rec_T 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 ∀ e∈set p. valid-edge e
  assumes ∃ s. 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,-) ← while_T
    (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* ≡
  filter-rev (λ*v*. *cf* (*u*,*v*) > *0*) (*am u*)

lemma (**in** *RGraph*) *rg-succ-ref1*: ⟦*is-adj-map am*⟧
  ⟹ (*rg-succ am cf u*, *Graph.E cf*‘‘{*u*}) ∈ ⟨*Id*⟩*list-set-rel*
  ⟨*proof*⟩

definition *ps-get-op* :: - ⇒ *node* ⇒ *node list nres*
  where *ps-get-op am u* ≡ *assert* (*u*∈*V*) ≫ *return* (*am u*)

definition *monadic-filter-rev-aux*
  :: ′*a list* ⇒ (′*a* ⇒ *bool nres*) ⇒ ′*a list* ⇒ ′*a list nres*
where
  monadic-filter-rev-aux a P l ≡ (rec_T *D*. (λ(*l*,*a*). *case l of*
    [] ⇒ *return a*
  | (*v*#*l*) ⇒ *do* {
      c ← *P v*;
      let a = (*if c then v*#*a else a*);
      D (*l*,*a*)
    }
  )) (*l*,*a*)

lemma *monadic-filter-rev-aux-rule*:
  assumes ⋀*x*. *x*∈*set l* ⟹ *P x* ≤ *SPEC* (λ*r*. *r*=*Q x*)
  shows *monadic-filter-rev-aux a P l* ≤ *SPEC* (λ*r*. *r*=*filter-rev-aux a Q l*)
  ⟨*proof*⟩

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

lemma *monadic-filter-rev-rule*:
  assumes ⋀*x*. *x*∈*set l* ⟹ *P x* ≤ (*spec r*. *r*=*Q x*)
  shows *monadic-filter-rev P l* ≤ (*spec r*. *r*=*filter-rev Q l*)
  ⟨*proof*⟩

definition *rg-succ2 am cf u* ≡ *do* {
  l ← *ps-get-op am u*;
  monadic-filter-rev (λ*v*. *do* {
    x ← *cf-get cf* (*u*,*v*);
    return (*x*>*0*)

}) *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}) $\in$ ⟨*Id*⟩*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*,-) $\leftarrow$ $while_T$
(λ(*cf*,*brk*). $\neg$*brk*)
(λ(*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*⊆{*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-nonzero c* = *E* ⟨*proof*⟩

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

lemma [*simp*]: $v \in V \Longrightarrow v < N$ ⟨*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* (*return c*),*uncurry0* (*return c*))∈*unit-assn*^{k} $\rightarrow_a$ *pure* (*nat-rel*$\times_r$*nat-rel* → *int-rel*)
⟨*proof*⟩

5.6.1 Implementation of Adjacency Map by Array

definition *is-am am psi*
≡ $\exists_A l.$ *psi* $\mapsto_a$ *l*
∗ ↑(*length l* = *N* ∧ (∀ *i*<*N*. *l*!*i* = *am i*)
∧ (∀ *i*≥*N*. *am i* = []))

lemma *is-am-precise*[*safe-constraint-rules*]: *precise* (*is-am*)
⟨*proof*⟩

sepref-decl-intf *i-ps* **is** *nat* ⇒ *nat list*

definition (**in** −) *ps-get-imp psi u* ≡ *Array.nth psi u*

lemma [*def-pat-rules*]: *Network.ps-get-op*\$*c* ≡ *UNPROTECT ps-get-op* ⟨*proof*⟩
sepref-register *PR-CONST ps-get-op* :: *i-ps* ⇒ *node* ⇒ *node list nres*

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

lemma *is-pred-succ-no-node*: $[\![$*is-adj-map a*; $u \notin V]\!] \Longrightarrow a\ u = []$
⟨*proof*⟩

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

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

5.6.2 Implementation of Capacity Matrix by Array

lemma [*def-pat-rules*]: *Network.cf-get*\$*c* $\equiv$ *UNPROTECT cf-get* ⟨*proof*⟩
lemma [*def-pat-rules*]: *Network.cf-set*\$*c* $\equiv$ *UNPROTECT cf-set* ⟨*proof*⟩

sepref-register
PR-CONST cf-get :: *capacity-impl i-mtx* $\Rightarrow$ *edge* $\Rightarrow$ *capacity-impl nres*
sepref-register
PR-CONST cf-set :: *capacity-impl i-mtx* $\Rightarrow$ *edge* $\Rightarrow$ *capacity-impl*
$\Rightarrow$ *capacity-impl 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*) :: (*asmtx-assn N id-assn*)$^k *_a$ (*prod-assn id-assn id-assn*)$^k \rightarrow_a$ *id-assn*
⟨*proof*⟩
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*) :: (*asmtx-assn N id-assn*)$^d *_a$ (*prod-assn id-assn id-assn*)$^k *_a$ *id-assn*$^k \rightarrow_a$ *asmtx-assn N id-assn*
⟨*proof*⟩
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*) :: *unit-assn*$^k \rightarrow_a$ *asmtx-assn N id-assn*
⟨*proof*⟩

concrete-definition (**in** −) *init-cf-impl* **uses** *Edka-Impl.init-cf-impl.refine-raw* **is** (*uncurry0 ?f*,-)∈-
prepare-code-thms (**in** −) *init-cf-impl-def*
lemmas [*sepref-fr-rules*] = *init-cf-impl.refine*[*OF this-loc*]

lemma *amtx-cnv*: *amtx-assn N M id-assn* = *IICF-Array-Matrix.is-amtx N M*
⟨*proof*⟩

lemma [*def-pat-rules*]: *Network.init-cf*\$*c* ≡ *UNPROTECT init-cf* ⟨*proof*⟩
sepref-register *PR-CONST init-cf* :: *capacity-impl i-mtx nres*

5.6.3 Representing Result Flow as Residual Graph

definition (**in** *Network-Impl*) *is-rflow N f cfi*
$\equiv \exists_A cf.$ *asmtx-assn N id-assn cf cfi* $*$ ↑(*RGraph c s t cf* ∧ *f* = *flow-of-cf cf*)
lemma *is-rflow-precise*[*safe-constraint-rules*]: *precise* (*is-rflow N*)
⟨*proof*⟩

sepref-decl-intf *i-rflow* **is** *nat*×*nat* ⇒ *int*

lemma [*sepref-fr-rules*]:
(λ*cfi*. *return cfi*, *PR-CONST compute-rflow*) ∈ (*asmtx-assn N id-assn*)$^d \rightarrow_a$ *is-rflow N*
⟨*proof*⟩

lemma [*def-pat-rules*]:
Network.compute-rflow\$*c*\$*s*\$*t* ≡ *UNPROTECT compute-rflow* ⟨*proof*⟩
sepref-register
PR-CONST compute-rflow :: *capacity-impl i-mtx* ⇒ *i-rflow nres*

5.6.4 Implementation of Functions

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

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

lemma [*def-pat-rules*]: *Network.rg-succ2*\$*c* ≡ *UNPROTECT rg-succ2* ⟨*proof*⟩
sepref-register
PR-CONST rg-succ2 :: *i-ps* ⇒ *capacity-impl i-mtx* ⇒ *node* ⇒ *node list nres*

lemma [*sepref-import-param*]: (*min*,*min*)∈*Id*→*Id*→*Id* ⟨*proof*⟩

abbreviation *is-path* ≡ *list-assn* (*prod-assn* (*pure Id*) (*pure Id*))

schematic-goal *resCap-imp-impl*:
fixes *am* :: *node* ⇒ *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* (*asmtx-assn N id-assn*) *cf cfi* ∗ *hn-ctxt is-path p pi*)
(*?c*::*?'c Heap*) *?Γ* *?R*
(*resCap-cf-impl cf p*)
⟨*proof*⟩
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*))
∈ (*asmtx-assn N id-assn*)^k $*_a$ (*is-path*)^k $\rightarrow_a$ (*pure Id*)
⟨*proof*⟩

lemma [*def-pat-rules*]:
Network.resCap-cf-impl$c ≡ *UNPROTECT resCap-cf-impl*
⟨*proof*⟩
sepref-register *PR-CONST resCap-cf-impl*
:: *capacity-impl i-mtx* ⇒ *path* ⇒ *capacity-impl nres*

sepref-thm *augment-imp* **is** *uncurry2* (*PR-CONST augment-cf-impl*) :: ((*asmtx-assn N id-assn*)^d $*_a$ (*is-path*)^k $*_a$ (*pure Id*)^k $\rightarrow_a$ *asmtx-assn N id-assn*)
⟨*proof*⟩
concrete-definition (**in** −) *augment-imp* **uses** *Edka-Impl.augment-imp.refine-raw* **is** (*uncurry2 ?f*,-)∈-
prepare-code-thms (**in** −) *augment-imp-def*
lemma *augment-impl-refine*[*sepref-fr-rules*]:
(*uncurry2* (*augment-imp N*), *uncurry2* (*PR-CONST augment-cf-impl*))
∈ (*asmtx-assn N id-assn*)^d $*_a$ (*is-path*)^k $*_a$ (*pure Id*)^k $\rightarrow_a$ *asmtx-assn N id-assn*
⟨*proof*⟩

lemma [*def-pat-rules*]:
Network.augment-cf-impl$c ≡ *UNPROTECT augment-cf-impl*
⟨*proof*⟩
sepref-register *PR-CONST augment-cf-impl*
:: *capacity-impl i-mtx* ⇒ *path* ⇒ *capacity-impl* ⇒ *capacity-impl i-mtx nres*

sublocale *bfs*: *Impl-Succ*
snd
TYPE(*i-ps* × *capacity-impl i-mtx*)
PR-CONST (λ(*am*,*cf*). *rg-succ2 am cf*)
prod-assn is-am (*asmtx-assn N id-assn*)
λ(*am*,*cf*). *succ-imp N am cf*
⟨*proof*⟩

definition (**in** −) *bfsi′ N s t psi cfi*
≡ *bfs-impl* (λ(*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*\$*c*\$*s*\$*t* ≡ *UNPROTECT bfs2-op* ⟨*proof*⟩
sepref-register *PR-CONST bfs2-op*
:: *i-ps* ⇒ *capacity-impl i-mtx* ⇒ *path option nres*

schematic-goal *edka-imp-tabulate-impl*:
notes [*sepref-opt-simps*] = *heap-WHILET-def*
fixes *am* :: *node* ⇒ *node list* **and** *cf* :: *capacity-impl graph*
notes [*id-rules*] =
itypeI[*Pure.of am TYPE*(*node* ⇒ *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\$*c* ≡ *UNPROTECT edka5-tabulate*
⟨*proof*⟩
sepref-register *PR-CONST edka5-tabulate*
:: (*node* ⇒ *node list*) ⇒ (*capacity-impl i-mtx* × *i-ps*) *nres*

schematic-goal *edka-imp-run-impl*:
notes [*sepref-opt-simps*] = *heap-WHILET-def*
fixes *am* :: *node* ⇒ *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 (asmtx-assn N id-assn) cf cfi ∗ hn-ctxt is-am am psi)
      (?c::?'c Heap) ?Γ ?R
      (edka5-run cf am)
    ⟨proof⟩

  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))
      ∈ (asmtx-assn N id-assn)^d ∗a (is-am)^k →a is-rflow N
    ⟨proof⟩

  lemma [def-pat-rules]:
    Network.edka5-run$c$s$t ≡ UNPROTECT edka5-run
    ⟨proof⟩
  sepref-register PR-CONST edka5-run
    :: capacity-impl i-mtx ⇒ i-ps ⇒ i-rflow nres


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

  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 ⊆ {0..<N}
      assumes ABS-PS: is-adj-map am
      shows
        <emp>
          edka-imp c s t N am
        <λfi. ∃A f. is-rflow N f fi ∗ ↑(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 ≫ m
  ⟨proof⟩
definition stat-inner-c :: unit Heap where stat-inner-c = return ()
lemma insert-stat-inner-c: m = stat-inner-c ≫ m
  ⟨proof⟩

code-printing
  code-module stat ⇀ (SML) ‹
    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
    ›
| constant stat-outer-c ⇀ (SML) stat.outer′-c′-incr
| constant stat-inner-c ⇀ (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* ≡ *do* {
case prepareNet el s t of
None ⇒ *return None*
| *Some* (*c*,*am*,*N*) ⇒ *do* {
f ← *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* ⟹ *Network-Impl c s t* ⟨*proof*⟩

theorem *edmonds-karp-correct*:
<*emp*> *edmonds-karp el s t* <λ
None ⇒ ↑(¬*ln-invar el* ∨ ¬*Network* (*ln-α el*) *s t*)
| *Some* (*c*,*am*,*N*,*fi*) ⇒
$\exists_A f$. *Network-Impl.is-rflow c s t N f fi*
∗ ↑(*ln-α el* = *c* ∧ *Graph.is-adj-map c am*
∧ *Network.isMaxFlow c s t f*
∧ *ln-invar el* ∧ *Network c s t* ∧ *Graph.V c* ⊆ {*0*..<*N*})
$>_t$
⟨*proof*⟩

context
begin
private definition *is-rflow* ≡ *Network-Impl.is-rflow* **theorem**
fixes *el* **defines** *c* ≡ *ln-α el*
shows
<*emp*>
edmonds-karp el s t
<λ *None* ⇒ ↑(¬*ln-invar el* ∨ ¬*Network c s t*)
| *Some* (-,-,*N*,*cf*) ⇒
↑(*ln-invar el* ∧ *Network c s t* ∧ *Graph.V c* ⊆ {*0*..<*N*})
∗ ($\exists_A f$. *is-rflow c s t N f cf* ∗ ↑(*Network.isMaxFlow c s t 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*} = *set* (*am 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* ≡ *c e*
definition (**in** −) *get-am* :: (*node* ⇒ *node list*) ⇒ *node* ⇒ *node list*
where *get-am am v* ≡ *am v*

definition *compute-flow-val am cf* ≡ *do* {
let succs = *get-am am s*;
sum-impl
(λ*v*. *do* {
let csv = *get-cap* (*s*,*v*);
cfsv ← *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* ≤ (*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* ≡ (*do* {
let succs = *get-am am s*;
nfoldli succs (λ-. *True*)
(λ*x a*. *do* {
b ← *do* {
let csv = *get-cap* (*s*, *x*);
cfsv ← *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* $\leq$ (*spec v. v* = *f.val*)
⟨*proof*⟩

end

context *Edka-Impl* **begin**
term *is-am*

lemma [*sepref-import-param*]: $(c, \textit{PR-CONST get-cap}) \in Id \times_r Id \rightarrow Id$
  ⟨*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*) *?Γ* *?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
    <λ*v. asmtx-assn N id-assn cf cfi* ∗ ↑(*v* = *Flow.val c s* (*flow-of-cf cf*))>$_t$
⟨*proof*⟩

lemma *compute-flow-val-imp-correct*:
  assumes *VN*: *Graph.V c* ⊆ {*0..<N*}
  assumes *ABS-PS*: *Graph.is-adj-map c am*
  shows
    <*is-rflow N f cfi*>
      compute-flow-val-imp c s N am cfi
    <λ*v. is-rflow N f cfi* ∗ ↑(*v* = *Flow.val c s f*)>$_t$
  ⟨*proof*⟩

end

definition *edmonds-karp-val el s t* ≡ *do* {
  r ← *edmonds-karp el s t*;
  case r of
    None ⇒ *return None*
  | *Some* (*c*,*am*,*N*,*cfi*) ⇒ *do* {
      v ← *compute-flow-val-imp c s N am cfi*;
      return (*Some v*)
    }
}

theorem *edmonds-karp-val-correct*:
  <*emp*> *edmonds-karp-val el s t* <λ
    None ⇒ ↑(¬*ln-invar el* ∨ ¬*Network* (*ln-*α *el*) *s t*)
  | *Some v* ⇒ ↑(∃*f N*.
        ln-invar el ∧ *Network* (*ln-*α *el*) *s t*
      ∧ *Graph.V* (*ln-*α *el*) ⊆ {*0..<N*}
      ∧ *Network.isMaxFlow* (*ln-*α *el*) *s t f*
      ∧ *v* = *Flow.val* (*ln-*α *el*) *s f*)
      >$_t$
  ⟨*proof*⟩

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. Correctnesss 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.