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)
  apply unfold-locales
  by (auto simp: s-node t-node cap-non-negative)
```

Augmentation preserves the flow property

lemma (**in** *NFlow*) *augment-pres-nflow*:
  assumes *AUG*: *isAugmentingPath p*
  shows *NFlow c s t* (*augment* (*augmentingFlow p*))
proof −
  from *augment-flow-presv*[*OF augFlow-resFlow*[*OF AUG*]]
  interpret *f′*: *Flow c s t augment* (*augmentingFlow p*) **.**
  show *?thesis* **by** *intro-locales*
qed

Augmenting paths cannot be empty

lemma (**in** *NFlow*) *augmenting-path-not-empty*:
  ¬*isAugmentingPath* []
  unfolding *isAugmentingPath-def* **using** *s-not-t* **by** *auto*

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

theorem *fofu-partial-correct*: *fofu* ≤ (*spec f*. *isMaxFlow f*)
  unfolding *fofu-def find-augmenting-spec-def*
  apply (*refine-vcg*)
  apply (*vc-solve simp*:
    zero-flow
    NFlow.augment-pres-nflow
    NFlow.augmenting-path-not-empty
    NFlow.noAugPath-iff-maxFlow[*symmetric*]
    NFlow.augment-with-path-def
)
  done

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 f_0 = (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 −
  have [simp]: (λ(u,v). 0) = (λ-. 0) by auto
  have ford-fulkerson-method ≤ fofu
    unfolding ford-fulkerson-method-def fofu-def Let-def find-augmenting-spec-def
    apply (rule refine-IdD)
    apply (refine-vcg)
    apply (refine-dref-type)
    apply (vc-solve simp: NFlow.augment-with-path-def solve: exI)
    done
  also note fofu-partial-correct
  finally show ?thesis .
qed

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    b'<N
  shows a*N + b < a'*N + b'
proof −
  have a*N + b + 1 ≤ a*N + N using ‹b<N› by linarith
  also have ... ≤ a'*N using ‹a<a'›
    by (metis Suc-leI ab-semigroup-add-class.add.commute
      ab-semigroup-mult-class.mult.commute mult-Suc-right mult-le-mono2)
  also have ... ≤ a'*N + b' by auto
  finally show ?thesis by auto
qed

lemma (in NFlow) augmenting-path-imp-shortest:
  isAugmentingPath p ⟹ ∃ p. Graph.isShortestPath cf s p t
  using Graph.obtain-shortest-path unfolding isAugmentingPath-def
  by (fastforce simp: Graph.isSimplePath-def Graph.connected-def)
```

lemma (**in** *NFlow*) *shortest-is-augmenting*:
Graph.isShortestPath cf s p t $\Longrightarrow$ *isAugmentingPath p*
unfolding *isAugmentingPath-def* **using** *Graph.shortestPath-is-simple*
by (*fastforce*)

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* $(u,v)\in$*set p*
assumes *isPath s p' t* $(v,u)\in$*set p'*
shows *length p'* $\geq$ *length p + 2*
using *assms*
proof −
from ‹*isShortestPath s p t*› **have**
MIN: *min-dist s t = length p* **and**
P: *isPath s p t* **and**
DV: *distinct* (*pathVertices s p*)
by (*auto simp*: *isShortestPath-alt isSimplePath-def*)

from ‹(*u,v*)∈*set p*› **obtain** *p1 p2* **where** [*simp*]: *p=p1@*(*u,v*)#*p2*
by (*auto simp*: *in-set-conv-decomp*)

from *P DV* **have** [*simp*]: *u≠v*
by (*cases p2*) (*auto simp add*: *isPath-append pathVertices-append*)

from *P* **have** *DISTS*: *dist s* (*length p1*) *u* *dist u 1 v* *dist v* (*length p2*) *t*
by (*auto simp*: *isPath-append dist-def intro*: *exI*[**where** *x*=[(*u,v*)]])

from *MIN* **have** *MIN'*: *min-dist s t = length p1 + 1 + length p2* **by** *auto*

from *min-dist-split*[*OF dist-trans*[*OF DISTS*(*1,2*)] *DISTS*(*3*) *MIN'*] **have**
MDSV: *min-dist s v = length p1 + 1* **by** *simp*

```
  from min-dist-split[OF DISTS(1) dist-trans[OF DISTS(2,3)]] MIN' have
    MDUT: min-dist u t = 1 + length p2 by simp

  from ‹(v,u)∈set p'› obtain p1' p2' where [simp]: p'=p1'@(v,u)#p2'
    by (auto simp: in-set-conv-decomp)

  from ‹isPath s p' t› have
    DISTS': dist s (length p1') v    dist u (length p2') t
    by (auto simp: isPath-append dist-def)

  from DISTS'[THEN min-dist-minD, unfolded MDSV MDUT] show
    length p + 2 ≤ length p' by auto
qed
```

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' ⊇ E − edges    Graph.E c' ⊆ E ∪ (prod.swap‘edges)
  assumes SP: isShortestPath s p t and EDGES-SS: edges ⊆ set p
  assumes P': Graph.isPath c' s p' t    prod.swap‘edges ∩ set p' ≠ {}
  shows length p + 2 ≤ length p'
proof −
  interpret g': Graph c' .


  {
    fix u v p1 p2'
    assume (u,v)∈edges
      and isPath s p1 v and g'.isPath u p2' t
    hence min-dist s t < length p1 + length p2'
    proof (induction p2' arbitrary: u v p1 rule: length-induct)
      case (1 p2')
      note IH = 1.IH[rule-format]
      note P1 = ‹isPath s p1 v›
      note P2' = ‹g'.isPath u p2' t›

      have length p1 > min-dist s u
      proof −
        from P1 have length p1 ≥ min-dist s v
          using min-dist-minD by (auto simp: dist-def)
        moreover from ‹(u,v)∈edges› EDGES-SS
        have min-dist s v = Suc (min-dist s u)
          using isShortestPath-level-edge[OF SP] by auto
        ultimately show ?thesis by auto
      qed

      from isShortestPath-level-edge[OF SP] ‹(u,v)∈edges› EDGES-SS
      have
```

```
    min-dist s t = min-dist s u + min-dist u t
  and connected s u
by auto

show ?case
proof (cases prod.swap`edges ∩ set p2' = {})
 — We proceed by a case distinction whether the suffix path contains swapped edges
  case True
  with g'.transfer-path[OF - P2', of c] ‹g'.E ⊆ E ∪ prod.swap ` edges›
  have isPath u p2' t by auto
  hence length p2' ≥ min-dist u t using min-dist-minD
    by (auto simp: dist-def)
  moreover note ‹length p1 > min-dist s u›
  moreover note ‹min-dist s t = min-dist s u + min-dist u t›
  ultimately show ?thesis by auto
next
  case False
  — Obtain first swapped edge on suffix path
  obtain p21' e' p22' where [simp]: p2'=p21'@e'#p22' and
     E-IN-EDGES: e'∈prod.swap`edges and
    P1-NO-EDGES: prod.swap`edges ∩ set p21' = {}
    apply (rule split-list-first-propE[of p2' λe. e∈prod.swap`edges])
    using ‹prod.swap ` edges ∩ set p2' ≠ {}› by fastforce+
  obtain u' v' where [simp]: e'=(v',u') by (cases e')

  — Split the suffix path accordingly
  from P2' have P21': g'.isPath u p21' v' and P22': g'.isPath u' p22' t
    by (auto simp: g'.isPath-append)
  — As we chose the first edge, the prefix of the suffix path is also a path in the original graph
  from
    g'.transfer-path[OF - P21', of c]
    ‹g'.E ⊆ E ∪ prod.swap ` edges›
    P1-NO-EDGES
  have P21: isPath u p21' v' by auto
  from min-dist-is-dist[OF ‹connected s u›]
  obtain psu where
       PSU: isPath s psu u and
    LEN-PSU: length psu = min-dist s u
    by (auto simp: dist-def)
  from PSU P21 have P1n: isPath s (psu@p21') v'
    by (auto simp: isPath-append)
  from IH[OF - - P1n P22'] E-IN-EDGES have
    min-dist s t < length psu + length p21' + length p22'
    by auto
  moreover note ‹length p1 > min-dist s u›
  ultimately show ?thesis by (auto simp: LEN-PSU)
qed
```

```
  qed
} note aux=this


— Obtain first swapped edge on path
obtain p1' e p2' where [simp]: p'=p1'@e#p2' and
  E-IN-EDGES: e∈prod.swap`edges and
  P1-NO-EDGES: prod.swap`edges ∩ set p1' = {}
  apply (rule split-list-first-propE[of p' λe. e∈prod.swap`edges])
  using ‹prod.swap ` edges ∩ set p' ≠ {}› by fastforce+
obtain u v where [simp]: e=(v,u) by (cases e)

— Split the new path accordingly
from ‹g'.isPath s p' t› have
  P1': g'.isPath s p1' v and
  P2': g'.isPath u p2' t
  by (auto simp: g'.isPath-append)
— As we chose the first edge, the prefix of the path is also a path in the original
    graph
from
  g'.transfer-path[OF - P1', of c]
  ‹g'.E ⊆ E ∪ prod.swap ` edges›
  P1-NO-EDGES
have P1: isPath s p1' v by auto

from aux[OF - P1 P2'] E-IN-EDGES
have min-dist s t < length p1' + length p2'
  by auto
thus ?thesis using SP
  by (auto simp: isShortestPath-min-dist-def)
qed

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
  using isPath-edgeset unfolding spEdges-def isShortestPath-def by auto

lemma finite-spEdges[simp, intro]: finite (spEdges)
```

using *finite-subset*[*OF spEdges-ss-E*]
by *blast*

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

lemma *finite-uE*[*simp*,*intro*]: *finite uE*
by (*auto simp*: *uE-def*)

lemma *E-ss-uE*: $E \subseteq uE$
by (*auto simp*: *uE-def*)

lemma *card-spEdges-le*:
shows *card spEdges* $\leq$ *card uE*
apply (*rule card-mono*)
apply (*auto simp*: *order-trans*[*OF spEdges-ss-E E-ss-uE*])
done

lemma *card-spEdges-less*:
shows *card spEdges* $<$ *card uE* + *1*
using *card-spEdges-le*
by *auto*

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'* $\neq \{\}$
$\Longrightarrow$ *ek-analysis-defs.ekMeasure c' s t* $<$ *ekMeasure*
proof $-$
interpret *g'*: *ek-analysis-defs c' s t* **.**

interpret *g'*: *ek-analysis c' s t*
apply *intro-locales*
apply (*rule g'.Finite-Graph-EI*)
using *finite-subset*[*OF Ebounds*(*2*)] *finite-subset*[*OF SP-EDGES*]
by *auto*

from *SP-EDGES SP* **have** $edges \subseteq E$
by (*auto simp*: *spEdges-def isShortestPath-def dest*: *isPath-edgeset*)

with *Ebounds* **have** *Veq*[*simp*]: *Graph.V c′ = V*
by (*force simp*: *Graph.V-def*)

from *Ebounds* ‹*edges* ⊆ *E*› **have** *uE-eq*[*simp*]: *g′.uE = uE*
by (*force simp*: *ek-analysis-defs.uE-def*)

from *SP* **have** *LENP*: *length p = min-dist s t*
by (*auto simp*: *isShortestPath-min-dist-def*)

from *SP* **have** *CONN*: *connected s t*
by (*auto simp*: *isShortestPath-def connected-def*)

{
assume *NCONN2*: *¬g′.connected s t*
hence *s≠t* **by** *auto*
with *CONN NCONN2* **have** *g′.ekMeasure < ekMeasure*
unfolding *g′.ekMeasure-def ekMeasure-def*
using *min-dist-less-V*[*OF SV*]
by *auto*
} moreover {
assume *SHORTER*: *g′.min-dist s t < min-dist s t*
assume *CONN2*: *g′.connected s t*

— Obtain a shorter path in *g′*
from *g′.min-dist-is-dist*[*OF CONN2*] **obtain** *p′* **where**
P′: *g′.isPath s p′ t* **and** *LENP′*: *length p′ = g′.min-dist s t*
by (*auto simp*: *g′.dist-def*)

{ — Case: It does not use *prod.swap ‘ edges*. Then it is also a path in *g*, which is shorter than the shortest path in *g*, yielding a contradiction.
assume *prod.swap‘edges ∩ set p′ = {}*
with *g′.transfer-path*[*OF - P′, of c*] *Ebounds* **have** *dist s* (*length p′*) *t*
by (*auto simp*: *dist-def*)
from *LENP′ SHORTER min-dist-minD*[*OF this*] **have** *False* **by** *auto*
} moreover {
— So assume the path uses the edge *prod.swap e*.
assume *prod.swap‘edges ∩ set p′ ≠ {}*
— Due to auxiliary lemma, those path must be longer
from *isShortestPath-flip-edges*[*OF - - SP SP-EDGES P′ this*] *Ebounds*
have *length p′ > length p* **by** *auto*
with *SHORTER LENP LENP′* **have** *False* **by** *auto*
} ultimately have *False* **by** *auto*
} moreover {
assume *LONGER*: *g′.min-dist s t > min-dist s t*
assume *CONN2*: *g′.connected s t*
have *g′.ekMeasure < ekMeasure*
unfolding *g′.ekMeasure-def ekMeasure-def*
apply (*simp only*: *Veq uE-eq CONN CONN2 if-True*)
apply (*rule mlex-fst-decrI*)

```
    using card-spEdges-less g'.card-spEdges-less
      and g'.min-dist-less-V[OF - CONN2] SV
      and LONGER
    apply auto

    done
  } moreover {
    assume EQ: g'.min-dist s t = min-dist s t
    assume CONN2: g'.connected s t

    {
      fix p'
      assume P': g'.isShortestPath s p' t
      have prod.swap`edges ∩ set p' = {}
      proof (rule ccontr)
        assume EIP': prod.swap`edges ∩ set p' ≠ {}
        from P' have
            P': g'.isPath s p' t and
          LENP': length p' = g'.min-dist s t
          by (auto simp: g'.isShortestPath-min-dist-def)
        from isShortestPath-flip-edges[OF - - SP SP-EDGES P' EIP'] Ebounds
        have length p + 2 ≤ length p' by auto
        with LENP LENP' EQ show False by auto
      qed
      with g'.transfer-path[of p' c s t] P' Ebounds have isShortestPath s p' t
        by (auto simp: Graph.isShortestPath-min-dist-def EQ)
    } hence SS: g'.spEdges ⊆ spEdges by (auto simp: g'.spEdges-def spEdges-def)

    {
      assume edges − Graph.E c' ≠ {}
      with g'.spEdges-ss-E SS SP SP-EDGES have g'.spEdges ⊂ spEdges
        unfolding g'.spEdges-def spEdges-def by fastforce
      hence g'.ekMeasure < ekMeasure
        unfolding g'.ekMeasure-def ekMeasure-def
        apply (simp only: Veq uE-eq EQ CONN CONN2 if-True)
        apply (rule add-strict-left-mono)
        apply (rule psubset-card-mono)
        apply simp
        by simp
    } note G1 = this

    have G2: g'.ekMeasure ≤ ekMeasure
      unfolding g'.ekMeasure-def ekMeasure-def
      apply (simp only: Veq uE-eq CONN CONN2 if-True)
      apply (rule add-mono[OF mult-right-mono])
      apply (simp add: EQ)
      apply simp
      apply (rule card-mono)
      apply simp
```

by *fact*
note *G1 G2*
} ultimately show
$g'.ekMeasure \leq ekMeasure$
$edges - Graph.E\ c' \neq \{\} \Longrightarrow g'.ekMeasure < ekMeasure$
using *less-linear*[*of g'.min-dist s t min-dist s t*]
apply −
apply (*fastforce*)+
done

qed

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* $\{\}\ cap = c$
by (*auto simp*: *augment-cf-def*)

lemma *augment-cf-ss-V*: $[\![edges \subseteq E]\!] \Longrightarrow Graph.V\ (augment\text{-}cf\ edges\ cap) \subseteq V$
unfolding *Graph.E-def Graph.V-def*
by (*auto simp add*: *augment-cf-def*) []

lemma *augment-saturate*:
fixes *edges e*
defines $c' \equiv augment\text{-}cf\ edges\ (c\ e)$
assumes *EIE*: $e \in edges$
shows $e \notin Graph.E\ c'$
using *EIE* **unfolding** *c'-def augment-cf-def*
by (*auto simp*: *Graph.E-def*)

lemma *augment-cf-split*:
assumes $edges1 \cap edges2 = \{\}$ $edges1^{-1} \cap edges2 = \{\}$
shows $Graph.augment\text{-}cf\ c\ (edges1 \cup edges2)\ cap$
$= Graph.augment\text{-}cf\ (Graph.augment\text{-}cf\ c\ edges1\ cap)\ edges2\ cap$
using *assms*
by (*fastforce simp*: *Graph.augment-cf-def intro*!: *ext*)

end — Graph

```
context NFlow begin

lemma augmenting-edge-no-swap: isAugmentingPath p ⟹ set p ∩ (set p)⁻¹ = {}
  using cf.isSPath-nt-parallel-pf
  by (auto simp: isAugmentingPath-def)

lemma aug-flows-finite[simp, intro!]:
  finite {cf e |e. e ∈ set p}
  apply (rule finite-subset[where B=cf`set p])
  by auto

lemma aug-flows-finite'[simp, intro!]:
  finite {cf (u,v) |u v. (u,v) ∈ set p}
  apply (rule finite-subset[where B=cf`set p])
  by auto

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 −
  {
    fix u v
    assume (u,v)∈set p
    hence resCap p ≤ cf (u,v)
      unfolding resCap-def by (auto intro: Min-le)
  } note bn-smallerI = this

  {
    fix u v
    assume (u,v)∈set p
    hence (u,v)∈cf.E using AUG cf.isPath-edgeset
      by (auto simp: isAugmentingPath-def cf.isSimplePath-def)
    hence (u,v)∈E ∨ (v,u)∈E using cfE-ss-invE by (auto)
  } note edge-or-swap = this

  show ?thesis
    apply (rule ext)
    unfolding cf.augment-cf-def
    using augmenting-edge-no-swap[OF AUG]
    apply (auto
      simp: augment-def augmentingFlow-def cf'-def f'-def residualGraph-def
      split: prod.splits
      dest: edge-or-swap
      )
    done
qed
```

lemma *augmenting-path-contains-resCap*:
assumes *isAugmentingPath p*
obtains *e* **where** *e∈set p* *cf e = resCap p*
proof −
from *assms* **have** *p≠[]* **by** (*auto simp*: *isAugmentingPath-def s-not-t*)
hence {*cf e* | *e*. *e* ∈ *set p*} ≠ {} **by** (*cases p*) *auto*
with *Min-in*[*OF aug-flows-finite this*, *folded resCap-def*]
obtain *e* **where** *e∈set p* *cf e = resCap p* **by** *auto*
thus *?thesis* **by** (*blast intro*: *that*)
qed

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 −
interpret *cf*: *ek-analysis cf* **by** *unfold-locales*
interpret *cf′*: *ek-analysis-defs cf′* **.**

from *SP* **have** *AUG*: *isAugmentingPath p*
unfolding *isAugmentingPath-def cf.isShortestPath-alt* **by** *simp*

note *BNGZ* = *resCap-gzero*[*OF AUG*]

have *cf′-alt*: *cf′* = *cf.augment-cf* (*set p*) (*resCap p*)
using *augment-alt*[*OF AUG*] **unfolding** *cf′-def f′-def* **by** *simp*

obtain *e* **where**
EIP: *e∈set p* **and** *EBN*: *cf e* = *resCap p*
by (*rule augmenting-path-contains-resCap*[*OF AUG*]) *auto*

have *ENIE′*: *e∉cf′.E*
using *cf.augment-saturate*[*OF EIP*] *EBN* **by** (*simp add*: *cf′-alt*)

{ fix *e*
have *cf e* + *resCap p* ≠ *0* **using** *resE-nonNegative*[*of e*] *BNGZ* **by** *auto*
} note [*simp*] = *this*

{ fix *e*
assume *e∈set p*
hence *e* ∈ *cf.E*
using *cf.shortestPath-is-path*[*OF SP*] *cf.isPath-edgeset* **by** *blast*
hence *cf e* > *0* ∧ *cf e* ≠ *0* **using** *resE-positive*[*of e*] **by** *auto*
} note [*simp*] = *this*

```
  show ?thesis
    apply (rule cf.measure-decr(2))
    apply (simp-all add: s-node)
    apply (rule SP)
    apply (rule order-refl)

    apply (rule conjI)
      apply (unfold Graph.E-def) []
      apply (auto simp: cf'-alt cf.augment-cf-def) []

      using augmenting-edge-no-swap[OF AUG]
      apply (fastforce
        simp: cf'-alt cf.augment-cf-def Graph.E-def
        simp del: cf.zero-cap-simp) []

    apply (unfold Graph.E-def) []
    apply (auto simp: cf'-alt cf.augment-cf-def) []
    using EIP ENIE' apply auto []
    done
qed

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)∈Id ⟹ find-shortest-augmenting-spec f' ≤ ⇓(⟨Id⟩option-rel) (find-augmenting-spec
f)
```

```
  unfolding find-shortest-augmenting-spec-def find-augmenting-spec-def
  apply (refine-vcg)
  apply (auto
    simp: NFlow.shortest-is-augmenting RELATESI
    dest: NFlow.augmenting-path-imp-shortest)
  done
```

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
  unfolding edka-partial-def fofu-def
  apply (refine-rcg bind-refine')
  apply (refine-dref-type)
  apply (vc-solve simp: find-shortest-augmenting-spec-def)
  done


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 (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
  unfolding edka-wf-rel-def by auto
```

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
  unfolding edka-def edka-partial-def
  apply (refine-rcg bind-refine'
    WHILEIT-refine-WHILEI[where V=edka-wf-rel])
  apply (refine-dref-type)
  apply (simp; fail)
  subgoal
```

Unfortunately, the verification condition for introducing the variant requires a bit of manual massaging to be solved:

```
    apply (simp)
    apply (erule bind-sim-select-rule)
    apply (auto split: option.split
      simp: NFlow.augment-with-path-def
      simp: assert-bind-spec-conv Let-def
      simp: find-shortest-augmenting-spec-def
      simp: edka-wf-rel-def NFlow.shortest-path-decr-ek-measure
```

```
  ; fail) []
 done
```

The other VCs are straightforward

```
 apply (vc-solve)
 done
```

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 −
 interpret NFlow c s t (λ-. 0)
  by unfold-locales (auto simp: s-node t-node cap-non-negative)

 interpret ek: ek-analysis cf
  by unfold-locales auto

 have cardV-positive: card V > 0 and cardE-positive: card E > 0
  using card-0-eq[OF finite-V] V-not-empty apply blast
  using card-0-eq[OF finite-E] E-not-empty apply blast
  done

 show ?thesis proof (cases cf.connected s t)
  case False hence ek.ekMeasure = 0 by (auto simp: ek.ekMeasure-def)
  with cardV-positive cardE-positive show ?thesis
   by auto
 next
  case True

  have cf.min-dist s t > 0
   apply (rule ccontr)
   apply (auto simp: Graph.min-dist-z-iff True s-not-t[symmetric])
   done

  have cf = c
   unfolding residualGraph-def E-def
   by auto
  hence ek.uE = E∪E⁻¹ unfolding ek.uE-def by simp

  from True have ek.ekMeasure
   = (card cf.V − cf.min-dist s t) * (card ek.uE + 1) + (card (ek.spEdges))
   unfolding ek.ekMeasure-def by simp
  also from
```

```
    mlex-bound[of card cf.V − cf.min-dist s t    card V,
              OF - ek.card-spEdges-less]
  have ... < card V * (card ek.uE+1)
    using ‹cf.min-dist s t > 0› ‹card V > 0›
    by (auto simp: resV-netV)
  also have card ek.uE ≤ 2*card E unfolding ‹ek.uE = E∪E⁻¹›
    apply (rule order-trans)
    apply (rule card-Un-le)
    by auto
  finally show ?thesis by (auto simp: algebra-simps)
 qed
qed
```

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 ≡ {((f,brk,itc), (f,brk)) | f brk itc.
   itc + ek-analysis-defs.ekMeasure (residualGraph c f) s t
 < 2 * card V * card E + card V
}

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

 (f,-,itc) ← while_T
   (λ(f,brk,-). ¬brk)
   (λ(f,-,itc). do {
     p ← find-shortest-augmenting-spec f;
     case p of
       None ⇒ return (f,True,itc)
     | Some p ⇒ 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 ≤ ⇓Id edka
proof −
 have [refine-dref-RELATES]:
   RELATES edkac-rel
   by (auto simp: RELATES-def)
```

show *?thesis*
unfolding *edka-complexity-def edka-def*
apply (*refine-rcg*)
apply (*refine-dref-type*)
apply (*vc-solve simp*: *edkac-rel-def NFlow.augment-with-path-def*)
subgoal using *ekMeasure-upper-bound* **by** *auto* []
subgoal by (*drule* (*1*) *NFlow.shortest-path-decr-ek-measure*; *auto*)
done
qed

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

theorem *edka-complexity* ≤ (*spec f*. *isMaxFlow f*)
proof −
note *edka-complexity-refine*
also note *edka-refine*
also note *edka-partial-refine*
also note *fofu-partial-correct*
finally show *?thesis* **.**
qed

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* ≡ *min-dist src v*

definition *Vd* :: *nat* ⇒ *node set*
where
⋀*d*. *Vd d* ≡ {*v*. *connected src v* ∧ *ndist v* = *d*}

lemma *Vd-disj*: $\bigwedge d\ d'.\ d{\neq}d' \Longrightarrow Vd\ d \cap Vd\ d' = \{\}$
by (*auto simp*: *Vd-def*)

lemma *src-Vd0*[*simp*]: $Vd\ 0 = \{src\}$
by (*auto simp*: *Vd-def*)

lemma *in-Vd-conv*: $v{\in}Vd\ d \longleftrightarrow connected\ src\ v \wedge ndist\ v = d$
by (*auto simp*: *Vd-def*)

lemma *Vd-succ*:
assumes $u{\in}Vd\ d$
assumes $(u,v){\in}E$
assumes $\forall\, i{\leq}d.\ v{\notin}Vd\ i$
shows $v{\in}Vd\ (Suc\ d)$
using *assms*
by (*metis connected-append-edge in-Vd-conv le-SucE min-dist-succ*)

end

locale *valid-PRED* = *pre-bfs-invar* +
fixes *PRED* :: $node \rightharpoonup node$
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*: $[\![v{\neq}src;\ PRED\ v = Some\ u]\!] \Longrightarrow ndist\ v = Suc\ (ndist\ u)$
assumes *PRED-E*: $[\![v{\neq}src;\ PRED\ v = Some\ u]\!] \Longrightarrow (u,v){\in}E$
assumes *PRED-closed*: $[\![\ PRED\ v = Some\ u\ ]\!] \Longrightarrow u{\in}dom\ PRED$
begin
lemma *FIN-E*[*simp*, *intro*!]: *finite E* **using** *E-ss-VxV* **by** *simp*
lemma *FIN-succ*[*simp*, *intro*!]: *finite* $(E``\{u\})$
by (*auto intro*: *finite-Image*)
end

locale *nf-invar'* = *valid-PRED c src dst PRED* **for** *c src dst*
and *PRED* :: $node \rightharpoonup node$
and *C N* :: *node set*
and *d* :: *nat*
+
assumes *VIS-eq*: $dom\ PRED = N \cup \{u.\ \exists\, i{\leq}d.\ u{\in}Vd\ i\}$
assumes *C-ss*: $C \subseteq Vd\ d$
assumes *N-eq*: $N = Vd\ (d{+}1) \cap E``(Vd\ d - C)$

assumes *dst-ne-VIS*: $dst \notin dom\ PRED$

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

locale *f-invar* = *valid-PRED c src dst PRED* **for** *c src dst*
and *PRED* :: $node \rightharpoonup node$

```
  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*
using *C-ss* **by** (*auto simp*: *Vd-def*)
lemma (**in** *nf-invar*) *CVdI*: ⟦*u*∈*C*⟧ ⟹ *u*∈*Vd d*
using *C-ss* **by** (*auto*)

lemma (**in** *nf-invar*) *inPREDD*:
⟦*PRED v* = *Some u*⟧ ⟹ *v*∈*N* ∨ (∃ *i*≤*d*. *v*∈*Vd i*)
using *VIS-eq* **by** (*auto*)

lemma (**in** *nf-invar'*) *C-ss-VIS*: ⟦*v*∈*C*⟧ ⟹ *v*∈*dom PRED*
using *C-ss VIS-eq* **by** *blast*

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 −
from *C-ss-VIS*[*OF* ‹*v*∈*C*›] *dst-ne-VIS* **have** *v*≠*dst* **by** *auto*

show *?thesis*
using ‹*v*∈*C*› ‹*v*≠*dst*›
apply *unfold-locales*
apply *simp*
apply *simp*
apply (*auto simp*: *map-mmupd-def*) []

apply (*erule map-mmupdE*)
using *PRED-dist* **apply** *blast*
apply (*unfold VIS-eq*) []
apply *clarify*
apply (*metis CVdI Vd-succ in-Vd-conv*)

using *PRED-E* **apply** (*auto elim*!: *map-mmupdE*) []
using *PRED-closed* **apply** (*auto elim*!: *map-mmupdE dest*: *C-ss-VIS*) []

using *VIS-eq* **apply** *auto* []
using *C-ss* **apply** *auto* []

apply (*unfold N-eq*) []
apply (*frule CVdI*)

apply (*auto*) []
apply (*erule* (*1*) *Vd-succ*)
using *VIS-eq* **apply** (*auto*) []
apply (*auto dest*!: *inPREDD simp*: *N-eq in-Vd-conv*) []

using *dst-ne-VIS assms*(*2*) **apply** *auto* []
done
qed

lemma *invar-init*: ⟦*src*≠*dst*; *src*∈*V*; *finite V*⟧
⟹ *nf-invar c src dst* [*src* ↦ *src*] {*src*} {} *0*
apply *unfold-locales*
apply (*auto*)
apply (*auto simp*: *pre-bfs-invar.Vd-def split*: *if-split-asm*)
done

lemma (**in** *nf-invar*) *invar-exit*:
assumes *dst*∈*C*
shows *f-invar c src dst PRED d*
apply *unfold-locales*
using *assms VIS-eq C-ss* **by** *auto*

lemma (**in** *nf-invar*) *invar-C-ss-V*: *u*∈*C* ⟹ *u*∈*V*
apply (*drule CVdI*)
apply (*auto simp*: *in-Vd-conv connected-inV-iff*)
done

lemma (**in** *nf-invar*) *invar-N-ss-Vis*: *u*∈*N* ⟹ ∃ *v*. *PRED u* = *Some v*
using *VIS-eq* **by** *auto*

lemma (**in** *pre-bfs-invar*) *Vdsucinter-conv*[*simp*]:
Vd (*Suc d*) ∩ *E* `` *Vd d* = *Vd* (*Suc d*)
apply (*auto*)
by (*metis Image-iff in-Vd-conv min-dist-suc*)

lemma (**in** *nf-invar′*) *invar-shift*:
assumes [*simp*]: *C*={}
shows *nf-invar c src dst PRED N* {} (*Suc d*)
apply *unfold-locales*
apply *vc-solve*
using *VIS-eq N-eq*[*simplified*] **apply** (*auto simp add*: *le-Suc-eq*) []
using *N-eq* **apply** *auto* []
using *N-eq*[*simplified*] **apply** *auto* []
using *dst-ne-VIS* **apply** *auto* []
done

lemma (**in** *nf-invar′*) *invar-restore*:
assumes [*simp*]: *C*≠{}
shows *nf-invar c src dst PRED C N d*

```
  apply unfold-locales by auto

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))
  unfolding bfs-spec-def
  apply simp
  using dst-found
  apply (auto simp: in-Vd-conv)
  by unfold-locales

lemma (in nf-invar) invar-not-found:
  assumes [simp]: C={}
  shows bfs-spec src dst None
  unfolding bfs-spec-def
  apply simp
proof (rule notI)
  have [simp]: N={} using empty-assm by simp

  assume C: connected src dst
  then obtain d' where dstd': dst ∈ Vd d'
    by (auto simp: in-Vd-conv)
```

We make a case-distinction whether $d' \leq d$:

```
  have d'≤d ∨ Suc d ≤ d' by auto
  moreover {
    assume d'≤d
    with VIS-eq dstd' have dst ∈ dom PRED by auto
    with dst-ne-VIS have False by auto
  } moreover {
    assume Suc d ≤ d'
```

In the case $d+1 \leq d'$, we also obtain a node that has a shortest path of length $d+1$:

```
      with min-dist-le[OF C] dstd' obtain v' where v' ∈ Vd (Suc d)
        by (auto simp: in-Vd-conv)
```

However, the invariant states that such nodes are either in N or are successors of C. As N and C are both empty, we again get a contradiction.

```
      with N-eq have False by auto
  } ultimately show False by blast
qed
```

lemma *map-le-mp*: $[\![m \subseteq_m m'; m\ k = Some\ v]\!] \Longrightarrow m'\ k = Some\ v$
by (*force simp*: *map-le-def*)

lemma (**in** *nf-invar*) *dst-notin-Vdd*[*intro*, *simp*]: $i \leq d \Longrightarrow dst \notin Vd\ i$
using *VIS-eq dst-ne-VIS* **by** *auto*

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*)
apply *unfold-locales*
apply *simp-all*

using *map-le-mp*[*OF SS1 PRED-src*] **apply** *simp*

apply (*drule map-le-mp*[*OF SS2*])
apply (*erule map-mmupdE*)
using *PRED-dist* **apply** *auto* []
apply (*unfold VIS-eq*) []
apply *clarify*
using ‹$u \in C$›
apply (*metis CVdI Vd-succ in-Vd-conv*)

apply (*drule map-le-mp*[*OF SS2*])
using *PRED-E* **apply** (*auto elim*!: *map-mmupdE*) []

apply (*drule map-le-mp*[*OF SS2*])
apply (*erule map-mmupdE*)
using *map-le-implies-dom-le*[*OF SS1*]
using *PRED-closed* **apply** (*blast*) []
using *C-ss-VIS*[*OF* ‹$u \in C$›] *map-le-implies-dom-le*[*OF SS1*] **apply** *blast*
using ‹$dst \in dom\ PRED'$› **apply** *simp*

using ‹$u \in C$› *CVdI*[*OF* ‹$u \in C$›] ‹$(u,dst) \in E$›
apply (*auto*) []
apply (*erule* (*1*) *Vd-succ*)
using *VIS-eq* **apply** (*auto*) []
done

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)
  using assms
  unfolding outer-loop-rel-def
  by auto

lemma (in nf-invar) C-ne-max-dist:
  assumes C≠{}
  shows d ≤ max-dist src
proof −
  from assms obtain u where u∈C by auto
  with C-ss have u∈Vd d by auto
  hence min-dist src u = d    u∈V
    by (auto simp: in-Vd-conv connected-inV-iff)
  thus d≤max-dist src
    unfolding max-dist-def by auto
qed

lemma (in nf-invar) Vd-ss-V: Vd d ⊆ V
  by (auto simp: Vd-def connected-inV-iff)

lemma (in nf-invar) finite-C[simp, intro!]: finite C
  using C-ss FIN-V Vd-ss-V by (blast intro: finite-subset)

lemma (in nf-invar) finite-succ: finite (E``{u})
  by auto

theorem pre-bfs-correct:
  assumes [simp]: src∈V    src≠dst
  assumes [simp]: finite V
  shows pre-bfs src dst ≤ SPEC (bfs-spec src dst)
  unfolding pre-bfs-def add-succ-spec-def
  apply (intro refine-vcg)
  apply (rule outer-loop-rel-wf[where src=src])
  apply (vc-solve simp:
    invar-init
    nf-invar.invar-exit'
    nf-invar.invar-C-ss-V
    nf-invar.invar-succ-step
    nf-invar'.invar-shift
    nf-invar'.invar-restore
    f-invar.invar-found
    nf-invar.invar-not-found
    nf-invar.invar-N-ss-Vis
    nf-invar.finite-succ
    )
  apply (vc-solve
```

```
    simp: remove-subset outer-loop-rel-def
    simp: nf-invar.C-ne-max-dist nf-invar.finite-C)
  done
```

```
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↦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)
  apply (rule order-trans[OF - pre-bfs-correct])
  apply (rule refine-IdD)
  unfolding bfs-core-def pre-bfs-def
  apply refine-rcg
  apply refine-dref-type
  apply (vc-solve simp: assms)
  done
```

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)
  using assms unfolding extract-rpath-def isSimplePath-def
  apply (refine-vcg wf-measure[where f=λ(d,-). ndist d])
  apply (vc-solve simp: PRED-closed[THEN domD] PRED-E PRED-dist)
  apply auto
  done

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∈V    finite V
  shows bfs src dst
    ≤ SPEC (λ
      None ⇒ ¬connected src dst
    | Some p ⇒ isShortestPath src p dst)
  unfolding bfs-def
  apply (refine-vcg
```

```
    pre-bfs-correct[THEN order-trans]
    valid-PRED.extract-rpath-correct[THEN order-trans]
    )
  using assms
  apply (vc-solve
    simp: bfs-spec-def isShortestPath-min-dist-def isSimplePath-def)
  done
end


context Finite-Graph begin
 interpretation Refine-Monadic-Syntax .
 theorem
  assumes src∈V
  shows bfs src dst ≤ (spec p. case p of
      None ⇒ ¬connected src dst
    | Some p ⇒ isShortestPath src p dst)
  unfolding bfs-def
  apply (refine-vcg
    pre-bfs-correct[THEN order-trans]
    valid-PRED.extract-rpath-correct[THEN order-trans]
    )
  using assms
  apply (vc-solve
    simp: bfs-spec-def isShortestPath-min-dist-def isSimplePath-def)
  done

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)
  unfolding inner-loop-def add-succ-spec-def
  apply refine-vcg
  apply (auto simp: it-step-insert-iff; fail) +
  apply (auto simp: it-step-insert-iff fun-neq-ext-iff map-mmupd-def
    split: if-split-asm) []
  apply (auto simp: it-step-insert-iff split: bool.split; fail)
  apply (auto simp: it-step-insert-iff split: bool.split; fail)
  apply (auto simp: it-step-insert-iff split: bool.split; fail)
  apply (auto simp: it-step-insert-iff intro: map-mmupd-update-less
    split: bool.split; fail)
  done


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)∈⟨Id⟩list-set-rel
 shows inner-loop2 dst succl u PRED N ≤ ⇓Id (inner-loop dst succ u PRED N)
  using assms
  unfolding inner-loop2-def inner-loop-def
  apply (refine-rcg LFOci-refine SR)
  apply (vc-solve)
  apply auto
  done

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*)
apply *simp*
apply (*rule conc-trans*[*OF inner-loop2-refine inner-loop-refine*, *simplified*])
using *assms*(*1*−*2*)
apply (*simp-all*)
done

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: ⋀ui u. ⟦(ui,u)∈Id; u∈V⟧
      ⟹ succ ui ≤ SPEC (λl. (l,E``{u}) ∈ ⟨Id⟩list-set-rel)
    shows pre-bfs2 src dst ≤⇓Id (pre-bfs src dst)
    unfolding pre-bfs-def pre-bfs2-def init-state-def
    apply (subst nres-monad1)
    apply (refine-rcg inner-loop2-correct succ-impl)
    apply refine-dref-type
    apply vc-solve
    done

  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: ⋀ui u. ⟦(ui,u)∈Id; u∈V⟧
      ⟹ succ ui ≤ SPEC (λl. (l,E``{u}) ∈ ⟨Id⟩list-set-rel)
    shows bfs2 succ src dst ≤ ⇓Id (bfs src dst)
    unfolding bfs-def bfs2-def
    apply (refine-vcg pre-bfs2-refine)
    apply refine-dref-type
    using assms
    apply (vc-solve)
    done

end


lemma bfs2-refine-succ:
  assumes [refine]: ⋀ui u. ⟦(ui,u)∈Id; u∈Graph.V c⟧
    ⟹ succi ui ≤ ⇓Id (succ u)
  assumes [simplified, simp]: (si,s)∈Id    (ti,t)∈Id    (ci,c)∈Id
  shows Graph.bfs2 ci succi si ti ≤ ⇓Id (Graph.bfs2 c succ s t)
  unfolding Graph.bfs2-def Graph.pre-bfs2-def
  apply (refine-rcg
    param-nfoldli[param-fo, THEN nres-relD] nres-relI fun-relI)
  apply refine-dref-type
```

apply *vc-solve*
done

4.5 Imperative Implementation

context *Impl-Succ* **begin**
definition *op-bfs* :: *'ga* ⇒ *node* ⇒ *node* ⇒ *path option nres*
where [*simp*]: *op-bfs c s t* ≡ *Graph.bfs2* (*absG c*) (*succ c*) *s t*

lemma *pat-op-dfs*[*pat-rules*]:
Graph.bfs2$(*absG*$*c*)$(*succ*$*c*)$*s*$*t* ≡ *UNPROTECT op-bfs*$*c*$*s*$*t* **by** *simp*

sepref-register *PR-CONST op-bfs*
:: *'ig* ⇒ *node* ⇒ *node* ⇒ *path option nres*

type-synonym *ibfs-state*
= *bool* × (*node*,*node*) *i-map* × *node set* × *node set* × *nat*

sepref-register *Graph.init-state* :: *node* ⇒ *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 srci*)
(*?c*::*?'c Heap*) *?Γ' ?R* (*Graph.init-state src*)
using [[*id-debug*, *goals-limit* = *1*]]
unfolding *Graph.init-state-def*
apply (*rewrite* **in** [*src*↦*src*] *iam.fold-custom-empty*)
apply (*subst ls.fold-custom-empty*)
apply (*subst ls.fold-custom-empty*)
apply (*rewrite* **in** *insert src - fold-set-insert-dj*)
apply (*rewrite* **in** -(¤↦*src*) *fold-COPY*)
apply *sepref*
done
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)
    unfolding op-bfs-def PR-CONST-def
    unfolding Graph.bfs2-def Graph.pre-bfs2-def
      Graph.inner-loop2-def Graph.extract-rpath-def
    unfolding nres-monad-laws
    apply (rewrite in nfoldli - - ⌑ - fold-set-insert-dj)
    apply (subst HOL-list.fold-custom-empty)+
    apply (rewrite in let N={} in - ls.fold-custom-empty)
    using [[id-debug, goals-limit = 1]]
    apply sepref
    done

  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*}
unfolding *cfi-rel-def br-def*
by (*auto*
simp: *NFlow.is-RGraph RGraph.is-NFlow*
simp: *RPreGraph.rg-fo-inv*[*OF RGraph.this-loc-rpg*]
simp: *NPreflow.fo-rg-inv*[*OF NFlow.axioms*(*1*)])

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

lemma *residualGraph-zero-flow*: *residualGraph c* (λ-. *0*) = *c*
unfolding *residualGraph-def* **by** (*auto intro*!: *ext*)
lemma *flow-of-c*: *flow-of-cf c* = (λ-. *0*)
by (*auto simp add*: *flow-of-cf-def*[*abs-def*])

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*
unfolding *resCap-cf-def resCap-def* **..**

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*))
using *augment-alt*[*OF AUG*] **by** (*auto simp*: *Graph.augment-cf-def*)

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 −
from *R* **have** [*simp*]: *cf* = *residualGraph c f* **and** *NFlow c s t f*
by (*auto simp*: *cfi-rel-alt br-def*)
then interpret *f*: *NFlow c s t f* **by** *simp*

show *?thesis*
unfolding *f.augment-with-path-def*
proof (*simp add*: *cfi-rel-alt*; *safe intro*!: *ext*)
fix *u v*

```
    show Graph.augment-cf f.cf (set p) (resCap-cf f.cf p) (u,v)
        = residualGraph c (f.augment (f.augmentingFlow p)) (u,v)
      unfolding f.augment-cf-refine-aux[OF AUG]
      unfolding f.cf.augment-cf-def
      by (auto simp: f.resCap-cf-refine)
  qed (rule f.augment-pres-nflow[OF AUG])
qed
```

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)
  unfolding f-def[symmetric]
  unfolding find-shortest-augmenting-spec-cf-def
    and find-shortest-augmenting-spec-def
  by (auto
    simp: pw-le-iff refine-pw-simps
    simp: this-loc rg-is-cf
    simp: f.isAugmentingPath-def Graph.connected-def Graph.isSimplePath-def
    dest: cf.shortestPath-is-path
    split: option.split)
```

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 −
   have [refine-dref-RELATES]: RELATES cfi-rel by (simp add: RELATES-def)

    show ?thesis
      unfolding edka2-def edka-def

      apply (refine-rcg)
      apply refine-dref-type
      apply vc-solve

      — Solve some left-over verification conditions one by one
      apply (drule NFlow.is-RGraph;
          auto simp: cfi-rel-def br-def residualGraph-zero-flow flow-of-c;
          fail)
      apply (auto simp: cfi-rel-def br-def; fail)
      using RGraph.find-shortest-augmenting-spec-cf-refine
      apply (auto simp: cfi-rel-def br-def; fail)
    apply (auto simp: cfi-rel-def br-def simp: RPreGraph.rg-fo-inv[OF RGraph.this-loc-rpg];
fail)
      apply (drule (1) augment-cf-refine; simp add: cfi-rel-def br-def; fail)
      apply (simp add: augment-cf-refine; fail)
      apply (auto simp: cfi-rel-def br-def; fail)
      apply (auto simp: cfi-rel-def br-def; fail)
      done

  qed
```

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 (λr. r = resCap-cf cf p)
proof −


  note [simp del] = Min-insert
  note [simp] = Min-insert[symmetric]
  from AUG[THEN cf.isSPath-distinct]
  have distinct p .
  moreover from AUG cf.isPath-edgeset have set p ⊆ cf.E
    by (auto simp: cf.isSimplePath-def)
  hence set p ⊆ Collect valid-edge
    using cf.E-ss-VxV by simp
  moreover from AUG have p≠[] by (auto simp: s-not-t)
    then obtain e p' where p=e#p' by (auto simp: neq-Nil-conv)
  ultimately show ?thesis
    unfolding resCap-cf-impl-def resCap-cf-def cf-get-def
    apply (simp only: list.case)
    apply (refine-vcg nfoldli-rule[where
        I = λl l' cap.
          cap = Min (cf‘insert e (set l))
        ∧ set (l@l') ⊆ Collect valid-edge])
    apply (auto intro!: arg-cong[where f=Min])
    done
qed

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 −
  obtain u v where [simp]: e=(u,v) by (cases e)

 from isSPath-no-selfloop[OF P] have [simp]: ⋀u. (u,u)∉set p    u≠v by auto

  from isSPath-nt-parallel[OF P] have [simp]: (v,u)∉set p by auto
  from isSPath-distinct[OF P] have [simp]: (u,v)∉set p by auto

  show augment-cf (insert e (set p)) cap = Graph.augment-cf c' (set p) cap
    apply (rule ext)
    unfolding Graph.augment-cf-def c'-def Graph.augment-edge-def
    by auto

  have Graph.isSimplePath c' v p t
    unfolding Graph.isSimplePath-def
    apply rule
    apply (rule transfer-path)
    unfolding Graph.E-def
    apply (auto simp: c'-def Graph.augment-edge-def) []
    using P apply (auto simp: isSimplePath-def) []
    using P apply (auto simp: isSimplePath-def) []
    done
  thus ∃ s'. Graph.isSimplePath c' s' p t ..

qed

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)
  using assms
  unfolding augment-edge-impl-def Graph.augment-edge-def
  unfolding cf-get-def cf-set-def
  apply refine-vcg
  apply auto
  done
```

```
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}
    apply (simp add: augment-cf-impl-def)
    apply (subst RECT-unfold, refine-mono)
    apply simp

    apply (simp add: augment-cf-impl-def)
    apply (subst RECT-unfold, refine-mono)
    apply simp
    done

  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)
    using assms
    apply (induction p arbitrary: cf)
    apply (simp add: Graph.augment-cf-empty)

    apply clarsimp
    apply (subst Graph.augment-cf-inductive, assumption)

    apply (refine-vcg augment-edge-impl-refine[THEN order-trans])
    apply simp
    apply simp
    apply (auto dest: Graph.isSPath-no-selfloop) []
    apply (rule order-trans, rprems)
      apply (drule Graph.augment-cf-inductive(2)[where cap=x]; simp)
      apply simp
    done
```

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*)
apply (*rule augment-cf-impl-aux*)
using *assms cf.E-ss-VxV* **apply** (*auto simp*: *cf.isSimplePath-def dest*!: *cf.isPath-edgeset*) []
using *assms* **by** *blast*

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* ≤ ⇓*Id edka2*
unfolding *edka3-def edka2-def*
apply (*rewrite* **in** *let cf* = *Graph.augment-cf - - - in - Let-def*)
apply *refine-rcg*
apply *refine-dref-type*
apply (*vc-solve*)
apply (*drule Graph.shortestPath-is-simple*)
apply (*frule* (*1*) *RGraph.resCap-cf-impl-refine*)
apply (*frule* (*1*) *RGraph.augment-cf-impl-refine*)
apply (*auto simp*: *pw-le-iff refine-pw-simps*)
done

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 ≤ find-shortest-augmenting-spec-cf cf
  unfolding find-shortest-augmenting-spec-cf-def
  apply (rule le-ASSERTI)
  apply (rule order-trans)
  apply (rule Graph.bfs-correct)
  apply (simp add: RPreGraph.resV-netV[OF RGraph.this-loc-rpg] s-node)
  apply (simp add: RPreGraph.resV-netV[OF RGraph.this-loc-rpg])
  apply (simp)
  done

lemma edka4-refine: edka4 ≤ ⇓Id edka3
  unfolding edka4-def edka3-def
  apply refine-rcg
  apply refine-dref-type
  apply (vc-solve simp: bfs-refines-shortest-augmenting-spec)
  done
```

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 (λv. 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
  unfolding Graph.E-def
  apply (clarsimp simp: list-set-rel-def br-def rg-succ-def filter-rev-alt;
    intro conjI)
  using cfE-ss-invE resE-nonNegative
  apply (auto
    simp: is-adj-map-def less-le Graph.E-def
    simp del: cf.zero-cap-simp zero-cap-simp) []
  apply (auto simp: is-adj-map-def) []
  done

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 (λr. r=Q x)
  shows monadic-filter-rev-aux a P l ≤ SPEC (λr. r=filter-rev-aux a Q l)
  using assms
  apply (induction l arbitrary: a)

  apply (unfold monadic-filter-rev-aux-def) []
  apply (subst RECT-unfold, refine-mono)
  apply (fold monadic-filter-rev-aux-def) []
```

```
  apply simp

  apply (unfold monadic-filter-rev-aux-def) []
  apply (subst RECT-unfold, refine-mono)
  apply (fold monadic-filter-rev-aux-def) []
  apply (auto simp: pw-le-iff refine-pw-simps)
  done

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)
  using monadic-filter-rev-aux-rule[where a=[]] assms
  by (auto simp: monadic-filter-rev-def filter-rev-def)

definition rg-succ2 am cf u ≡ do {
  l ← ps-get-op am u;
  monadic-filter-rev (λv. 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∈V
  shows rg-succ2 am cf u ≤ return (rg-succ am cf u)
proof −
  have ∀ v∈set (am u). valid-edge (u,v)
    using PS V
    by (auto simp: is-adj-map-def Graph.V-def)

  thus ?thesis
    unfolding rg-succ2-def rg-succ-def ps-get-op-def cf-get-def
    apply (refine-vcg monadic-filter-rev-rule[
        where Q=(λv. 0 < cf (u, v)), THEN order-trans])
    by (vc-solve simp: V)
qed

lemma (in RGraph) rg-succ-ref:
  assumes A: is-adj-map am
  assumes B: u∈V
  shows rg-succ2 am cf u ≤ SPEC (λl. (l,cf.E``{u}) ∈ ⟨Id⟩list-set-rel)
  using rg-succ-ref1[OF A, of u] rg-succ-ref2[OF A B]
  by (auto simp: pw-le-iff refine-pw-simps)
```

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 ≡ RETURN c
definition init-ps :: (node ⇒ node list) ⇒ -
  — Initialization of adjacency map
  where init-ps am ≡ ASSERT (is-adj-map am) ≫ RETURN am

definition compute-rflow :: 'capacity graph ⇒ 'capacity flow nres
  — Extraction of result flow from residual graph
  where
  compute-rflow cf ≡ ASSERT (RGraph c s t cf) ≫ RETURN (flow-of-cf cf)

definition bfs2-op am cf ≡ 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 ≡ do {
  cf ← init-cf;
  am ← init-ps am;
  return (cf,am)
}

definition edka5-run cf am ≡ do {
  (cf,-) ← whileT
    (λ(cf,brk). ¬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
  unfolding edka5-def edka5-tabulate-def edka5-run-def
    edka4-def init-cf-def compute-rflow-def
    init-ps-def Let-def nres-monad-laws bfs2-op-def
  apply refine-rcg
  apply refine-dref-type
  apply (vc-solve simp: )
  apply (rule refine-IdD)
  apply (rule Graph.bfs2-refine)
  apply (simp add: RPreGraph.resV-netV[OF RGraph.this-loc-rpg])
  apply (simp add: RGraph.rg-succ-ref)
  done

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 by unfold-locales

  lemma E-ss: E ⊆ {0..<N}×{0..<N} using E-ss-VxV V-ss by auto
```

lemma *mtx-nonzero-iff*[*simp*]: *mtx-nonzero c* = *E* **unfolding** *E-def* **by** (*auto simp*: *mtx-nonzero-def*)

lemma *mtx-nonzeroN*: *mtx-nonzero c* ⊆ {*0*..<*N*}×{*0*..<*N*} **using** *E-ss* **by** *simp*

lemma [*simp*]: *v*∈*V* ⟹ *v*<*N* **using** *V-ss* **by** *auto*

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} →$_a$ *pure* (*nat-rel*×$_r$*nat-rel* → *int-rel*)
apply *sepref-to-hoare* **by** *sep-auto*

5.6.1 Implementation of Adjacency Map by Array

definition *is-am am psi*
≡ ∃$_A$*l*. *psi* ↦$_a$ *l*
∗ ↑(*length l* = *N* ∧ (∀ *i*<*N*. *l*!*i* = *am i*)
∧ (∀ *i*≥*N*. *am i* = []))

lemma *is-am-precise*[*safe-constraint-rules*]: *precise* (*is-am*)
apply *rule*
unfolding *is-am-def*
apply *clarsimp*
apply (*rename-tac l l'*)
apply *prec-extract-eqs*
apply (*rule ext*)
apply (*rename-tac i*)
apply (*case-tac i*<*length l'*)
apply *fastforce*+
done

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* **by** *simp*
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 →$_a$ *list-assn* (*pure Id*)
unfolding *list-assn-pure-conv*
apply *sepref-to-hoare*
using *V-ss*
by (*sep-auto*
simp: *is-am-def pure-def ps-get-imp-def*
simp: *ps-get-op-def refine-pw-simps*)

lemma *is-pred-succ-no-node*: ⟦*is-adj-map a*; *u*∉*V*⟧ ⟹ *a u* = []
unfolding *is-adj-map-def V-def*
by *auto*

lemma [*sepref-fr-rules*]: (*Array.make N*, *PR-CONST init-ps*)
∈ (*pure Id*)^k →$_a$ *is-am*
apply *sepref-to-hoare*
using *V-ss*
by (*sep-auto simp*: *init-ps-def refine-pw-simps is-am-def pure-def*
intro: *is-pred-succ-no-node*)

lemma [*def-pat-rules*]: *Network.init-ps$c* ≡ *UNPROTECT init-ps* **by** *simp*
sepref-register *PR-CONST init-ps* :: (*node* ⇒ *node list*) ⇒ *i-ps nres*

5.6.2 Implementation of Capacity Matrix by Array

lemma [*def-pat-rules*]: *Network.cf-get$c* ≡ *UNPROTECT cf-get* **by** *simp*
lemma [*def-pat-rules*]: *Network.cf-set$c* ≡ *UNPROTECT cf-set* **by** *simp*

sepref-register
PR-CONST cf-get :: *capacity-impl i-mtx* ⇒ *edge* ⇒ *capacity-impl nres*
sepref-register
PR-CONST cf-set :: *capacity-impl i-mtx* ⇒ *edge* ⇒ *capacity-impl*
⇒ *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 →$_a$ *id-assn*
unfolding *PR-CONST-def cf-get-def*[*abs-def*]
by *sepref*
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*
unfolding *PR-CONST-def cf-set-def*[*abs-def*]
by *sepref*
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*
unfolding *PR-CONST-def init-cf-def*
using *E-ss*
apply (*rewrite op-mtx-new-def*[*of c*, *symmetric*])
apply (*rewrite amtx-fold-custom-new*[*of N N*])
by *sepref*

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*
by (*simp add*: *amtx-assn-def*)

lemma [*def-pat-rules*]: *Network.init-cf*$*c* ≡ *UNPROTECT init-cf* **by** *simp*
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*
≡ ∃$_A$*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*)
apply *rule*
unfolding *is-rflow-def*
apply (*clarsimp simp*: *amtx-assn-def*)
apply *prec-extract-eqs*
apply *simp*
done

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*
unfolding *amtx-cnv*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *amtx-cnv compute-rflow-def is-rflow-def refine-pw-simps*

hn-ctxt-def)
done

lemma [*def-pat-rules*]:
Network.compute-rflow$c$s$t ≡ *UNPROTECT compute-rflow* **by** *simp*
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*)
unfolding *rg-succ2-def APP-def monadic-filter-rev-def monadic-filter-rev-aux-def*

using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*
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 →$_a$ *list-assn* (*pure Id*)
apply *rule*
using *succ-imp.refine*[*OF this-loc*]
by (*auto simp*: *hn-ctxt-def mult-ac split*: *prod.split*)

lemma [*def-pat-rules*]: *Network.rg-succ2$c* ≡ *UNPROTECT rg-succ2* **by** *simp*
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* **by** *simp*

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*)
unfolding *resCap-cf-impl-def APP-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*
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 →$_a$ (*pure Id*)
apply *sepref-to-hnr*
apply (*rule hn-refine-preI*)
apply (*clarsimp*
simp: *uncurry-def list-assn-pure-conv hn-ctxt-def*
split: *prod.split*)
apply (*clarsimp simp*: *pure-def*)
apply (*rule hn-refine-cons*[*OF - resCap-imp.refine*[*OF this-loc*] -])
apply (*simp add*: *list-assn-pure-conv hn-ctxt-def*)
apply (*simp add*: *pure-def*)
apply (*sep-auto simp add*: *hn-ctxt-def pure-def intro*!: *enttI*)
apply (*simp add*: *pure-def*)
done

lemma [*def-pat-rules*]:
Network.resCap-cf-impl$*c* ≡ *UNPROTECT resCap-cf-impl*
by *simp*
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 →$_a$ *asmtx-assn N id-assn*)
unfolding *augment-cf-impl-def*[*abs-def*] *augment-edge-impl-def PR-CONST-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*
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 →$_a$ *asmtx-assn N id-assn*
using *augment-imp.refine*[*OF this-loc*] **.**

lemma [*def-pat-rules*]:
Network.augment-cf-impl$*c* ≡ *UNPROTECT augment-cf-impl*

by *simp*
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*
unfolding *APP-def*
apply *unfold-locales*
apply (*simp add*: *fold-partial-uncurry*)
apply (*rule hfref-cons*[*OF succ-imp-refine*[*unfolded PR-CONST-def*]])
by *auto*

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*))
∈ *is-am*^{k} ∗$_a$ (*asmtx-assn N id-assn*)^k →$_a$ *option-assn is-path*
unfolding *bfsi′-def*[*abs-def*] *bfs2-op-def*[*abs-def*]
using *bfs.bfs-impl-fr-rule* **unfolding** *bfs.op-bfs-def*[*abs-def*]
apply (*clarsimp simp*: *hfref-def all-to-meta*)
apply (*rule hn-refine-cons*[*rotated*])
apply *rprems*
apply (*sep-auto simp*: *pure-def intro*!: *enttI*)
apply (*sep-auto simp*: *pure-def*)
apply (*sep-auto simp*: *pure-def*)
done

lemma [*def-pat-rules*]: *Network.bfs2-op*$*c*$*s*$*t* ≡ *UNPROTECT bfs2-op* **by** *simp*
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*)
unfolding *edka5-tabulate-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*

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*)
∈ (*pure Id*)^k →$_a$ *prod-assn* (*asmtx-assn N id-assn*) *is-am*
apply (*rule*)
apply (*rule hn-refine-preI*)
apply (*clarsimp*
simp: *uncurry-def list-assn-pure-conv hn-ctxt-def*
split: *prod.split*)
apply (*rule hn-refine-cons*[*OF - edka-imp-tabulate.refine*[*OF this-loc*]])
apply (*sep-auto simp*: *hn-ctxt-def pure-def*)+
done

lemma [*def-pat-rules*]:
Network.edka5-tabulate$*c* ≡ *UNPROTECT edka5-tabulate*
by *simp*
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*)
unfolding *edka5-run-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*

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*
apply *rule*
apply (*clarsimp*
simp: *uncurry-def list-assn-pure-conv hn-ctxt-def*
split: *prod.split*)
apply (*rule hn-refine-cons*[*OF - edka-imp-run.refine*[*OF this-loc*] -])
apply (*sep-auto simp*: *hn-ctxt-def*)+

done

lemma [*def-pat-rules*]:
Network.edka5-run$c$s$t ≡ *UNPROTECT edka5-run*
by *simp*
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*)
unfolding *edka5-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*

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
<λ*fi*. ∃ *A f*. *is-rflow N f fi* ∗ ↑(*isMaxFlow f*)>*t*
proof −
interpret *Edka-Impl* **by** *unfold-locales fact*

note *edka5-refine*[*OF ABS-PS*]
also note *edka4-refine*
also note *edka3-refine*

```
    also note edka2-refine
    also note edka-refine
    also note edka-partial-refine
    also note fofu-partial-correct
    finally have edka5 am ≤ SPEC isMaxFlow .
    from hn-refine-ref[OF this edka-imp-refine]
    show ?thesis
      by (simp add: hn-refine-def)
  qed
 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
  unfolding stat-outer-c-def by simp
definition stat-inner-c :: unit Heap where stat-inner-c = return ()
lemma insert-stat-inner-c: m = stat-inner-c ≫ m
  unfolding stat-inner-c-def by simp

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
  apply (subst edka-imp-run.code)
  apply (rewrite in ⌑ insert-stat-outer-c)
  by (rule refl)
```

thm *bfs-impl.code*
schematic-goal [*code*]: *bfs-impl-0 succ-impl ci ti x = ?foo*
apply (*subst bfs-impl.code*)
apply (*rewrite* **in** *imp-nfoldli - - ⌑ - insert-stat-inner-c*)
by (*rule refl*)

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* **by** *intro-locales*

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*) ⇒
∃*_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*
unfolding *edmonds-karp-def*
using *prepareNet-correct*[*of el s t*]
by (*sep-auto*
split: *option.splits*
heap: *Network-Impl.edka-imp-correct*
simp: *ln-rel-def br-def network-is-impl*)

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*})
∗ (∃*_A f*. *is-rflow c s t N f cf* ∗ ↑(*Network.isMaxFlow c s t f*))
>*_t* **unfolding** *c-def is-rflow-def*

by (*sep-auto heap*: *edmonds-karp-correct*[*of el s t*] *split*: *option.split*)

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 $E``\{s\} = set\ (am\ s)$
using *assms no-incoming-s*
unfolding *is-adj-map-def*
by *auto*

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 −
have $f.val = (\sum v{\in}E``\{s\}.\ c\ (s,v) - cf\ (s,v))$
unfolding *f.val-alt*
by (*simp add*: *sum-outgoing-pointwise f-def flow-of-cf-def*)
also have $\ldots = (\sum v{\in}set\ (am\ s).\ c\ (s,v) - cf\ (s,v))$
by (*simp add*: *am-s-is-incoming*[*OF assms*])
finally show *?thesis* **.**
qed

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)
  unfolding val-by-adj-map[OF assms]
  unfolding compute-flow-val-def cf-get-def get-cap-def get-am-def
  apply (refine-vcg sum-impl-correct)
  apply (vc-solve simp: s-node)
  unfolding am-s-is-incoming[symmetric, OF assms]
  by (auto simp: V-def)
```

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)
   (λx 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 ≤ (spec v. v = f.val)
proof −
  have [refine-dref-RELATES]: RELATES (⟨Id⟩list-set-rel)
    by (simp add: RELATES-def)
  show ?thesis
    apply (rule order-trans[OF - compute-flow-val-correct[OF assms]])
    unfolding compute-flow-val2-def compute-flow-val-def sum-impl-def
    apply (rule refine-IdD)
    apply (refine-rcg LFO-refine bind-refine')
    apply refine-dref-type
    apply vc-solve
    using assms
    by (auto
        simp: list-set-rel-def br-def get-am-def is-adj-map-def
        simp: refine-pw-simps)
qed
```

end

context *Edka-Impl* **begin**

```
  term is-am

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

  lemma [sepref-import-param]: (get-am,get-am) ∈ Id → Id → ⟨Id⟩list-rel
    by (auto simp: get-am-def intro!: ext)

  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)
    unfolding compute-flow-val2-def
    using [[id-debug, goals-limit = 1]]
    by sepref
  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
    <λv. asmtx-assn N id-assn cf cfi * ↑(v = Flow.val c s (flow-of-cf cf))>ₜ
proof −
  interpret rg: RGraph c s t cf by fact

  have EI: Edka-Impl c s t N by unfold-locales fact
  from hn-refine-ref[OF
      rg.compute-flow-val2-correct[OF ABS-PS]
      compute-flow-val-imp.refine[OF EI], of cfi]
  show ?thesis
    apply (simp add: hn-ctxt-def pure-def hn-refine-def rg.f-def)
    apply (erule cons-post-rule)
```

```
    apply sep-auto
    done
qed

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
    <λv. is-rflow N f cfi * ↑(v = Flow.val c s f)>t
  apply (rule hoare-triple-preI)
  apply (clarsimp simp: is-rflow-def)
  apply vcg
  apply (rule cons-rule[OF - - compute-flow-val-imp-correct-aux[where cfi=cfi]])
  apply (sep-auto simp: VN ABS-PS)+
  done

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
  unfolding edmonds-karp-val-def
  by (sep-auto
    intro: network-is-impl
    heap: edmonds-karp-correct Network-Impl.compute-flow-val-imp-correct)


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.