Formalizing the Edmonds-Karp Algorithm

Peter Lammich and S. Reza Sefidgar

March 13, 2016

Abstract

We present a formalization of the Ford-Fulkerson method 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 then use stepwise refinement to obtain the Edmonds-Karp algorithm, and formally prove a bound on its complexity. Further refinement yields a verified implementation, whose execution time compares well to an unverified reference implementation in Java.

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 this paper, 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 [20]. Stepwise refinement techniques [24, 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]. Being developed in the Isar [23] proof language, our proofs are accessible even to non-Isabelle experts.

While there exists another formalization of the Ford-Fulkerson method in Mizar [17], 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 paper is a case study on elegantly formalizing algorithms.

2 Flows, Cuts, and Networks

theory *Network*
imports *Graph*
begin

In this theory, we define the basic concepts of flows, cuts, and (flow) networks.

2.1 Definitions

2.1.1 Flows

An *s*-*t* flow on a graph is a labeling of the edges with real values, such that:

capacity constraint the flow on each edge is non-negative and does not exceed the edge's capacity;

conservation constraint for all nodes except *s* and *t*, the incoming flows equal the outgoing flows.

type-synonym $'capacity\ flow = edge \Rightarrow 'capacity$

locale *Flow* = *Graph c* **for** $c :: 'capacity::linordered\text{-}idom\ graph$ +
  fixes $s\ t :: node$
  fixes $f :: 'capacity::linordered\text{-}idom\ flow$

  assumes *capacity-const*: $\forall e.\ 0 \leq f\ e \wedge f\ e \leq c\ e$
  assumes *conservation-const*: $\forall v \in V - \{s, t\}.$
    $(\sum e \in incoming\ v.\ f\ e) = (\sum e \in outgoing\ v.\ f\ e)$
begin

The value of a flow is the flow that leaves s and does not return.

  definition $val :: 'capacity$
    where $val \equiv (\sum e \in outgoing\ s.\ f\ e) - (\sum e \in incoming\ s.\ f\ e)$
end

2.1.2 Cuts

A cut is a partitioning of the nodes into two sets. We define it by just specifying one of the partitions.

type-synonym $cut = node\ set$

locale *Cut* = *Graph* +
  fixes $k :: cut$
  assumes *cut-ss-V*: $k \subseteq V$

2.1.3 Networks

A network is a finite graph with two distinct nodes, source and sink, such that all edges are labeled with positive capacities. Moreover, we assume that

- the source has no incoming edges, and the sink has no outgoing edges
- we allow no parallel edges, i.e., for any edge, the reverse edge must not be in the network
- Every node must lay on a path from the source to the sink

locale *Network* = *Graph c* **for** $c :: 'capacity::linordered\text{-}idom\ graph$ +
  fixes $s\ t :: node$
  assumes *s-node*: $s \in V$
  assumes *t-node*: $t \in V$
  assumes *s-not-t*: $s \neq t$
  assumes *cap-non-negative*: $\forall u\ v.\ c\ (u, v) \geq 0$
  assumes *no-incoming-s*: $\forall u.\ (u, s) \notin E$
  assumes *no-outgoing-t*: $\forall u.\ (t, u) \notin E$
  assumes *no-parallel-edge*: $\forall u\ v.\ (u, v) \in E \longrightarrow (v, u) \notin E$

assumes *nodes-on-st-path*: $\forall v \in V.\ connected\ s\ v \wedge connected\ v\ t$
assumes *finite-reachable*: $finite\ (reachableNodes\ s)$
begin

Our assumptions imply that there are no self loops

lemma *no-self-loop*: $\forall u.\ (u,\ u) \notin E$
$\langle proof \rangle$

A flow is maximal, if it has a maximal value

definition *isMaxFlow* :: - $flow \Rightarrow bool$
where $isMaxFlow\ f \equiv Flow\ c\ s\ t\ f\ \wedge$
$(\forall f'.\ Flow\ c\ s\ t\ f' \longrightarrow Flow.val\ c\ s\ f' \le Flow.val\ c\ s\ f)$

end

2.1.4 Networks with Flows and Cuts

For convenience, we define locales for a network with a fixed flow, and a network with a fixed cut

locale $NFlow = Network\ c\ s\ t + Flow\ c\ s\ t\ f$
for c :: *'capacity::linordered-idom graph* **and** $s\ t\ f$

lemma (**in** *Network*) *isMaxFlow-alt*:
$isMaxFlow\ f \longleftrightarrow NFlow\ c\ s\ t\ f\ \wedge$
$(\forall f'.\ NFlow\ c\ s\ t\ f' \longrightarrow Flow.val\ c\ s\ f' \le Flow.val\ c\ s\ f)$
$\langle proof \rangle$

A cut in a network separates the source from the sink

locale $NCut = Network\ c\ s\ t + Cut\ c\ k$
for c :: *'capacity::linordered-idom graph* **and** $s\ t\ k$ +
assumes *s-in-cut*: $s \in k$
assumes *t-ni-cut*: $t \notin k$
begin

The capacity of the cut is the capacity of all edges going from the source's side to the sink's side.

definition *cap* :: *'capacity*
where $cap \equiv (\sum e \in outgoing'\ k.\ c\ e)$
end

A minimum cut is a cut with minimum capacity.

definition *isMinCut* :: - $graph \Rightarrow nat \Rightarrow nat \Rightarrow cut \Rightarrow bool$
where $isMinCut\ c\ s\ t\ k \equiv NCut\ c\ s\ t\ k\ \wedge$
$(\forall k'.\ NCut\ c\ s\ t\ k' \longrightarrow NCut.cap\ c\ k \le NCut.cap\ c\ k')$

2.2 Properties

2.2.1 Flows

context *Flow*
begin

Only edges are labeled with non-zero flows

lemma *zero-flow-simp*[*simp*]:
  $(u,v) \notin E \Longrightarrow f(u,v) = 0$
  ⟨*proof*⟩

We provide a useful equivalent formulation of the conservation constraint.

lemma *conservation-const-pointwise*:
  assumes $u \in V - \{s,t\}$
  shows $(\sum v \in E``\{u\}.\ f\ (u,v)) = (\sum v \in E^{-1}``\{u\}.\ f\ (v,u))$
  ⟨*proof*⟩

The summation of flows over incoming/outgoing edges can be extended to a summation over all possible predecessor/successor nodes, as the additional flows are all zero.

lemma *sum-outgoing-alt-flow*:
  fixes g :: *edge* ⇒ *'capacity*
  assumes *finite V* $u \in V$
  shows $(\sum e \in outgoing\ u.\ f\ e) = (\sum v \in V.\ f\ (u,v))$
  ⟨*proof*⟩

lemma *sum-incoming-alt-flow*:
  fixes g :: *edge* ⇒ *'capacity*
  assumes *finite V* $u \in V$
  shows $(\sum e \in incoming\ u.\ f\ e) = (\sum v \in V.\ f\ (v,u))$
  ⟨*proof*⟩

end — Flow

2.2.2 Networks

context *Network*
begin

The network constraints implies that all nodes are reachable from the source node

lemma *reachable-is-V*[*simp*]: *reachableNodes s* = *V*
⟨*proof*⟩

This also implies that we have a finite graph, as we assumed a finite set of reachable nodes in the locale definition.

corollary *finite-V*[*simp*, *intro*!]: *finite V*

⟨*proof*⟩

corollary *finite-E*[*simp*, *intro*!]: *finite E*
⟨*proof*⟩

lemma *cap-positive*: $e \in E \Longrightarrow c\ e > 0$
⟨*proof*⟩

lemma *V-not-empty*: $V \neq \{\}$ ⟨*proof*⟩
lemma *E-not-empty*: $E \neq \{\}$ ⟨*proof*⟩

end — Network

2.2.3 Networks with Flow

context *NFlow*
begin

As there are no edges entering the source/leaving the sink, also the corresponding flow values are zero:

lemma *no-inflow-s*: $\forall\, e \in \mathit{incoming}\ s.\ f\ e = 0$ (**is** *?thesis*)
⟨*proof*⟩

lemma *no-outflow-t*: $\forall\, e \in \mathit{outgoing}\ t.\ f\ e = 0$
⟨*proof*⟩

Thus, we can simplify the definition of the value:

corollary *val-alt*: $\mathit{val} = (\sum e \in \mathit{outgoing}\ s.\ f\ e)$
⟨*proof*⟩

For an edge, there is no reverse edge, and thus, no flow in the reverse direction:

lemma *zero-rev-flow-simp*[*simp*]: $(u,v) \in E \Longrightarrow f(v,u) = 0$
⟨*proof*⟩

end — Network with flow

end — Theory

3 Residual Graph

theory *ResidualGraph*
imports *Network*
begin

In this theory, we define the residual graph.

3.1 Definition

The *residual graph* of a network and a flow indicates how much flow can be effectively pushed along or reverse to a network edge, by increasing or decreasing the flow on that edge:

```
definition residualGraph :: - graph ⇒ - flow ⇒ - graph
where residualGraph c f ≡ λ(u, v).
  if (u, v) ∈ Graph.E c then
    c (u, v) − f (u, v)
  else if (v, u) ∈ Graph.E c then
    f (v, u)
  else
    0
```

Let's fix a network with a flow *f* on it

context *NFlow*
begin

We abbreviate the residual graph by *cf*.

abbreviation *cf* ≡ *residualGraph c f*
sublocale *cf*!: *Graph cf* ⟨*proof*⟩
lemmas *cf-def* = *residualGraph-def*[*of c f*]

3.2 Properties

The edges of the residual graph are either parallel or reverse to the edges of the network.

lemma *cfE-ss-invE*: *Graph.E cf* $\subseteq$ *E* $\cup$ E^{-1}
⟨*proof*⟩

The nodes of the residual graph are exactly the nodes of the network.

lemma *resV-netV*[*simp*]: *cf.V* = *V*
⟨*proof*⟩

Note, that Isabelle is powerful enough to prove the above case distinctions completely automatically, although it takes some time:

lemma *cf.V* = *V*
⟨*proof*⟩

As the residual graph has the same nodes as the network, it is also finite:

lemma *finite-cf-incoming*[*simp*, *intro*!]: *finite* (*cf.incoming v*)
⟨*proof*⟩

lemma *finite-cf-outgoing*[*simp*, *intro*!]: *finite* (*cf.outgoing v*)
⟨*proof*⟩

The capacities on the edges of the residual graph are non-negative

lemma *resE-nonNegative*: $cf\ e \geq 0$
⟨*proof*⟩

Again, there is an automatic proof

lemma $cf\ e \geq 0$
  ⟨*proof*⟩

All edges of the residual graph are labeled with positive capacities:

corollary *resE-positive*: $e \in cf.E \Longrightarrow cf\ e > 0$
⟨*proof*⟩

lemma *reverse-flow*: $Flow\ cf\ s\ t\ f' \Longrightarrow \forall (u,\ v) \in E.\ f'\ (v,\ u) \leq f\ (u,\ v)$
⟨*proof*⟩

end — Network with flow

end

4 Augmenting Flows

theory *Augmenting-Flow*
imports *ResidualGraph*
begin

In this theory, we define the concept of an augmenting flow, augmentation with a flow, and show that augmentation of a flow with an augmenting flow yields a valid flow again.

We assume that there is a network with a flow f on it

context *NFlow*
begin

4.1 Augmentation of a Flow

The flow can be augmented by another flow, by adding the flows of edges parallel to edges in the network, and subtracting the edges reverse to edges in the network.

definition *augment* :: *'capacity flow* ⇒ *'capacity flow*
where *augment* $f' \equiv \lambda(u,\ v).$
  if $(u,\ v) \in E$ *then*
    $f\ (u,\ v) + f'\ (u,\ v) - f'\ (v,\ u)$
  else
    0

We define a syntax similar to Cormen et el.:

abbreviation (*input*) *augment-syntax* (**infix** $\uparrow$ *55*)
  where $\bigwedge f\ f'.\ f\uparrow f' \equiv$ *NFlow.augment c f f'*

such that we can write $f\uparrow f'$ for the flow f augmented by f'.

4.2 Augmentation yields Valid Flow

We show that, if we augment the flow with a valid flow of the residual graph, the augmented flow is a valid flow again, i.e. it satisfies the capacity and conservation constraints:

context
  — Let the *residual flow* f' be a flow in the residual graph
  fixes f' :: *'capacity flow*
  assumes *f'-flow*: *Flow cf s t f'*
begin

interpretation *f'!*: *Flow cf s t f'* ⟨*proof*⟩

4.2.1 Capacity Constraint

First, we have to show that the new flow satisfies the capacity constraint:

lemma *augment-flow-presv-cap*:
  shows $0 \le (f\uparrow f')(u,v) \wedge (f\uparrow f')(u,v) \le c(u,v)$
⟨*proof*⟩ **lemma** *split-rflow-incoming*:
  $(\sum v{\in}cf.E^{-1}``\{u\}.\ f'\ (v,u)) = (\sum v{\in}E``\{u\}.\ f'(v,u)) + (\sum v{\in}E^{-1}``\{u\}.\ f'(v,u))$
  (**is** *?LHS = ?RHS*)
⟨*proof*⟩

For proving the conservation constraint, let's fix a node u, which is neither the source nor the sink:

context
  fixes u :: *node*
  assumes *U-ASM*: $u{\in}V - \{s,t\}$
begin

We first show an auxiliary lemma to compare the effective residual flow on incoming network edges to the effective residual flow on outgoing network edges.

Intuitively, this lemma shows that the effective residual flow added to the network edges satisfies the conservation constraint.

private lemma *flow-summation-aux*:
  shows $(\sum v{\in}E``\{u\}.\ f'\ (u,v)) - (\sum v{\in}E``\{u\}.\ f'\ (v,u))$
      $= (\sum v{\in}E^{-1}``\{u\}.\ f'\ (v,u)) - (\sum v{\in}E^{-1}``\{u\}.\ f'\ (u,v))$
   (**is** *?LHS = ?RHS* **is** *?A − ?B = ?RHS*)
⟨*proof*⟩

Finally, we are ready to prove that the augmented flow satisfies the conservation constraint:

lemma *augment-flow-presv-con*:
  shows $(\sum e \in outgoing\ u.\ augment\ f'\ e) = (\sum e \in incoming\ u.\ augment\ f'\ e)$
    (**is** *?LHS* = *?RHS*)
⟨*proof*⟩

Note that we tried to follow the proof presented by Cormen et al. [5] as closely as possible. Unfortunately, this proof generalizes the summation to all nodes immediately, rendering the first equation invalid. Trying to fix this error, we encountered that the step that uses the conservation constraints on the augmenting flow is more subtle as indicated in the original proof. Thus, we moved this argument to an auxiliary lemma.

end — u is node

As main result, we get that the augmented flow is again a valid flow.

corollary *augment-flow-presv*: $Flow\ c\ s\ t\ (f \uparrow f')$
  ⟨*proof*⟩

4.3 Value of the Augmented Flow

Next, we show that the value of the augmented flow is the sum of the values of the original flow and the augmenting flow.

lemma *augment-flow-value*: $Flow.val\ c\ s\ (f \uparrow f') = val + Flow.val\ cf\ s\ f'$
⟨*proof*⟩

end — Augmenting flow
end — Network flow

end — Theory

5 Augmenting Paths

theory *Augmenting-Path*
imports *ResidualGraph*
begin

We define the concept of an augmenting path in the residual graph, and the residual flow induced by an augmenting path.

We fix a network with a flow f on it.

context *NFlow*
begin

5.1 Definitions

An *augmenting path* is a simple path from the source to the sink in the residual graph:

definition *isAugmenting* :: *path* ⇒ *bool*
where *isAugmenting p* ≡ *cf.isSimplePath s p t*

The *residual capacity* of an augmenting path is the smallest capacity annotated to its edges:

definition *bottleNeck* :: *path* ⇒ *'capacity*
where *bottleNeck p* ≡ *Min* {*cf e* | *e. e* ∈ *set p*}

lemma *bottleNeck-alt*: *bottleNeck p* = *Min* (*cf'set p*)
  — Useful characterization for finiteness arguments
  ⟨*proof*⟩

An augmenting path induces an *augmenting flow*, which pushes as much flow as possible along the path:

definition *augmentingFlow* :: *path* ⇒ *'capacity flow*
where *augmentingFlow p* ≡ λ(*u*, *v*).
  if (*u*, *v*) ∈ (*set p*) *then*
    bottleNeck p
  else
    0

5.2 Augmenting Flow is Valid Flow

In this section, we show that the augmenting flow induced by an augmenting path is a valid flow in the residual graph.

We start with some auxiliary lemmas.

The residual capacity of an augmenting path is always positive.

lemma *bottleNeck-gzero-aux*: *cf.isPath s p t* ⟹ *0*<*bottleNeck p*
⟨*proof*⟩

lemma *bottleNeck-gzero*: *isAugmenting p* ⟹ *0*<*bottleNeck p*
  ⟨*proof*⟩

As all edges of the augmenting flow have the same value, we can factor this out from a summation:

lemma *setsum-augmenting-alt*:
  assumes *finite A*
  shows (∑ *e* ∈ *A*. (*augmentingFlow p*) *e*)
      = *bottleNeck p* ∗ *of-nat* (*card* (*A*∩*set p*))
⟨*proof*⟩

lemma *augFlow-resFlow*: *isAugmenting p* $\Longrightarrow$ *Flow cf s t* (*augmentingFlow p*)
⟨*proof*⟩

5.3 Value of Augmenting Flow is Residual Capacity

Finally, we show that the value of the augmenting flow is the residual capacity of the augmenting path

lemma *augFlow-val*:
  isAugmenting p $\Longrightarrow$ *Flow.val cf s* (*augmentingFlow p*) = *bottleNeck p*
⟨*proof*⟩

end — Network with flow
end — Theory

6 The Ford-Fulkerson Theorem

theory *Ford-Fulkerson*
imports *Augmenting-Flow Augmenting-Path*
begin

In this theory, we prove the Ford-Fulkerson theorem, and its well-known corollary, the min-cut max-flow theorem.

We fix a network with a flow and a cut

locale *NFlowCut* = *NFlow c s t f* + *NCut c s t k*
  for *c* :: *'capacity*::*linordered-idom graph* **and** *s t f k*
begin

6.1 Net Flow

We define the *net flow* to be the amount of flow effectively passed over the cut from the source to the sink:

definition *netFlow* :: *'capacity*
  where *netFlow* $\equiv$ ($\sum$ *e* $\in$ *outgoing' k*. *f e*) − ($\sum$ *e* $\in$ *incoming' k*. *f e*)

We can show that the net flow equals the value of the flow. Note: Cormen et al. [5] present a whole page full of summation calculations for this proof, and our formal proof also looks quite complicated.

lemma *flow-value*: *netFlow* = *val*
⟨*proof*⟩

The value of any flow is bounded by the capacity of any cut. This is intuitively clear, as all flow from the source to the sink has to go over the cut.

corollary *weak-duality*: *val* $\leq$ *cap*
⟨*proof*⟩

end — Cut

6.2 Ford-Fulkerson Theorem

context *NFlow* **begin**

We prove three auxiliary lemmas first, and the state the theorem as a corollary

lemma *fofu-I-II*: $isMaxFlow\ f \Longrightarrow \neg\ (\exists\ p.\ isAugmenting\ p)$
$\langle proof \rangle$

lemma *fofu-II-III*:
  $\neg\ (\exists\ p.\ isAugmenting\ p) \Longrightarrow \exists\, k'.\ NCut\ c\ s\ t\ k' \wedge val = NCut.cap\ c\ k'$
$\langle proof \rangle$

lemma *fofu-III-I*:
  $\exists\, k.\ NCut\ c\ s\ t\ k \wedge val = NCut.cap\ c\ k \Longrightarrow isMaxFlow\ f$
$\langle proof \rangle$

Finally we can state the Ford-Fulkerson theorem:

theorem *ford-fulkerson*: **shows**
  $isMaxFlow\ f \longleftrightarrow$
  $\neg\ Ex\ isAugmenting$ **and** $\neg\ Ex\ isAugmenting \longleftrightarrow$
  $(\exists\, k.\ NCut\ c\ s\ t\ k \wedge val = NCut.cap\ c\ k)$
  $\langle proof \rangle$

6.3 Corollaries

In this subsection we present a few corollaries of the flow-cut relation and the Ford-Fulkerson theorem.

The outgoing flow of the source is the same as the incoming flow of the sink. Intuitively, this means that no flow is generated or lost in the network, except at the source and sink.

lemma *inflow-t-outflow-s*: $(\sum e \in incoming\ t.\ f\ e) = (\sum e \in outgoing\ s.\ f\ e)$
$\langle proof \rangle$

As an immediate consequence of the Ford-Fulkerson theorem, we get that there is no augmenting path if and only if the flow is maximal.

lemma *noAugPath-iff-maxFlow*: $\neg\ (\exists\ p.\ isAugmenting\ p) \longleftrightarrow isMaxFlow\ f$
  $\langle proof \rangle$

end — Network with flow

The value of the maximum flow equals the capacity of the minimum cut

lemma (**in** *Network*) *maxFlow-minCut*: $[\![isMaxFlow\ f;\ isMinCut\ c\ s\ t\ k]\!]$

```
  ⟹ Flow.val c s f = NCut.cap c k
⟨proof⟩

end — Theory
```

7 The Ford-Fulkerson Method

```
theory FordFulkerson-Algo
imports
  Ford-Fulkerson
  Refine-Add-Fofu
  Refine-Monadic-Syntax-Sugar
begin
```

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

```
context Network
begin
```

7.1 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);
    selectp p. NFlow.isAugmenting c s t f 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. ¬NFlow.isAugmenting 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 f = (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 (NFlow.isAugmenting c s t f p);
        let f' = NFlow.augmentingFlow c f p;
        let f = NFlow.augment c f f';
        assert (NFlow c s t f);
        return (f, False)
      }
    })
    (f,False);
  assert (NFlow c s t f);
  return f
}
```

7.2 Partial Correctness

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

The zero flow is a valid flow

lemma *zero-flow*: *NFlow c s t* (λ-. *0*)
  ⟨*proof*⟩

Augmentation preserves the flow property

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

Augmenting paths cannot be empty

lemma (**in** *NFlow*) *augmenting-path-not-empty*:
  ¬*isAugmenting* []
  ⟨*proof*⟩

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

theorem *fofu-partial-correct*: *fofu* ≤ (*spec f*. *isMaxFlow f*)
  ⟨*proof*⟩

7.3 Algorithm without Assertions

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

definition (**in** *NFlow*) *augment-with-path p* ≡ *augment* (*augmentingFlow p*)

context begin

private abbreviation (*input*) *augment*
  ≡ *NFlow.augment-with-path*

private abbreviation (*input*) *is-augmenting-path f p*
  ≡ *NFlow.isAugmenting c s t f p*

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

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

end — Anonymous context
end — Network

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

⟨*proof*⟩

end — Theory

8 Edmonds-Karp Algorithm

theory *EdmondsKarp-Algo*
imports *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.

8.1 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*) ≫
  SELECTp (λ*p. Graph.isShortestPath* (*residualGraph c f*) *s p t*)

Note, if there is an augmenting path, there is always a shortest one

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

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

We show that our refined procedure is actually a refinement

lemma *find-shortest-augmenting-refine*[*refine*]:
  $(f',f) \in Id \Longrightarrow$ *find-shortest-augmenting-spec* $f' \leq \Downarrow Id$ (*find-augmenting-spec* f)
  ⟨*proof*⟩

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

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

  (f,-) ← while^{fofu-invar}
    (λ(f,brk). ¬brk)
    (λ(f,-). do {
      p ← find-shortest-augmenting-spec f;
      case p of
        None ⇒ return (f,True)
      | Some p ⇒ do {
          assert (p≠[]);
          assert (NFlow.isAugmenting c s t f p);
          assert (Graph.isShortestPath (residualGraph c f) s p t);
          let f' = NFlow.augmentingFlow c f p;
          let f = NFlow.augment c f f';
          assert (NFlow c s t f);
          return (f, False)
        }
    })
    (f,False);
  assert (NFlow c s t f);
  return f
}
```

lemma *edka-partial-refine*[*refine*]: *edka-partial* $\leq \Downarrow Id$ *fofu*
  ⟨*proof*⟩

end — Network

8.2 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 $length\ p' \geq length\ p + 2$
  ⟨*proof*⟩

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

lemma *isShortestPath-flip-edges*:
  assumes $Graph.E\ c' \supseteq E - edges$ $Graph.E\ c' \subseteq E \cup (prod.swap`edges)$
  assumes *SP*: *isShortestPath s p t* **and** *EDGES-SS*: $edges \subseteq set\ p$
  assumes *P′*: *Graph.isPath c′ s p′ t* $prod.swap`edges \cap set\ p' \neq \{\}$
  shows $length\ p + 2 \leq length\ p'$
⟨*proof*⟩

end — Graph

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

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

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

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

lemma *spEdges-ss-E*: $spEdges \subseteq E$
  ⟨*proof*⟩

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

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

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

lemma *E-ss-uE*: $E \subseteq uE$
  ⟨*proof*⟩

lemma *card-spEdges-le*:
  shows *card spEdges* $\leq$ *card uE*
  ⟨*proof*⟩

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

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*⟩

end — Analysis locale

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

context *Graph*
begin

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

lemma *augment-cf-empty*[*simp*]: *augment-cf* {} *cap* $=$ *c*
  ⟨*proof*⟩

lemma *augment-cf-ss-V*: ⟦*edges* $\subseteq$ *E*⟧ $\Longrightarrow$ *Graph.V* (*augment-cf edges cap*) $\subseteq$ *V*

⟨*proof*⟩

lemma *augment-saturate*:
  fixes *edges e*
  defines $c' \equiv$ *augment-cf edges* (*c e*)
  assumes *EIE*: *e*∈*edges*
  shows *e*∉*Graph.E c'*
  ⟨*proof*⟩

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

end — Graph

context *NFlow* **begin**

lemma *augmenting-edge-no-swap*: *isAugmenting p* ⟹ *set p* ∩ $(set\ p)^{-1}$ = {}
  ⟨*proof*⟩

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

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

lemma *augment-alt*:
  assumes *AUG*: *isAugmenting p*
  defines $f' \equiv$ *augment* (*augmentingFlow p*)
  defines $cf' \equiv$ *residualGraph c f'*
  shows *cf'* = *Graph.augment-cf cf* (*set p*) (*bottleNeck p*)
⟨*proof*⟩

lemma *augmenting-path-contains-bottleneck*:
  assumes *isAugmenting p*
  obtains *e* **where** *e*∈*set p*   *cf e* = *bottleNeck p*
⟨*proof*⟩

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

theorem *shortest-path-decr-ek-measure*:
  fixes *p*

```
  assumes SP: Graph.isShortestPath cf s p t
  defines f' ≡ augment (augmentingFlow p)
  defines cf' ≡ residualGraph c f'
  shows ek-analysis-defs.ekMeasure cf' s t < ek-analysis-defs.ekMeasure cf s t
⟨proof⟩
```

end — Network with flow

8.2.1 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 (NFlow.isAugmenting c s t f p);
          assert (Graph.isShortestPath (residualGraph c f) s p t);
          let f' = NFlow.augmentingFlow c f p;
          let f = NFlow.augment c f f';
          assert (NFlow c s t f);
          return (f, False)
        }
    })
    (f,False);
  assert (NFlow c s t f);
  return f
}
```

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

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

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

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

theorem *edka-refine*[*refine*]: *edka* ≤ ⇓*Id edka-partial*

⟨*proof*⟩

8.2.2 Complexity Analysis

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

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

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

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

```
definition edkac-rel ≡ {((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.augmentingFlow c f p;
          let f = NFlow.augment c f f';
          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⟩
```

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

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

end — Network
end — Theory

9 Implementation of the Edmonds-Karp Algorithm

theory *EdmondsKarp-Impl*
imports
  EdmondsKarp-Algo
  Augmenting-Path-BFS
  Capacity-Matrix-Impl
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.

9.1 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.

definition (**in** *Network*) *flow-of-cf cf e* $\equiv$ (*if* $(e \in E)$ *then c e* $-$ *cf e else 0*)

lemma (**in** *NFlow*) *E-ss-cfinvE*: $E \subseteq$ *Graph.E cf* $\cup$ (*Graph.E cf*)$^{-1}$
  ⟨*proof*⟩

locale *RGraph* — Locale that characterizes a residual graph of a network
= *Network* +
  fixes *cf*
  assumes *EX-RG*: $\exists f$. *NFlow c s t f* $\wedge$ *cf* = *residualGraph c f*
begin

  lemma *this-loc*: *RGraph c s t cf*
    ⟨*proof*⟩

  definition *f* $\equiv$ *flow-of-cf cf*

  lemma *f-unique*:

  assumes *NFlow c s t f'*
  assumes *A*: *cf* = *residualGraph c f'*
  shows *f'* = *f*
⟨*proof*⟩

lemma *is-NFlow*: *NFlow c s t* (*flow-of-cf cf*)
  ⟨*proof*⟩

sublocale *f*!: *NFlow c s t f* ⟨*proof*⟩

lemma *rg-is-cf*[*simp*]: *residualGraph c f* = *cf*
  ⟨*proof*⟩

lemma *rg-fo-inv*[*simp*]: *residualGraph c* (*flow-of-cf cf*) = *cf*
  ⟨*proof*⟩

sublocale *cf*!: *Graph cf* ⟨*proof*⟩

lemma *resV-netV*[*simp*]: *cf.V* = *V*
  ⟨*proof*⟩

sublocale *cf*!: *Finite-Graph cf*
  ⟨*proof*⟩

lemma *E-ss-cfinvE*: $E \subseteq cf.E \cup cf.E^{-1}$
  ⟨*proof*⟩

lemma *cfE-ss-invE*: $cf.E \subseteq E \cup E^{-1}$
  ⟨*proof*⟩

lemma *resE-nonNegative*: *cf e* ≥ *0*
  ⟨*proof*⟩

end

context *NFlow* **begin**
lemma *is-RGraph*: *RGraph c s t cf*
  ⟨*proof*⟩

lemma *fo-rg-inv*: *flow-of-cf cf* = *f*
  ⟨*proof*⟩

end

lemma (**in** *NFlow*)

flow-of-cf (*residualGraph c f*) = *f*
⟨*proof*⟩

9.1.1 Refinement of Operations

context *Network*
begin

We define the relation between residual graphs and flows

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

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

lemma *cfi-rel-alt*: *cfi-rel* = {(*cf*,*f*). *cf* = *residualGraph c f* ∧ *NFlow c s t f*}
⟨*proof*⟩

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

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

The bottleneck capacity is naturally defined on residual graphs

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

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

lemma (**in** *NFlow*) *augment-cf-refine-aux*:
assumes *AUG*: *isAugmenting p*
shows *residualGraph c* (*augment* (*augmentingFlow p*)) (*u*,*v*) = (
if (*u*,*v*)∈*set p then* (*residualGraph c f* (*u*,*v*) − *bottleNeck p*)
else if (*v*,*u*)∈*set p then* (*residualGraph c f* (*u*,*v*) + *bottleNeck p*)
else residualGraph c f (*u*,*v*))
⟨*proof*⟩

lemma *augment-cf-refine*:
assumes *R*: (*cf*,*f*)∈*cfi-rel*
assumes *AUG*: *NFlow.isAugmenting c s t f p*
shows (*Graph.augment-cf cf* (*set p*) (*bottleNeck-cf cf p*),
NFlow.augment c f (*NFlow.augmentingFlow c f p*)) ∈ *cfi-rel*
⟨*proof*⟩

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

definition *find-shortest-augmenting-spec-cf cf* ≡
assert (*RGraph c s t cf*) ≫
SPEC (λ

```
    None ⇒ ¬Graph.connected cf s t
  | Some p ⇒ Graph.isShortestPath cf s p t)
```

```
lemma (in RGraph) find-shortest-augmenting-spec-cf-refine:
   find-shortest-augmenting-spec-cf cf
  ≤ find-shortest-augmenting-spec (flow-of-cf cf)
  ⟨proof⟩
```

This leads to the following refined algorithm

```
definition edka2 ≡ do {
  let cf = c;

  (cf,-) ← while_T
    (λ(cf,brk). ¬brk)
    (λ(cf,-). do {
      assert (RGraph c s t cf);
      p ← find-shortest-augmenting-spec-cf cf;
      case p of
        None ⇒ return (cf,True)
      | Some p ⇒ do {
          assert (p≠[]);
          assert (Graph.isShortestPath cf s p t);
          let cf = Graph.augment-cf cf (set p) (bottleNeck-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⟩
```

9.2 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 bottleNeck-cf-impl :: 'capacity graph ⇒ path ⇒ 'capacity nres
where bottleNeck-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) bottleNeck-cf-impl-refine:
  assumes AUG: cf.isSimplePath s p t
  shows bottleNeck-cf-impl cf p ≤ SPEC (λr. r = bottleNeck-cf cf p)
⟨proof⟩

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


lemma (in Graph) augment-cf-inductive:
  fixes e cap
  defines c' ≡ augment-edge e cap
  assumes P: isSimplePath s (e#p) t
  shows augment-cf (insert e (set p)) cap = Graph.augment-cf c' (set p) cap
  and ∃s'. Graph.isSimplePath c' s' p t
⟨proof⟩

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

lemma augment-edge-impl-refine:
  assumes valid-edge e    ∀u. e≠(u,u)
  shows augment-edge-impl cf e cap
      ≤ (spec r. r = Graph.augment-edge cf e cap)
  ⟨proof⟩

definition augment-cf-impl
  :: 'capacity graph ⇒ path ⇒ 'capacity ⇒ 'capacity graph nres
  where
  augment-cf-impl cf p x ≡ do {
    (rec_T D. λ
      ([],cf) ⇒ return cf
    | (e#p,cf) ⇒ do {
        cf ← augment-edge-impl cf e x;
        D (p,cf)
      }
    ) (p,cf)
  }
```

Deriving the corresponding recursion equations

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

lemma augment-cf-impl-aux:
  assumes ∀ e∈set p. valid-edge e
  assumes ∃ s. Graph.isSimplePath cf s p t
  shows augment-cf-impl cf p x ≤ RETURN (Graph.augment-cf cf (set p) x)
  ⟨proof⟩

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

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

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

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

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

9.3 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 ← bottleNeck-cf-impl cf p;
        cf ← augment-cf-impl cf p bn;
        assert (RGraph c s t cf);
        return (cf, False)
      }
  })
  (cf,False);
 assert (RGraph c s t cf);
 let f = flow-of-cf cf;
 return f
}
```

A shortest path can be obtained by BFS

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

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

⟨*proof*⟩

9.4 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 ps cf u ≡
  filter-rev (λv. cf (u,v) > 0) (ps u)

lemma (in RGraph) rg-succ-ref1: ⟦is-pred-succ ps c⟧
  ⟹ (rg-succ ps cf u, Graph.E cf``{u}) ∈ ⟨Id⟩list-set-rel
  ⟨proof⟩

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

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

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

definition rg-succ2 ps cf u ≡ do {
  l ← ps-get-op ps 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-pred-succ ps c and V: u∈V
  shows rg-succ2 ps cf u ≤ return (rg-succ ps cf u)
⟨proof⟩
```

```
lemma (in RGraph) rg-succ-ref:
  assumes A: is-pred-succ ps c
  assumes B: u∈V
  shows rg-succ2 ps cf u ≤ SPEC (λl. (l,cf.E``{u}) ∈ ⟨Id⟩list-set-rel)
  ⟨proof⟩
```

9.5 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 ps ≡ ASSERT (is-pred-succ ps c) ≫ RETURN ps
```

```
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 ps cf ≡ Graph.bfs2 cf (rg-succ2 ps cf) s t
```

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

```
definition edka5-tabulate ps ≡ do {
  cf ← init-cf;
  ps ← init-ps ps;
  return (cf,ps)
}
```

```
definition edka5-run cf ps ≡ do {
  (cf,-) ← while_T
    (λ(cf,brk). ¬brk)
    (λ(cf,-). do {
      assert (RGraph c s t cf);
      p ← bfs2-op ps cf;
      case p of
        None ⇒ return (cf,True)
      | Some p ⇒ do {
          assert (p≠[]);
          assert (Graph.isShortestPath cf s p t);
          bn ← bottleNeck-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 ps ≡ do {
  (cf,ps) ← edka5-tabulate ps;
  edka5-run cf ps
}

lemma edka5-refine: ⟦is-pred-succ ps c⟧ ⟹ edka5 ps ≤ ⇓Id edka4
  ⟨proof⟩

end
```

9.6 Imperative Implementation

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

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

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

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

```
locale Edka-Impl = Network-Impl +
  fixes N :: nat
```

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

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*]
IdI[*of c*]

9.6.1 Implementation of Adjacency Map by Array

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

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

typedecl *i-ps*

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

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

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

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

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

lemma [*def-pat-rules*]: *Network.init-ps\$c* ≡ *UNPROTECT init-ps* ⟨*proof*⟩

sepref-register *PR-CONST init-ps* $(node \Rightarrow node\ list) \Rightarrow i\text{-}ps\ nres$

9.6.2 Implementation of Capacity Matrix by Array

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

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*

lemma [*sepref-fr-rules*]: (*uncurry* (*mtx-get N*), *uncurry* (*PR-CONST cf-get*))
$\in (is\text{-}mtx\ N)^k *_a (hn\text{-}prod\text{-}aux\ (pure\ Id)\ (pure\ Id))^k \rightarrow_a pure\ Id$
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
(*uncurry2* (*mtx-set N*), *uncurry2* (*PR-CONST cf-set*))
$\in (is\text{-}mtx\ N)^d *_a (hn\text{-}prod\text{-}aux\ (pure\ Id)\ (pure\ Id))^k *_a (pure\ Id)^k$
$\rightarrow_a (is\text{-}mtx\ N)$
⟨*proof*⟩

lemma *init-cf-imp-refine*[*sepref-fr-rules*]:
(*uncurry0* (*mtx-new N c*), *uncurry0* (*PR-CONST init-cf*))
$\in (pure\ unit\text{-}rel)^k \rightarrow_a is\text{-}mtx\ N$
⟨*proof*⟩

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

9.6.3 Representing Result Flow as Residual Graph

definition (**in** *Network-Impl*) *is-rflow N f cfi*
$\equiv \exists_A cf.\ is\text{-}mtx\ N\ cf\ cfi * \uparrow(RGraph\ c\ s\ t\ cf \wedge f = flow\text{-}of\text{-}cf\ cf)$
lemma *is-rflow-precise*[*constraint-rules*]: *precise* (*is-rflow N*)
⟨*proof*⟩

typedecl *i-rflow*

lemma [*sepref-fr-rules*]:
$(\lambda cfi.\ return\ cfi,\ PR\text{-}CONST\ compute\text{-}rflow) \in (is\text{-}mtx\ N)^d \rightarrow_a is\text{-}rflow\ N$
⟨*proof*⟩

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

9.6.4 Implementation of Functions

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

lemma *succ-imp-refine*[*sepref-fr-rules*]:
  (*uncurry2* (*succ-imp N*), *uncurry2* (*PR-CONST rg-succ2*))
    $\in$ *is-ps*^{k} $*_a$ (*is-mtx N*)^k $*_a$ (*pure Id*)^k $\rightarrow_a$ *hn-list-aux* (*pure Id*)
  ⟨*proof*⟩

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

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

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

schematic-lemma *bottleNeck-imp-impl*:
  fixes *ps* :: *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* (*is-mtx N*) *cf cfi* ∗ *hn-ctxt is-path p pi*)
    (*?c*::*?'c Heap*) *?*Γ *?R*
    (*bottleNeck-cf-impl cf p*)
  ⟨*proof*⟩
concrete-definition (**in** −) *bottleNeck-imp* **uses** *Edka-Impl.bottleNeck-imp-impl*
prepare-code-thms (**in** −) *bottleNeck-imp-def*

lemma *bottleNeck-impl-refine*[*sepref-fr-rules*]:
  (*uncurry* (*bottleNeck-imp N*), *uncurry* (*PR-CONST bottleNeck-cf-impl*))
    $\in$ (*is-mtx N*)^k $*_a$ (*is-path*)^k $\rightarrow_a$ (*pure Id*)
  ⟨*proof*⟩

lemma [*def-pat-rules*]:

Network.bottleNeck-cf-impl$c ≡ *UNPROTECT bottleNeck-cf-impl*
⟨*proof*⟩
sepref-register *PR-CONST bottleNeck-cf-impl*
capacity-impl i-mtx ⇒ *path* ⇒ *capacity-impl nres*

schematic-lemma *augment-imp-impl*:
fixes *ps* :: *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*)]
itypeI[*Pure.of cap TYPE*(*capacity-impl*)]
notes [*sepref-import-param*] = *IdI*[*of N*]
shows *hn-refine*
(*hn-ctxt* (*is-mtx N*) *cf cfi* ∗ *hn-ctxt is-path p pi* ∗ *hn-val Id cap capi*)
(*?c*::*?'c Heap*) *?*Γ *?R*
(*augment-cf-impl cf p cap*)
⟨*proof*⟩
concrete-definition (**in** −) *augment-imp* **uses** *Edka-Impl.augment-imp-impl*
prepare-code-thms (**in** −) *augment-imp-def*

lemma *augment-impl-refine*[*sepref-fr-rules*]:
(*uncurry2* (*augment-imp N*), *uncurry2* (*PR-CONST augment-cf-impl*))
$\in$ (*is-mtx N*)^d $*_a$ (*is-path*)^k $*_a$ (*pure Id*)^k $\rightarrow_a$ *is-mtx N*
⟨*proof*⟩

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

sublocale *bfs*!: *Impl-Succ*
snd
TYPE(*i-ps* × *capacity-impl i-mtx*)
λ(*ps*,*cf*). *rg-succ2 ps cf*
hn-prod-aux is-ps (*is-mtx N*)
λ(*ps*,*cf*). *succ-imp N ps cf*
⟨*proof*⟩

definition (**in** −) *bfsi' N s t psi cfi*
≡ *bfs-impl* (λ(*ps*, *cf*). *succ-imp N ps cf*) (*psi*,*cfi*) *s t*

lemma [*sepref-fr-rules*]:
(*uncurry* (*bfsi' N s t*),*uncurry* (*PR-CONST bfs2-op*))
$\in$ *is-ps*^{k} $*_a$ (*is-mtx N*)^k $\rightarrow_a$ *hn-option-aux is-path*
⟨*proof*⟩

lemma [*def-pat-rules*]: *Network.bfs2-op*$c$s$t ≡ *UNPROTECT bfs2-op* ⟨*proof*⟩
sepref-register *PR-CONST bfs2-op*

i-ps ⇒ *capacity-impl i-mtx* ⇒ *path option nres*

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

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

lemma *edka-imp-tabulate-refine*[*sepref-fr-rules*]:
  (*edka-imp-tabulate c N*, *PR-CONST edka5-tabulate*)
  $\in$ (*pure Id*)^k $\rightarrow_a$ *hn-prod-aux* (*is-mtx N*) *is-ps*
  ⟨*proof*⟩

lemma [*def-pat-rules*]:
  Network.edka5-tabulate$c ≡ *UNPROTECT edka5-tabulate*
  ⟨*proof*⟩
sepref-register *PR-CONST edka5-tabulate*
  (*node* ⇒ *node list*) ⇒ (*capacity-impl i-mtx* × *i-ps*) *nres*

schematic-lemma *edka-imp-run-impl*:
  notes [*sepref-opt-simps*] = *heap-WHILET-def*
  fixes *ps* :: *node* ⇒ *node list* **and** *cf* :: *capacity-impl graph*
  notes [*id-rules*] =
    itypeI[*Pure.of cf TYPE*(*capacity-impl i-mtx*)]
    itypeI[*Pure.of ps TYPE*(*i-ps*)]
  shows *hn-refine*
    (*hn-ctxt* (*is-mtx N*) *cf cfi* ∗ *hn-ctxt is-ps ps psi*)
    (*?c*::*?'c Heap*) *?*Γ *?R*
    (*edka5-run cf ps*)
  ⟨*proof*⟩

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

thm *edka-imp-run-def*
lemma *edka-imp-run-refine*[*sepref-fr-rules*]:
  (*uncurry* (*edka-imp-run s t N*), *uncurry* (*PR-CONST edka5-run*))
    $\in$ (*is-mtx N*)^d $*_a$ (*is-ps*)^k $\rightarrow_a$ *is-rflow N*
  ⟨*proof*⟩

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


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

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

export-code edka-imp checking SML-imp
```

9.7 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-pred-succ ps c
    shows
      <emp>
        edka-imp c s t N ps
      <λfi. ∃Af. is-rflow N f fi * ↑(isMaxFlow f)>t
  ⟨proof⟩
end
end
```

10 Combination with Network Checker

```
theory Edka-Checked-Impl
imports NetCheck EdmondsKarp-Impl
begin
```

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

10.1 Adding Statistic Counters

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

```
definition stat-outer-c :: unit Heap where stat-outer-c = return ()
lemma insert-stat-outer-c: m = stat-outer-c ≫ m
  ⟨proof⟩
definition stat-inner-c :: unit Heap where stat-inner-c = return ()
lemma insert-stat-inner-c: m = stat-inner-c ≫ m
  ⟨proof⟩

code-printing
  code-module stat ⇀ (SML) ‹
    structure stat = struct
      val outer-c = ref 0;
      fun outer-c-incr () = (outer-c := !outer-c + 1; ())
      val inner-c = ref 0;
      fun inner-c-incr () = (inner-c := !inner-c + 1; ())
    end
    ›
| constant stat-outer-c ⇀ (SML) stat.outer'-c'-incr
| constant stat-inner-c ⇀ (SML) stat.inner'-c'-incr


schematic-lemma [code]: edka-imp-run-0 s t N f brk = ?foo
  ⟨proof⟩


schematic-lemma [code]: bfs-impl-0 t u l = ?foo
  ⟨proof⟩
```

10.2 Combined Algorithm

```
definition edmonds-karp el s t ≡ do {
  case prepareNet el s t of
    None ⇒ return None
  | Some (c,ps,N) ⇒ do {
      f ← edka-imp c s t N ps ;
      return (Some (c,ps,N,f))
  }
}
export-code edmonds-karp checking SML

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

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

context
begin
private definition is-rflow ≡ Network-Impl.is-rflow theorem
  fixes el defines c ≡ ln-α el
  shows <emp> edmonds-karp el s t <λ
      None ⇒ ↑(¬ln-invar el ∨ ¬Network c s t)
    | Some (-,-,N,cf) ⇒
      ↑(ln-invar el ∧ Network c s t ∧ Graph.V c ⊆ {0..<N})
    * (∃_A f. is-rflow c s t N f cf * ↑(Network.isMaxFlow c s t f))>_t  ⟨proof⟩

end
```

10.3 Usage Example: Computing Maxflow Value

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

```
lemma (in Network) ps-s-is-incoming:
  assumes is-pred-succ ps c
  shows E``{s} = set (ps s)
  ⟨proof⟩

context RGraph begin

  lemma val-by-adj-map:
    assumes is-pred-succ ps c
    shows f.val = (∑ v∈set (ps s). c (s,v) − cf (s,v))
  ⟨proof⟩

end


context Network
begin

  definition get-cap e ≡ c e
  definition (in −) get-ps :: (node ⇒ node list) ⇒ node ⇒ node list
    where get-ps ps v ≡ ps v

  definition compute-flow-val ps cf ≡ do {
      let succs = get-ps ps s;
      setsum-impl
      (λv. 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-pred-succ ps c*
  shows *compute-flow-val ps cf* ≤ (*spec v. v* = *f.val*)
  ⟨*proof*⟩

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

```
definition compute-flow-val2 ps cf ≡ (do {
  let succs = get-ps ps 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-pred-succ ps c*
  shows *compute-flow-val2 ps cf* ≤ (*spec v. v* = *f.val*)
⟨*proof*⟩

end

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

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

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

  schematic-lemma *compute-flow-val-imp*:
    fixes *ps* :: *node* ⇒ *node list* **and** *cf* :: *capacity-impl graph*

```
    notes [id-rules] =
      itypeI[Pure.of ps TYPE(node ⇒ node list)]
      itypeI[Pure.of cf TYPE(capacity-impl i-mtx)]
    notes [sepref-import-param] = IdI[of N] IdI[of ps]
    shows hn-refine
      (hn-ctxt (is-mtx N) cf cfi)
      (?c::?'d Heap) ?Γ ?R (compute-flow-val2 ps cf)
    ⟨proof⟩

  concrete-definition (in −) compute-flow-val-imp for c s N ps 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-pred-succ ps c
  assumes RG: RGraph c s t cf
  shows
    <is-mtx N cf cfi>
      compute-flow-val-imp c s N ps cfi
    <λv. is-mtx N cf cfi ∗ ↑(v = Flow.val c s (flow-of-cf cf))>t
⟨proof⟩

lemma compute-flow-val-imp-correct:
  assumes VN: Graph.V c ⊆ {0..<N}
  assumes ABS-PS: is-pred-succ ps c
  shows
    <is-rflow N f cfi>
      compute-flow-val-imp c s N ps cfi
    <λv. is-rflow N f cfi ∗ ↑(v = Flow.val c s f)>t
  ⟨proof⟩

end


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

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

10.4 Exporting Code

export-code *nat-of-integer integer-of-nat int-of-integer integer-of-int*
  edmonds-karp edka-imp edka-imp-tabulate edka-imp-run prepareNet
  compute-flow-val-imp edmonds-karp-val
  in *SML-imp*
  module-name *Fofu*
  file *evaluation/fofu−SML/Fofu-Export.sml*

end

11 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 [23], we were able to provide a completely rigorous but still accessible proof of the Ford-Fulkerson theorem. The Isabelle Refinement Framework [16, 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.

11.1 Related Work

We are only aware of one other formalization of the Ford-Fulkerson method conducted in Mizar [19] by Lee. Unfortunately, there seems to be no publication on this formalization except [17], 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. [18], 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, 21], there are several other verifications of graph algorithms and their implementations, using different techniques and proof assistants. Noschinski [22] 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].

11.2 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*. http://afp.sf.net/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 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.

[17] G. Lee. Correctnesss of ford-fulkersons maximum flow algorithm1. *Formalized Mathematics*, 13(2):305–314, 2005.

[18] G. Lee and P. Rudnicki. Alternative aggregates in mizar. In *Calculemus '07 / MKM '07*, pages 327–341. Springer, 2007.

[19] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, page 2005, 2005.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] B. Nordhoff and P. Lammich. Formalization of Dijkstra's algorithm. *Archive of Formal Proofs*, Jan. 2012. http://afp.sf.net/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.

[22] L. Noschinski. *Formalizing Graph Theory and Planarity Certificates.* PhD thesis, Fakultt fr Informatik, Technische Universitt Mnchen, November 2015.

[23] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184. Springer, 1999.

[24] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), Apr. 1971.