Formalizing the Edmonds-Karp Algorithm

Peter Lammich and S. Reza Sefidgar

March 9, 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* ⇒ *'capacity*

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

  assumes *capacity-const*: $\forall e.\ 0 \le f\ e \wedge f\ e \le c\ e$
  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-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*: $\forall u\ v.\ c\ (u, v) \ge 0$
  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*: $\forall\, u.\ (u,\ u) \notin E$
using *no-parallel-edge* **by** *auto*

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'* $\leq$ *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'* $\leq$ *Flow.val c s f*)
unfolding *isMaxFlow-def*
by (*auto simp*: *NFlow-def*) (*intro-locales*)

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* $\leq$ *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*]:
  $(u,v) \notin E \Longrightarrow f(u,v) = 0$
  by (*metis capacity-const eq-iff zero-cap-simp*)

We provide a useful equivalent formulation of the conservation constraint.

lemma *conservation-const-pointwise*:
  assumes $u \in V - \{s,t\}$
  shows $(\sum v \in E``\{u\}.\ f\ (u,v)) = (\sum v \in E^{-1}``\{u\}.\ f\ (v,u))$
  using *conservation-const assms*
  by (*auto simp*: *sum-incoming-pointwise sum-outgoing-pointwise*)

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 $(\sum e \in outgoing\ u.\ f\ e) = (\sum v \in V.\ f\ (u,v))$
  apply (*subst sum-outgoing-alt*)
  using *assms capacity-const*
  by *auto*

lemma *sum-incoming-alt-flow*:
  fixes g :: *edge* ⇒ *'capacity*
  assumes *finite V* $u \in V$
  shows $(\sum e \in incoming\ u.\ f\ e) = (\sum v \in V.\ f\ (v,u))$
  apply (*subst sum-incoming-alt*)
  using *assms capacity-const*
  by *auto*

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

show $V \subseteq$ *reachableNodes s*
unfolding *reachableNodes-def* **using** *s-node nodes-on-st-path*
by *auto*
qed (*simp add*: *s-node reachable-ss-V*)

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*
using *reachable-is-V finite-reachable* **by** *auto*

corollary *finite-E*[*simp*, *intro*!]: *finite E*
proof –
have *finite* $(V \times V)$ **using** *finite-V* **by** *auto*
moreover have $E \subseteq V \times V$ **using** *V-def* **by** *auto*
ultimately show *?thesis* **by** (*metis finite-subset*)
qed

lemma *cap-positive*: $e \in E \Longrightarrow c\ e > 0$
unfolding *E-def* **using** *cap-non-negative le-neq-trans* **by** *fastforce*

lemma *V-not-empty*: $V \neq \{\}$ **using** *s-node* **by** *auto*
lemma *E-not-empty*: $E \neq \{\}$ **using** *V-not-empty* **by** (*auto simp*: *V-def*)

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*: $\forall e \in$ *incoming s*. $f\ e = 0$ (**is** *?thesis*)
proof (*rule ccontr*)
assume $\neg(\forall e \in$ *incoming s*. $f\ e = 0)$
then obtain *e* **where** *obt1*: $e \in$ *incoming s* $\wedge f\ e \neq 0$ **by** *blast*
then have $e \in E$ **using** *incoming-def* **by** *auto*
thus *False* **using** *obt1 no-incoming-s incoming-def* **by** *auto*
qed

lemma *no-outflow-t*: $\forall e \in$ *outgoing t*. $f\ e = 0$
proof (*rule ccontr*)
assume $\neg(\forall e \in$ *outgoing t*. $f\ e = 0)$
then obtain *e* **where** *obt1*: $e \in$ *outgoing t* $\wedge f\ e \neq 0$ **by** *blast*
then have $e \in E$ **using** *outgoing-def* **by** *auto*
thus *False* **using** *obt1 no-outgoing-t outgoing-def* **by** *auto*
qed

Thus, we can simplify the definition of the value:

corollary *val-alt*: $val = (\sum e \in outgoing\ s.\ f\ e)$
  unfolding *val-def* **by** (*auto simp*: *no-inflow-s*)

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

lemma *zero-rev-flow-simp*[*simp*]: $(u,v) \in E \Longrightarrow f(v,u) = 0$
  using *no-parallel-edge* **by** *auto*

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 $residualGraph\ c\ f \equiv \lambda(u, v).$
  if $(u, v) \in Graph.E\ c$ *then*
    $c\ (u, v) - f\ (u, v)$
  else if $(v, u) \in Graph.E\ c$ *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 $cf \equiv residualGraph\ c\ f$
  sublocale *cf*!: *Graph cf* **.**
  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 ⊆ E ∪ E⁻¹
  unfolding residualGraph-def Graph.E-def
  by auto
```

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

```
lemma resV-netV[simp]: cf.V = V
proof
  show V ⊆ Graph.V cf
  proof
    fix u
    assume u ∈ V
    then obtain v where (u, v) ∈ E ∨ (v, u) ∈ E unfolding V-def by auto

    moreover {
      assume (u, v) ∈ E
      then have (u, v) ∈ Graph.E cf ∨ (v, u) ∈ Graph.E cf
      proof (cases)
        assume f (u, v) = 0
        then have cf (u, v) = c (u, v)
          unfolding residualGraph-def using ‹(u, v) ∈ E› by (auto simp:)
        then have cf (u, v) ≠ 0 using ‹(u, v) ∈ E› unfolding E-def by auto
        thus ?thesis unfolding Graph.E-def by auto
      next
        assume f (u, v) ≠ 0
        then have cf (v, u) = f (u, v) unfolding residualGraph-def
          using ‹(u, v) ∈ E› no-parallel-edge by auto
        then have cf (v, u) ≠ 0 using ‹f (u, v) ≠ 0› by auto
        thus ?thesis unfolding Graph.E-def by auto
      qed
    } moreover {
      assume (v, u) ∈ E
      then have (v, u) ∈ Graph.E cf ∨ (u, v) ∈ Graph.E cf
      proof (cases)
        assume f (v, u) = 0
        then have cf (v, u) = c (v, u)
          unfolding residualGraph-def using ‹(v, u) ∈ E› by (auto)
        then have cf (v, u) ≠ 0 using ‹(v, u) ∈ E› unfolding E-def by auto
        thus ?thesis unfolding Graph.E-def by auto
      next
        assume f (v, u) ≠ 0
        then have cf (u, v) = f (v, u) unfolding residualGraph-def
          using ‹(v, u) ∈ E› no-parallel-edge by auto
        then have cf (u, v) ≠ 0 using ‹f (v, u) ≠ 0› by auto
        thus ?thesis unfolding Graph.E-def by auto
      qed
    } ultimately show u∈cf.V unfolding cf.V-def by auto
  qed
next
  show Graph.V cf ⊆ V using cfE-ss-invE unfolding Graph.V-def by auto
```

```
qed
```

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

```
lemma cf.V = V
  unfolding residualGraph-def Graph.E-def Graph.V-def
  using no-parallel-edge[unfolded E-def]
  by auto
```

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)
  unfolding cf.incoming-def
  apply (rule finite-subset[where B=V×V])
  using cf.E-ss-VxV by auto

lemma finite-cf-outgoing[simp, intro!]: finite (cf.outgoing v)
  unfolding cf.outgoing-def
  apply (rule finite-subset[where B=V×V])
  using cf.E-ss-VxV by auto
```

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

```
lemma resE-nonNegative: cf e ≥ 0
proof (cases e; simp)
  fix u v
  {
    assume (u, v) ∈ E
    then have cf (u, v) = c (u, v) − f (u, v) unfolding cf-def by auto
    hence cf (u,v) ≥ 0
      using capacity-const cap-non-negative by auto
  } moreover {
    assume (v, u) ∈ E
    then have cf (u,v) = f (v, u)
      using no-parallel-edge unfolding cf-def by auto
    hence cf (u,v) ≥ 0
      using capacity-const by auto
  } moreover {
    assume (u, v) ∉ E    (v, u) ∉ E
    hence cf (u,v) ≥ 0 unfolding residualGraph-def by simp
  } ultimately show cf (u,v) ≥ 0 by blast
qed
```

Again, there is an automatic proof

```
lemma cf e ≥ 0
  apply (cases e)
  unfolding residualGraph-def
  using no-parallel-edge capacity-const cap-positive
  by auto
```

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

corollary *resE-positive*: $e \in cf.E \Longrightarrow cf\ e > 0$
proof −
  assume $e \in cf.E$
  hence $cf\ e \neq 0$ **unfolding** *cf.E-def* **by** *auto*
  thus *?thesis* **using** *resE-nonNegative* **by** (*meson eq-iff not-le*)
qed

lemma *reverse-flow*: $Flow\ cf\ s\ t\ f' \Longrightarrow \forall (u, v) \in E.\ f'\ (v, u) \leq f\ (u, v)$
proof −
  assume *asm*: $Flow\ cf\ s\ t\ f'$
  {
    fix $u\ v$
    assume $(u, v) \in E$

    then have $cf\ (v, u) = f\ (u, v)$
      unfolding *residualGraph-def* **using** *no-parallel-edge* **by** *auto*
    moreover have $f'\ (v, u) \leq cf\ (v, u)$ **using** *asm*[*unfolded Flow-def*] **by** *auto*
    ultimately have $f'\ (v, u) \leq f\ (u, v)$ **by** *metis*
  }
  thus *?thesis* **by** *auto*
qed

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'* **by** (*rule f'-flow*)

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 (*cases* $(u,v)\in E$; *rule conjI*)
  assume [*simp*]: $(u,v)\in E$
  hence $f(u,v) = cf(v,u)$
    using *no-parallel-edge* **by** (*auto simp*: *residualGraph-def*)
  also have $cf(v,u) \ge f'(v,u)$ **using** *f'.capacity-const* **by** *auto*
  finally have $f'(v,u) \le f(u,v)$ **.**

  have $(f\uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u)$
    by (*auto simp*: *augment-def*)
  also have $\ldots \ge f(u,v) + f'(u,v) - f(u,v)$
    using ‹$f'(v,u) \le f(u,v)$› **by** *auto*
  also have $\ldots = f'(u,v)$ **by** *auto*
  also have $\ldots \ge 0$ **using** *f'.capacity-const* **by** *auto*
  finally show $(f\uparrow f')(u,v) \ge 0$ **.**

  have $(f\uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u)$
    by (*auto simp*: *augment-def*)
  also have $\ldots \le f(u,v) + f'(u,v)$ **using** *f'.capacity-const* **by** *auto*

also have … $\le f(u,v) + cf(u,v)$ **using** *f′.capacity-const* **by** *auto*
also have … $= f(u,v) + c(u,v) - f(u,v)$
by (*auto simp*: *residualGraph-def*)
also have … $= c(u,v)$ **by** *auto*
finally show $(f \uparrow f')(u, v) \le c(u, v)$ **.**
qed (*auto simp*: *augment-def cap-positive*)

4.2.2 Conservation Constraint

In order to show the conservation constraint, we need some auxiliary lemmas first.

As there are no parallel edges in the network, and all edges in the residual graph are either parallel or reverse to a network edge, we can split summations of the residual flow over outgoing/incoming edges in the residual graph to summations over outgoing/incoming edges in the network.

private lemma *split-rflow-outgoing*:
$(\sum v{\in}cf.E``\{u\}.\ f'\,(u,v)) = (\sum v{\in}E``\{u\}.\ f'(u,v)) + (\sum v{\in}E^{-1}``\{u\}.\ f'(u,v))$
(**is** *?LHS* = *?RHS*)
proof −
from *no-parallel-edge* **have** *DJ*: $E``\{u\} \cap E^{-1}``\{u\} = \{\}$ **by** *auto*

have *?LHS* = $(\sum v{\in}E``\{u\} \cup E^{-1}``\{u\}.\ f'\,(u,v))$
apply (*rule setsum.mono-neutral-left*)
using *cfE-ss-invE*
by (*auto intro*: *finite-Image*)
also have … = *?RHS*
apply (*subst setsum.union-disjoint*[*OF - - DJ*])
by (*auto intro*: *finite-Image*)
finally show *?LHS* = *?RHS* **.**
qed

private 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 −
from *no-parallel-edge* **have** *DJ*: $E``\{u\} \cap E^{-1}``\{u\} = \{\}$ **by** *auto*

have *?LHS* = $(\sum v{\in}E``\{u\} \cup E^{-1}``\{u\}.\ f'\,(v,u))$
apply (*rule setsum.mono-neutral-left*)
using *cfE-ss-invE*
by (*auto intro*: *finite-Image*)
also have … = *?RHS*
apply (*subst setsum.union-disjoint*[*OF - - DJ*])
by (*auto intro*: *finite-Image*)
finally show *?LHS* = *?RHS* **.**
qed

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 −

The proof is by splitting the flows, and careful cancellation of the summands.

  have $?A = (\sum v \in cf.E``\{u\}.\ f'\ (u,\ v)) - (\sum v \in E^{-1}``\{u\}.\ f'\ (u,\ v))$
    by (*simp add*: *split-rflow-outgoing*)
  also have $(\sum v \in cf.E``\{u\}.\ f'\ (u,\ v)) = (\sum v \in cf.E^{-1}``\{u\}.\ f'\ (v,\ u))$
    using *U-ASM*
    by (*simp add*: *f′.conservation-const-pointwise*)
  finally have $?A = (\sum v \in cf.E^{-1}``\{u\}.\ f'\ (v,\ u)) - (\sum v \in E^{-1}``\{u\}.\ f'\ (u,\ v))$
    by *simp*
  moreover
  have $?B = (\sum v \in cf.E^{-1}``\{u\}.\ f'\ (v,\ u)) - (\sum v \in E^{-1}``\{u\}.\ f'\ (v,\ u))$
    by (*simp add*: *split-rflow-incoming*)
  ultimately show $?A - ?B = ?RHS$ **by** *simp*
qed

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 −

We define shortcuts for the successor and predecessor nodes of u in the network:

  let $?Vo = E``\{u\}$ **let** $?Vi = E^{-1}``\{u\}$

Using the auxiliary lemma for the effective residual flow, the proof is straightforward:

  have $?LHS = (\sum v \in ?Vo.$ *augment* $f'\ (u,v))$
    by (*auto simp*: *sum-outgoing-pointwise*)

also have ...
  $= (\sum v{\in}?Vo.\ f\ (u,v) + f'(u,v) - f'(v,u))$
  by (*auto simp*: *augment-def*)
also have ...
  $= (\sum v{\in}?Vo.\ f\ (u,v)) + (\sum v{\in}?Vo.\ f'\ (u,v)) - (\sum v{\in}?Vo.\ f'\ (v,u))$
  by (*auto simp*: *setsum-subtractf setsum.distrib*)
also have ...
  $= (\sum v{\in}?Vi.\ f\ (v,u)) + (\sum v{\in}?Vi.\ f'\ (v,u)) - (\sum v{\in}?Vi.\ f'\ (u,v))$
  by (*auto simp*: *conservation-const-pointwise*[*OF U-ASM*] *flow-summation-aux*)
also have ...
  $= (\sum v{\in}?Vi.\ f\ (v,u) + f'\ (v,u) - f'\ (u,v))$
  by (*auto simp*: *setsum-subtractf setsum.distrib*)
also have ...
  $= (\sum v{\in}?Vi.\ augment\ f'\ (v,u))$
  by (*auto simp*: *augment-def*)
also have ...
  $= ?RHS$
  by (*auto simp*: *sum-incoming-pointwise*)
finally show $?LHS = ?RHS$ **.**
qed

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* $(f{\uparrow}f')$
  using *augment-flow-presv-cap augment-flow-presv-con*
  by *unfold-locales auto*

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 −
  interpret $f''!$: *Flow c s t* $f{\uparrow}f'$ **using** *augment-flow-presv*[*OF assms*] **.**

For this proof, we set up Isabelle's rewriting engine for rewriting of sums. In particular, we add lemmas to convert sums over incoming or outgoing edges to sums over all vertices. This allows us to write the summations from Cormen et al. a bit more concise, leaving some of the tedious calculation work to the computer.

Note that, if neither an edge nor its reverse is in the graph, there is also no edge in the residual graph, and thus the flow value is zero.

```
{
  fix u v
  assume (u,v)∉E    (v,u)∉E
  with cfE-ss-invE have (u,v)∉cf.E by auto
  hence f'(u,v) = 0 by auto
} note aux1 = this
```

Now, the proposition follows by straightforward rewriting of the summations:

```
have f''.val = (∑ u∈V. augment f' (s, u) − augment f' (u, s))
  unfolding f''.val-def by simp
also have ... = (∑ u∈V. f (s, u) − f (u, s) + (f' (s, u) − f' (u, s)))
  — Note that this is the crucial step of the proof, which Cormen et al. leave as
    an exercise.
  by (rule setsum.cong) (auto simp: augment-def no-parallel-edge aux1)
also have ... = val + Flow.val cf s f'
  unfolding val-def f'.val-def by simp
finally show ?thesis .
```

qed

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
  unfolding *bottleNeck-def* **apply** (*rule arg-cong*[**where** *f*=*Min*]) **by** *auto*

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 −
  assume *PATH*: *cf*.*isPath s p t*
  hence *set p*≠{} **using** *s-not-t* **by** (*auto*)
  moreover have ∀ *e*∈*set p*. *cf e* > *0*
    using *cf*.*isPath-edgeset*[*OF PATH*] *resE-positive* **by** (*auto*)
  ultimately show *?thesis* **unfolding** *bottleNeck-alt* **by** (*auto*)
qed

lemma *bottleNeck-gzero*: *isAugmenting p* ⟹ *0*<*bottleNeck p*
 using *bottleNeck-gzero-aux*[*of p*] **by** (*auto simp*: *isAugmenting-def cf*.*isSimplePath-def*)

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 −
  have (∑ *e* ∈ *A*. (*augmentingFlow p*) *e*) = *setsum* (λ-. *bottleNeck p*) (*A*∩*set p*)
    apply (*subst setsum*.*inter-restrict*)
    apply (*auto simp*: *augmentingFlow-def assms*)
    done
  thus *?thesis* **by** *auto*
qed

```
lemma augFlow-resFlow: isAugmenting p ⟹ Flow cf s t (augmentingFlow p)
proof (unfold-locales; intro allI ballI)
  assume AUG: isAugmenting p
  hence SPATH: cf.isSimplePath s p t by (simp add: isAugmenting-def)
  hence PATH: cf.isPath s p t by (simp add: cf.isSimplePath-def)

  {
```

We first show the capacity constraint

```
    fix e
    show 0 ≤ (augmentingFlow p) e ∧ (augmentingFlow p) e ≤ cf e
    proof cases
      assume e ∈ set p
      hence bottleNeck p ≤ cf e unfolding bottleNeck-alt by auto
      moreover have (augmentingFlow p) e = bottleNeck p
        unfolding augmentingFlow-def using ‹e ∈ set p› by auto
      moreover have 0 < bottleNeck p using bottleNeck-gzero[OF AUG] by simp
      ultimately show ?thesis by auto
    next
      assume e ∉ set p
      hence (augmentingFlow p) e = 0 unfolding augmentingFlow-def by auto
      thus ?thesis using resE-nonNegative by auto
    qed
  }

  {
```

Next, we show the conservation constraint

```
    fix v
    assume asm-s: v ∈ Graph.V cf − {s, t}

    have card (Graph.incoming cf v ∩ set p) = card (Graph.outgoing cf v ∩ set p)
    proof (cases)
      assume v∈set (cf.pathVertices-fwd s p)
      from cf.split-path-at-vertex[OF this PATH] obtain p1 p2 where
        P-FMT: p=p1@p2
        and 1: cf.isPath s p1 v
        and 2: cf.isPath v p2 t
        .
      from 1 obtain p1' u1 where [simp]: p1=p1'@[(u1,v)]
        using asm-s by (cases p1 rule: rev-cases) (auto simp: split-path-simps)
      from 2 obtain p2' u2 where [simp]: p2=(v,u2)#p2'
        using asm-s by (cases p2) (auto)
      from
        cf.isSPath-sg-outgoing[OF SPATH, of v u2]
        cf.isSPath-sg-incoming[OF SPATH, of u1 v]
        cf.isPath-edgeset[OF PATH]
      have cf.outgoing v ∩ set p = {(v,u2)}    cf.incoming v ∩ set p = {(u1,v)}
```

```
      by (fastforce simp: P-FMT cf.outgoing-def cf.incoming-def)+
    thus ?thesis by auto
  next
    assume v∉set (cf.pathVertices-fwd s p)
    then have ∀ u. (u,v)∉set p ∧ (v,u)∉set p
      by (auto dest: cf.pathVertices-edge[OF PATH])
    hence cf.incoming v ∩ set p = {}    cf.outgoing v ∩ set p = {}
      by (auto simp: cf.incoming-def cf.outgoing-def)
    thus ?thesis by auto
  qed
  thus (∑ e ∈ Graph.incoming cf v. (augmentingFlow p) e) =
    (∑ e ∈ Graph.outgoing cf v. (augmentingFlow p) e)
    by (auto simp: setsum-augmenting-alt)
}
qed
```

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 ⟹ Flow.val cf s (augmentingFlow p) = bottleNeck p
proof –
  assume AUG: isAugmenting p
  with augFlow-resFlow interpret f!: Flow cf s t augmentingFlow p .

  note AUG
  hence SPATH: cf.isSimplePath s p t by (simp add: isAugmenting-def)
  hence PATH: cf.isPath s p t by (simp add: cf.isSimplePath-def)
  then obtain v p' where p=(s,v)#p'    (s,v)∈cf.E
    using s-not-t by (cases p) auto
  hence cf.outgoing s ∩ set p = {(s,v)}
    using cf.isSPath-sg-outgoing[OF SPATH, of s v]
    using cf.isPath-edgeset[OF PATH]
    by (fastforce simp: cf.outgoing-def)
  moreover have cf.incoming s ∩ set p = {} using SPATH no-incoming-s
    by (auto
      simp: cf.incoming-def ‹p=(s,v)#p'› in-set-conv-decomp[where xs=p']
      simp: cf.isSimplePath-append cf.isSimplePath-cons)
  ultimately show ?thesis
    unfolding f.val-def
    by (auto simp: setsum-augmenting-alt)
qed

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 $c :: {}'capacity::linordered\text{-}idom\ graph$ **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 −
  let $?LCL = \{(u, v) \mid u\ v.\ u \in k \wedge v \in k \wedge (u, v) \in E\}$
  let $?AOG = \{(u, v) \mid u\ v.\ u \in k \wedge (u, v) \in E\}$
  let $?AIN = \{(v, u) \mid u\ v.\ u \in k \wedge (v, u) \in E\}$
  let $?SOG = \lambda u.\ (\sum e \in outgoing\ u.\ f\ e)$
  let $?SIN = \lambda u.\ (\sum e \in incoming\ u.\ f\ e)$
  let $?SOG' = (\sum e \in outgoing'\ k.\ f\ e)$
  let $?SIN' = (\sum e \in incoming'\ k.\ f\ e)$

Some setup to make finiteness reasoning implicit

  have [*simp*, *intro*!]: *finite* $?LCL$
    using *finite-subset*[*of* $?LCL$ E] *finite-E* **by** *auto*

  have [*simp*, *intro*!]: *finite* $\{(u, v).\ u \in k \wedge v \in k \wedge (u, v) \in E\}$
    using *finite-subset*[*of* $?LCL$ E] *finite-E* **by** *auto*

  have [*simp*, *intro*!]: *finite* $\{(a, y) \mid y\ a.\ (a, y) \in E\}$
    by (*rule finite-subset*[*of* - E]) *auto*

  have [*simp*, *intro*!]: *finite* $(outgoing'\ k)$
    using *finite-subset*[*of* $(outgoing'\ k)$ E] *finite-E*
    by (*auto simp*: *outgoing'-def*)

```
have [simp, intro!]: finite k
  using cut-ss-V finite-V finite-subset[of k V] by blast

have [simp, intro!]: finite (incoming′ k)
  using finite-subset[of (incoming′ k) E] finite-E
  by (auto simp: incoming′-def)


have fct1:
  netFlow = ?SOG′ + (∑ e ∈ ?LCL. f e) − (?SIN′ + (∑ e ∈ ?LCL. f e))
  (is - = ?SAOG − ?SAIN)
  using netFlow-def by auto
{
  {
    note f = setsum.union-disjoint[of ?LCL (outgoing′ k) f]
    have f3: ?LCL ∩ outgoing′ k = {} unfolding outgoing′-def by auto
    have ?SAOG = (∑ e ∈ ?LCL ∪ (outgoing′ k). f e)
      using f[OF - - f3] by auto
    moreover have ?LCL ∪ (outgoing′ k) = ?AOG
      unfolding outgoing′-def by auto
    ultimately have ?SAOG = (∑ e ∈ ?AOG. f e) by simp
  } note fct1 = this
  {
    note f = setsumExt.decomp-2[of k Pair λy a. (y, a) ∈ E f]
    have f3: ∀ x y a b. x ≠ y ⟶ (x, a) ≠ (y, b) by simp
    have (∑ e ∈ ?AOG. f e) = (∑ y ∈ k. (∑ x ∈ outgoing y. f x))
      using f[OF - - f3] outgoing-def by auto
  } note fct2 = this
  {
    note f = setsumExt.decomp-1[of k − {s} s ?SOG]
    have f2: s ∉ k − {s} by blast
    have (∑ y ∈ k − {s} ∪ {s}. ?SOG y)
        = (∑ y ∈ k − {s}. ?SOG y) + (∑ y ∈ {s}. ?SOG y)
      using f[OF - f2] by auto
    moreover have k − {s} ∪ {s} = k using s-in-cut by force
    ultimately have
      (∑ y ∈ k. ?SOG y) = (∑ y ∈ k − {s}. ?SOG y) + ?SOG s
      by auto
  } note fct3 = this
  have ?SAOG = (∑ y ∈ k − {s}. ?SOG y) + ?SOG s
    using fct1 fct2 fct3 by simp
} note fct2 = this
{
  {
    note f = setsum.union-disjoint[of ?LCL (incoming′ k) f]
    have f3: ?LCL ∩ incoming′ k = {} unfolding incoming′-def by auto
    have ?SAIN = (∑ e ∈ ?LCL ∪ (incoming′ k). f e)
      using f[OF - - f3] by auto
    moreover have ?LCL ∪ (incoming′ k) = ?AIN
```

unfolding *incoming'-def* **by** *auto*
ultimately have $?SAIN = (\sum e \in ?AIN.\ f\ e)$ **by** *simp*
} **note** $fct1 = this$
{
note $f = setsumExt.decomp\text{-}2[of\ k\ \lambda y\ a.\ Pair\ a\ y\quad \lambda y\ a.\ (a,\ y) \in E\ f]$
have $f3$: $\forall x\ y\ a\ b.\ x \neq y \longrightarrow (a,\ x) \neq (b,\ y)$ **by** *simp*
have $(\sum e \in ?AIN.\ f\ e) = (\sum y \in k.\ (\sum x \in incoming\ y.\ f\ x))$
using $f[OF\ \text{-}\ \text{-}\ f3]$ *incoming-def* **by** *auto*
} **note** $fct2 = this$
{
note $f = setsumExt.decomp\text{-}1[of\ k - \{s\}\ s\ ?SIN]$
have $f2$: $s \notin k - \{s\}$ **by** *blast*
have $(\sum y \in k - \{s\} \cup \{s\}.\ ?SIN\ y)$
$= (\sum y \in k - \{s\}.\ ?SIN\ y) + (\sum y \in \{s\}.\ ?SIN\ y)$
using $f[OF\ \text{-}\ f2]$ **by** *auto*
moreover have $k - \{s\} \cup \{s\} = k$ **using** *s-in-cut* **by** *force*
ultimately have $(\sum y \in k.\ ?SIN\ y) = (\sum y \in k - \{s\}.\ ?SIN\ y) + ?SIN\ s$
by *auto*
} **note** $fct3 = this$
have $?SAIN = (\sum y \in k - \{s\}.\ ?SIN\ y) + ?SIN\ s$
using *fct1 fct2 fct3* **by** *simp*
} **note** $fct3 = this$
have $netFlow =$
$((\sum y \in k - \{s\}.\ ?SOG\ y) + ?SOG\ s)$
$- ((\sum y \in k - \{s\}.\ ?SIN\ y) + ?SIN\ s)$
(**is** $netFlow = ?R$)
using *fct1 fct2 fct3* **by** *auto*
moreover have $?R = ?SOG\ s - ?SIN\ s$
proof −
note $f = setsum.cong[of\ k - \{s\}\quad k - \{s\}\ ?SOG\ ?SIN]$
have $f1$: $k - \{s\} = k - \{s\}$ **by** *blast*
have $f2$: $(\bigwedge u.\ u \in k - \{s\} \Longrightarrow ?SOG\ u = ?SIN\ u)$
using *conservation-const cut-ss-V t-ni-cut* **by** *force*
have $(\sum y \in k - \{s\}.\ ?SOG\ y) = (\sum y \in k - \{s\}.\ ?SIN\ y)$
using $f[OF\ f1\ f2]$ **by** *blast*
thus *?thesis* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *val-def* **by** *simp*
qed

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 −
have $(\sum e \in outgoing'\ k.\ f\ e) \leq (\sum e \in outgoing'\ k.\ c\ e)$ (**is** $?L \leq ?R$)
using *capacity-const* **by** (*metis setsum-mono*)
then have $(\sum e \in outgoing'\ k.\ f\ e) \leq cap$ **unfolding** *cap-def* **by** *simp*
moreover have $val \leq (\sum e \in outgoing'\ k.\ f\ e)$ **using** *netFlow-def*

by (*simp add*: *capacity-const flow-value setsum-nonneg*)
ultimately show *?thesis* **by** *simp*
qed

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*: $isMaxFlow\ f \Longrightarrow \neg\ (\exists\ p.\ isAugmenting\ p)$
unfolding *isMaxFlow-alt*
proof (*rule ccontr*)
assume *asm*: $NFlow\ c\ s\ t\ f$
$\wedge\ (\forall f'.\ NFlow\ c\ s\ t\ f' \longrightarrow Flow.val\ c\ s\ f' \leq Flow.val\ c\ s\ f)$
assume *asm-c*: $\neg\ \neg\ (\exists\ p.\ isAugmenting\ p)$
then obtain *p* **where** *obt*: *isAugmenting p* **by** *blast*
have *fct1*: *Flow cf s t* (*augmentingFlow p*) **using** *obt augFlow-resFlow* **by** *auto*
have *fct2*: *Flow.val cf s* (*augmentingFlow p*) $>$ *0* **using** *obt augFlow-val*
bottleNeck-gzero isAugmenting-def cf.isSimplePath-def **by** *auto*
have *NFlow c s t* (*augment* (*augmentingFlow p*))
using *fct1 augment-flow-presv Network-axioms* **unfolding** *NFlow-def* **by** *auto*
moreover have *Flow.val c s* (*augment* (*augmentingFlow p*)) $>$ *val*
using *fct1 fct2 augment-flow-value* **by** *auto*
ultimately show *False* **using** *asm* **by** *auto*
qed

lemma *fofu-II-III*:
$\neg\ (\exists\ p.\ isAugmenting\ p) \Longrightarrow \exists k'.\ NCut\ c\ s\ t\ k' \wedge val = NCut.cap\ c\ k'$
proof (*intro exI conjI*)
let *?S* = *cf.reachableNodes s*
assume *asm*: $\neg\ (\exists\ p.\ isAugmenting\ p)$
hence $t \notin ?S$
unfolding *isAugmenting-def cf.reachableNodes-def cf.connected-def*
by (*auto dest*: *cf.isSPath-pathLE*)
then show *CUT*: *NCut c s t ?S*
proof *unfold-locales*
show *Graph.reachableNodes cf s* $\subseteq$ *V*
using *cf.reachable-ss-V s-node resV-netV* **by** *auto*
show *s* $\in$ *Graph.reachableNodes cf s*
unfolding *Graph.reachableNodes-def Graph.connected-def*
by (*metis Graph.isPath.simps*(*1*) *mem-Collect-eq*)
qed
then interpret *NCut c s t ?S* **.**
interpret *NFlowCut c s t f ?S* **by** *intro-locales*

have $\forall (u,v) \in outgoing'\ ?S.\ f\ (u,v) = c\ (u,v)$

```
  proof (rule ballI, rule ccontr, clarify) — Proof by contradiction
    fix u v
    assume (u,v)∈outgoing' ?S
    hence (u,v)∈E   u∈?S   v∉?S
      by (auto simp: outgoing'-def)
    assume f (u,v) ≠ c (u,v)
    hence f (u,v) < c (u,v)
      using capacity-const by (metis (no-types) eq-iff not-le)
    hence cf (u, v) ≠ 0
      unfolding residualGraph-def using ‹(u,v)∈E› by auto
    hence (u, v) ∈ cf.E unfolding cf.E-def by simp
    hence v∈?S using ‹u∈?S› by (auto intro: cf.reachableNodes-append-edge)
    thus False using ‹v∉?S› by auto
  qed
  hence (∑ e ∈ outgoing' ?S. f e) = cap
    unfolding cap-def by auto
  moreover
  have ∀(u,v)∈incoming' ?S. f (u,v) = 0
  proof (rule ballI, rule ccontr, clarify) — Proof by contradiction
    fix u v
    assume (u,v)∈incoming' ?S
    hence (u,v)∈E   u∉?S   v∈?S by (auto simp: incoming'-def)
    hence (v,u)∉E using no-parallel-edge by auto

    assume f (u,v) ≠ 0
    hence cf (v, u) ≠ 0
      unfolding residualGraph-def using ‹(u,v)∈E› ‹(v,u)∉E› by auto
    hence (v, u) ∈ cf.E unfolding cf.E-def by simp
    hence u∈?S using ‹v∈?S› cf.reachableNodes-append-edge by auto
    thus False using ‹u∉?S› by auto
  qed
  hence (∑ e ∈ incoming' ?S. f e) = 0
    unfolding cap-def by auto
  ultimately show val = cap
    unfolding flow-value[symmetric] netFlow-def by simp
qed

lemma fofu-III-I:
  ∃k. NCut c s t k ∧ val = NCut.cap c k ⟹ isMaxFlow f
proof clarify
  fix k
  assume NCut c s t k
  then interpret NCut c s t k .
  interpret NFlowCut c s t f k by intro-locales

  assume val = cap
  {
    fix f'
    assume Flow c s t f'
```

then interpret *fc'!*: *NFlow c s t f'* **by** *intro-locales*
interpret *fc'!*: *NFlowCut c s t f' k* **by** *intro-locales*

have $fc'.val \le cap$ **using** *fc'.weak-duality* **.**
also note ‹$val = cap$›[*symmetric*]
finally have $fc'.val \le val$ **.**
}
thus *isMaxFlow f* **unfolding** *isMaxFlow-def*
by *simp unfold-locales*
qed

Finally we can state the Ford-Fulkerson theorem:

theorem *ford-fulkerson*: **shows**
isMaxFlow f $\longleftrightarrow$
$\neg$ *Ex isAugmenting* **and** $\neg$ *Ex isAugmenting* $\longleftrightarrow$
$(\exists\, k.\ NCut\ c\ s\ t\ k \wedge val = NCut.cap\ c\ k)$
using *fofu-I-II fofu-II-III fofu-III-I* **by** *auto*

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*: $(\sum e \in incoming\ t.\ f\ e) = (\sum e \in outgoing\ s.\ f\ e)$
proof $-$

We choose a cut between the sink and all other nodes

let $?K = V - \{t\}$
interpret *NFlowCut c s t f ?K*
using *s-node s-not-t* **by** *unfold-locales auto*

The cut is chosen such that its outgoing edges are the incoming edges to the sink, and its incoming edges are the outgoing edges from the sink. Note that the sink has no outgoing edges.

have $outgoing'\ ?K = incoming\ t$
and $incoming'\ ?K = \{\}$
using *no-self-loop no-outgoing-t*
unfolding *outgoing'-def incoming-def incoming'-def outgoing-def V-def*
by *auto*
hence $(\sum e \in incoming\ t.\ f\ e) = netFlow$ **unfolding** *netFlow-def* **by** *auto*
also have $netFlow = val$ **by** (*rule flow-value*)
also have $val = (\sum e \in outgoing\ s.\ f\ e)$ **by** (*auto simp*: *val-alt*)
finally show *?thesis* **.**
qed

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*
 using *ford-fulkerson* **by** *blast*

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*$]\!]$
 $\Longrightarrow$ *Flow.val c s f* = *NCut.cap c k*
proof −
 assume *isMaxFlow f* *isMinCut c s t k*
 then interpret *Flow c s t f* + *NCut c s t k*
  unfolding *isMaxFlow-def isMinCut-def* **by** *simp-all*
 interpret *NFlowCut c s t f k* **by** *intro-locales*

 from *ford-fulkerson* ‹*isMaxFlow f*›
 obtain *k′* **where** *K′*: *NCut c s t k′* *val* = *NCut.cap c k′*
  by *blast*
 show *val* = *cap*
  using ‹*isMinCut c s t k*› *K′ weak-duality*
  unfolding *isMinCut-def* **by** *auto*
qed

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* $\equiv$ *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)
  unfolding NFlow-def Flow-def
  using Network-axioms
  by (auto simp: s-node t-node cap-non-negative)
```

Augmentation preserves the flow property

```
lemma (in NFlow) augment-pres-nflow:
```

assumes *AUG*: *isAugmenting p*
shows *NFlow c s t* (*augment* (*augmentingFlow p*))
proof −
note *augment-flow-presv*[*OF augFlow-resFlow*[*OF AUG*]]
thus *?thesis*
by *intro-locales*
qed

Augmenting paths cannot be empty

lemma (**in** *NFlow*) *augmenting-path-not-empty*:
$\neg$*isAugmenting* []
unfolding *isAugmenting-def* **using** *s-not-t* **by** *auto*

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

theorem *fofu-partial-correct*: *fofu* $\leq$ (*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*])
done

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* $\equiv$ *augment* (*augmentingFlow p*)

context begin

private abbreviation (*input*) *augment*
$\equiv$ *NFlow.augment-with-path*
private abbreviation (*input*) *is-augmenting-path f p*
$\equiv$ *NFlow.isAugmenting c s t f p*

definition *ford-fulkerson-method* $\equiv$ *do* {
let f = (λ(*u*,*v*). *0*);

(*f*,*brk*) $\leftarrow$ *while* (λ(*f*,*brk*). $\neg$*brk*)
(λ(*f*,*brk*). *do* {
p $\leftarrow$ *selectp p. is-augmenting-path f p*;
case p of
None $\Rightarrow$ *return* (*f*,*True*)
| *Some p* $\Rightarrow$ *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 −
  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*)
    done
  also note *fofu-partial-correct*
  finally show *?thesis* **.**
qed

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 ⟹ ∃ *p*. *Graph.isShortestPath cf s p t*
  using *Graph.obtain-shortest-path* **unfolding** *isAugmenting-def*

by (*fastforce simp*: *Graph.isSimplePath-def Graph.connected-def*)

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

We show that our refined procedure is actually a refinement

lemma *find-shortest-augmenting-refine*[*refine*]:
(f',f)∈*Id* $\Longrightarrow$ *find-shortest-augmenting-spec* $f' \leq \Downarrow$*Id* (*find-augmenting-spec f*)
unfolding *find-shortest-augmenting-spec-def find-augmenting-spec-def*
apply (*refine-vcg*)
apply (*auto*
simp: *NFlow.shortest-is-augmenting*
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 (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*
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

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)∈set p
  assumes isPath s p' t    (v,u)∈set p'
  shows length p' ≥ 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'* ≠ {}› **apply** *auto* []
apply (*rprems*, *assumption*)
apply *auto*
done
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' ≠ {}› apply auto []
  apply (rprems, assumption)
  apply auto
  done
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* $\subseteq$ *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*[*OF assms*]
  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* $\subseteq$ *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 finite-V 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 mlex-snd-decrI)
      apply (simp add: EQ)
      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 mlex-leI)
```

```
      apply (simp add: EQ)
      apply (rule card-mono)
      apply simp
      by fact
    note G1 G2
  } ultimately show
    g'.ekMeasure ≤ ekMeasure
    edges − Graph.E c' ≠ {} ⟹ 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 ≡ λe.
  if e∈edges then c e − cap
  else if prod.swap e∈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 ⊆ E⟧ ⟹ Graph.V (augment-cf edges cap) ⊆ V

  unfolding Graph.E-def Graph.V-def
  by (auto simp add: augment-cf-def) []

lemma augment-saturate:
  fixes edges e
  defines c' ≡ augment-cf edges (c e)
  assumes EIE: e∈edges
  shows e∉Graph.E c'
  using EIE unfolding c'-def augment-cf-def
  by (auto simp: Graph.E-def)


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
  using assms
```

by (*fastforce simp*: *Graph.augment-cf-def intro*!: *ext*)

end — Graph

context *NFlow* **begin**

lemma *augmenting-edge-no-swap*: *isAugmenting p* $\Longrightarrow$ *set p* $\cap$ (*set p*)$^{-1}$ = {}
  using *cf.isSPath-nt-parallel-pf*
  by (*auto simp*: *isAugmenting-def*)

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

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 −
  {
    fix *u v*
    assume (*u*,*v*)$\in$*set p*
    hence *bottleNeck p* $\leq$ *cf* (*u*,*v*)
      unfolding *bottleNeck-def* **by** (*auto intro*: *Min-le*)
  } **note** *bn-smallerI* = *this*

  {
    fix *u v*
    assume (*u*,*v*)$\in$*set p*
    hence (*u*,*v*)$\in$*cf.E* **using** *AUG cf.isPath-edgeset*
      by (*auto simp*: *isAugmenting-def cf.isSimplePath-def*)
    hence (*u*,*v*)$\in$*E* $\vee$ (*v*,*u*)$\in$*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-bottleneck:
  assumes isAugmenting p
  obtains e where e∈set p    cf e = bottleNeck p
proof −
  from assms have p≠[] by (auto simp: isAugmenting-def s-not-t)
  hence {cf e | e. e ∈ set p} ≠ {} by (cases p) auto
  with Min-in[OF aug-flows-finite this, folded bottleNeck-def]
    obtain e where e∈set p    cf e = bottleNeck 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
    apply unfold-locales
    by (auto simp: resV-netV finite-V)

  interpret cf'!: ek-analysis-defs cf' .

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

  note BNGZ = bottleNeck-gzero[OF AUG]

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

  obtain e where
    EIP: e∈set p and EBN: cf e = bottleNeck p
    by (rule augmenting-path-contains-bottleneck[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 + bottleNeck 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
```

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
 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)
```

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: assert-bind-spec-conv
  simp: find-shortest-augmenting-spec-def
  simp: edka-wf-rel-def NFlow.shortest-path-decr-ek-measure
 ; fail)
```

The other VCs are straightforward

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

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 −
  interpret NFlow c s t (λ-. 0)
    unfolding NFlow-def Flow-def using Network-axioms
      by (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.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 −
  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)
    using ekMeasure-upper-bound apply auto []
    apply auto []
    apply (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* $\le$ (*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

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.

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*)

  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*
      by *unfold-locales*

    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 −
  interpret *f'*!: *NFlow c s t f'* **by** *fact*

  show *?thesis*
    unfolding *f-def*[*abs-def*] *flow-of-cf-def*[*abs-def*]
    unfolding *A residualGraph-def*
    apply (*rule ext*)
    using *f'.capacity-const* **unfolding** *E-def*
    apply (*auto split*: *prod.split*)
    by (*metis antisym*)
qed

lemma *is-NFlow*: *NFlow c s t* (*flow-of-cf cf*)
  apply (*fold f-def*)
  using *EX-RG f-unique* **by** *metis*

sublocale *f*!: *NFlow c s t f* **unfolding** *f-def* **by** (*rule is-NFlow*)

lemma *rg-is-cf*[*simp*]: *residualGraph c f* = *cf*
  using *EX-RG f-unique* **by** *auto*

lemma *rg-fo-inv*[*simp*]: *residualGraph c* (*flow-of-cf cf*) = *cf*
  using *rg-is-cf*
  unfolding *f-def*
  .

sublocale *cf*!: *Graph cf* **.**

lemma *resV-netV*[*simp*]: *cf.V* = *V*
  using *f.resV-netV* **by** *simp*

sublocale *cf*!: *Finite-Graph cf*
  apply *unfold-locales*
  apply *simp*
  done

lemma *finite-cf*: *finite* (*cf.V*) **by** *simp*

end

context *NFlow* **begin**
  lemma *is-RGraph*: *RGraph c s t cf*
    apply *unfold-locales*
    apply (*rule exI*[**where** *x*=*f*])
    apply (*safe*; *unfold-locales*)
    done

```
lemma fo-rg-inv: flow-of-cf cf = f
  unfolding flow-of-cf-def[abs-def]
  unfolding residualGraph-def
  apply (rule ext)
  using capacity-const unfolding E-def
  apply (clarsimp split: prod.split)
  by (metis antisym)

end
```

```
lemma (in NFlow)
 flow-of-cf (residualGraph c f) = f
 by (rule fo-rg-inv)
```

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}
  unfolding cfi-rel-def br-def
 by (auto simp: NFlow.is-RGraph RGraph.is-NFlow RGraph.rg-fo-inv NFlow.fo-rg-inv)
```

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 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
  unfolding bottleNeck-cf-def bottleNeck-def ..
```

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

```
lemma (in NFlow)
  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))
  using augment-alt[OF AUG] by (auto simp: Graph.augment-cf-def)
```

```
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 -
  from R have FEQ: f = flow-of-cf cf    RGraph c s t cf
    by (auto simp: cfi-rel-def br-def)
  then interpret cf!: RGraph c s t cf by simp

  from FEQ have [simp]: f = cf.f by (simp add: cf.f-def)
  note AUG'=AUG[simplified]

  show (Graph.augment-cf cf (set p) (bottleNeck-cf cf p),
    NFlow.augment c f (NFlow.augmentingFlow c f p)) ∈ cfi-rel

    apply (subst cf.f.bottleNeck-cf-refine[simplified])
    apply (clarsimp simp: cfi-rel-def br-def; safe)
    apply (subst cf.f.augment-alt[OF AUG', simplified, symmetric])
    apply (subst NFlow.fo-rg-inv)
    apply (rule cf.f.augment-pres-nflow)
    apply fact
    apply (rule refl)
    apply (subst cf.f.augment-alt[OF AUG', simplified, symmetric])
    apply (rule NFlow.is-RGraph)
    apply (rule cf.f.augment-pres-nflow)
    apply fact
    done
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 (λNone ⇒ ¬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 find-shortest-augmenting-spec-def
  by (auto
    simp: pw-le-iff refine-pw-simps
    simp: this-loc rg-is-cf
    simp: f.isAugmenting-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,-) ← WHILET
    (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 -
 have [refine-dref-RELATES]: RELATES cfi-rel by (simp add: RELATES-def)

  show ?thesis
    unfolding edka2-def edka-def
    apply (rewrite in let f' = NFlow.augmentingFlow c - - in - Let-def)
    apply (rewrite in let f = flow-of-cf - in - Let-def)
    apply (refine-rcg)
    apply refine-dref-type
    apply vc-solve

  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: RGraph.rg-fo-inv; 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
```

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 −


    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 bottleNeck-cf-impl-def bottleNeck-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 []
      apply auto []
      apply auto []
      apply auto []
      apply auto []
      apply auto []
      apply auto []
      apply simp
      apply (fo-rule arg-cong; auto)
      apply auto []
      apply auto []
      apply simp
      apply (fo-rule arg-cong; auto)
      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 $\exists s'$. *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*:
⟦*valid-edge e*; ∀ *u*. *e*≠(*u*,*u*)⟧ ⟹ *augment-edge-impl cf e cap* ≤ *SPEC* (λ*r*. *r* = *Graph.augment-edge cf e cap*)
unfolding *augment-edge-impl-def Graph.augment-edge-def 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* {
RECT (λ*D*. λ
([],*cf*) ⇒ *RETURN cf*
| (*e*#*p*,*cf*) ⇒ *do* {
cf ← *augment-edge-impl cf e x*;
D (*p*,*cf*)
}
) (*p*,*cf*)
}

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

definition edka3 ≡ do {
  let cf = c;

  (cf,-) ← WHILET
    (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* ≤ ⇓*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.bottleNeck-cf-impl-refine*)
apply (*frule* (*1*) *RGraph.augment-cf-impl-refine*)
apply (*auto simp*: *pw-le-iff refine-pw-simps*)
done

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,-) ← WHILET
    (λ(cf,brk). ¬brk)
    (λ(cf,-). do {
      ASSERT (RGraph c s t cf);
      p ← Graph.bfs cf s t;
      case p of
        None ⇒ RETURN (cf,True)
      | Some p ⇒ do {
          ASSERT (p≠[]);
          ASSERT (Graph.isShortestPath cf s p t);
          bn ← 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*
unfolding *find-shortest-augmenting-spec-cf-def*
apply (*rule le-ASSERTI*)
apply (*rule order-trans*)
apply (*rule Graph.bfs-correct*)
apply (*simp add*: *RGraph.resV-netV s-node*)
apply (*simp add*: *RGraph.resV-netV*)
apply (*simp*)
done

lemma *edka4-refine*: *edka4* $\leq \Downarrow$*Id edka3*
unfolding *edka4-def edka3-def*
apply *refine-rcg*
apply *refine-dref-type*
apply (*vc-solve simp*: *bfs-refines-shortest-augmenting-spec*)
done

9.4 Implementing the Successor Function for BFS

— 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* $\equiv$
filter-rev ($\lambda v.$ *cf* $(u,v) > 0$) (*ps u*)

lemma (**in** *NFlow*) *E-ss-cfinvE*: $E \subseteq$ *Graph.E cf* $\cup$ (*Graph.E cf*)$^{-1}$
unfolding *residualGraph-def Graph.E-def*
apply (*clarsimp*)
using *no-parallel-edge*
unfolding *E-def*
apply (*simp add*:)
done

lemma (**in** *RGraph*) *E-ss-cfinvE*: $E \subseteq cf.E \cup cf.E^{-1}$
using *f.E-ss-cfinvE* **by** *simp*

lemma (**in** *RGraph*) *cfE-ss-invE*: $cf.E \subseteq E \cup E^{-1}$
using *f.cfE-ss-invE* **by** *simp*

lemma (**in** *RGraph*) *resE-nonNegative*: *cf e* ≥ 0
using *f.resE-nonNegative* **by** *auto*

lemma (**in** *RGraph*) *rg-succ-ref1*: $[\![$*is-pred-succ ps c*$]\!]$
$\Longrightarrow$ (*rg-succ ps cf u*, *Graph.E cf*``$\{u\}$) $\in \langle$*Id*$\rangle$*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-pred-succ-def less-le Graph.E-def simp del*: *cf.zero-cap-simp*

```
zero-cap-simp) []
   apply (auto simp: is-pred-succ-def) []
   done

 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 ≡ RECT (λD (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 (λr. r=Q x)
   shows monadic-filter-rev P l ≤ SPEC (λr. 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 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 −
  have ∀ v∈set (ps u). valid-edge (u,v)
    using PS V
    by (auto simp: is-pred-succ-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-pred-succ ps c
  assumes B: u∈V
  shows rg-succ2 ps 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)


definition init-cf :: 'capacity graph nres where init-cf ≡ RETURN c
definition init-ps :: (node ⇒ node list) ⇒ - where
  init-ps ps ≡ ASSERT (is-pred-succ ps c) ≫ RETURN ps

definition compute-rflow :: 'capacity graph ⇒ 'capacity flow nres 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

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

definition edka5-run cf ps ≡ do {
  (cf,-) ← WHILET
    (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*
  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*: *RGraph.resV-netV*)
  apply (*simp add*: *RGraph.rg-succ-ref*)
  done

end

9.5 Imperative Implementation

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

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*

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

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*)
apply *rule*
unfolding *is-ps-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

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* **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*)) $\in$ *is-ps*$^k *_a$ (*pure Id*)^k $\rightarrow_a$ *hn-list-aux* (*pure Id*)
unfolding *hn-list-pure-conv*
apply *rule* **apply** *rule*
using *V-ss*
by (*sep-auto simp*: *is-ps-def pure-def ps-get-imp-def ps-get-op-def refine-pw-simps*)

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*

lemma [*sepref-fr-rules*]: (*uncurry* (*mtx-get* N), *uncurry* (*PR-CONST cf-get*)) $\in$ (*is-mtx* N)^k $*_a$ (*hn-prod-aux* (*pure Id*) (*pure Id*))^k $\rightarrow_a$ *pure Id*
apply *rule* **apply** *rule*
using *V-ss*
by (*sep-auto simp*: *cf-get-def refine-pw-simps pure-def*)

lemma [*sepref-fr-rules*]: (*uncurry2* (*mtx-set* N), *uncurry2* (*PR-CONST cf-set*))

$\in$ (*is-mtx* N)^d $*_a$ (*hn-prod-aux* (*pure Id*) (*pure Id*))^k $*_a$ (*pure Id*)^k $\rightarrow_a$ (*is-mtx* N)
apply *rule* **apply** *rule*
using *V-ss*
by (*sep-auto simp*: *cf-set-def refine-pw-simps pure-def hn-ctxt-def*)

lemma *is-pred-succ-no-node*: $[\![$*is-pred-succ a c*; $u \notin V]\!] \Longrightarrow a\ u = []$
unfolding *is-pred-succ-def V-def*
by *auto*

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

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

lemma *init-cf-imp-refine*[*sepref-fr-rules*]:
(*uncurry0* (*mtx-new* N c), *uncurry0* (*PR-CONST init-cf*)) $\in$ (*pure unit-rel*)^k $\rightarrow_a$ *is-mtx* N
apply *rule* **apply** *rule*
using *V-ss*
by (*sep-auto simp*: *init-cf-def*)

lemma [*def-pat-rules*]: *Network.init-cf*\$*c* $\equiv$ *UNPROTECT init-cf* **by** *simp*
sepref-register *PR-CONST init-cf* *capacity-impl i-mtx nres*

definition (**in** *Network-Impl*) *is-rflow N f cfi* $\equiv \exists_A cf.$ *is-mtx N cf cfi* $* \uparrow(f =$ *flow-of-cf cf*)
lemma *is-rflow-precise*[*constraint-rules*]: *precise* (*is-rflow* N)
apply *rule*
unfolding *is-rflow-def*
apply *clarsimp*
apply (*rename-tac l l'*)
apply *prec-extract-eqs*
apply *simp*
done

typedecl *i-rflow*

lemma [*sepref-fr-rules*]: (λ*cfi. return cfi*, *PR-CONST compute-rflow*) $\in$ (*is-mtx N*)^d $\rightarrow_a$ *is-rflow N*
apply *rule*
apply *rule*
apply (*sep-auto simp*: *compute-rflow-def is-rflow-def refine-pw-simps hn-ctxt-def*)
done

lemma [*def-pat-rules*]: *Network.compute-rflow*\$*c*\$*s*\$*t* $\equiv$ *UNPROTECT compute-rflow* **by** *simp*
sepref-register *PR-CONST compute-rflow* *capacity-impl i-mtx* $\Rightarrow$ *i-rflow nres*

schematic-lemma *rg-succ2-impl*:
fixes *ps* :: *node* $\Rightarrow$ *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*)
unfolding *rg-succ2-def APP-def monadic-filter-rev-def monadic-filter-rev-aux-def*

using [[*id-debug*, *goals-limit* = *1*]]
by *sepref-keep*
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*)
apply *rule*
using *succ-imp.refine*[*OF this-loc*]
by (*auto simp*: *hn-ctxt-def hn-prod-aux-def mult-ac split*: *prod.split*)

lemma [*def-pat-rules*]: *Network.rg-succ2*\$*c* $\equiv$ *UNPROTECT rg-succ2* **by** *simp*
sepref-register *PR-CONST rg-succ2* *i-ps* $\Rightarrow$ *capacity-impl i-mtx* $\Rightarrow$ *node* $\Rightarrow$ *node list nres*

lemma [*sepref-import-param*]: (*min*,*min*)$\in$*Id*$\rightarrow$*Id*$\rightarrow$*Id* **by** *simp*

abbreviation *is-path* $\equiv$ *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*)
unfolding *bottleNeck-cf-impl-def APP-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref-keep*
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*)
apply *rule*
apply (*rule hn-refine-preI*)
apply (*clarsimp simp*: *uncurry-def hn-list-pure-conv hn-ctxt-def split*: *prod.split*)
apply (*clarsimp simp*: *pure-def*)
apply (*rule hn-refine-cons'*[*OF* - *bottleNeck-imp.refine*[*OF this-loc*] -])
apply (*simp add*: *hn-list-pure-conv hn-ctxt-def*)
apply (*simp add*: *pure-def*)
apply (*simp add*: *hn-ctxt-def*)
apply (*simp add*: *pure-def*)
done

lemma [*def-pat-rules*]: *Network.bottleNeck-cf-impl$c* ≡ *UNPROTECT bottleNeck-cf-impl*
by *simp*
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*)
unfolding *augment-cf-impl-def augment-edge-impl-def APP-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref-keep*
concrete-definition (**in** −) *augment-imp* **uses** *Edka-Impl.augment-imp-impl*
prepare-code-thms (**in** −) *augment-imp-def*

thm *augment-imp-def augment-cf-impl-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*
apply *rule*
apply (*rule hn-refine-preI*)
apply (*clarsimp simp*: *uncurry-def hn-list-pure-conv hn-ctxt-def split*: *prod.split*)
apply (*clarsimp simp*: *pure-def*)
apply (*rule hn-refine-cons'*[*OF - augment-imp.refine*[*OF this-loc*] -])
apply (*simp add*: *hn-list-pure-conv hn-ctxt-def*)
apply (*simp add*: *pure-def*)
apply (*simp add*: *hn-ctxt-def*)
apply (*simp add*: *pure-def*)
done

lemma [*def-pat-rules*]: *Network.augment-cf-impl*\$*c* $\equiv$ *UNPROTECT augment-cf-impl* **by** *simp*
sepref-register *PR-CONST augment-cf-impl* *capacity-impl i-mtx* $\Rightarrow$ *path* $\Rightarrow$ *capacity-impl* $\Rightarrow$ *capacity-impl i-mtx nres*

thm *succ-imp-def*
sublocale *bfs*!: *Impl-Succ snd* *TYPE*(*i-ps* $\times$ *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*
unfolding *APP-def*
apply *unfold-locales*
apply *constraint-rules*
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* $\equiv$ *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*
unfolding *bfsi'-def*[*abs-def*]
using *bfs.bfs-impl-fr-rule*
apply (*simp add*: *uncurry-def bfs.op-bfs-def*[*abs-def*] *bfs2-op-def*)
apply (*clarsimp simp*: *hfref-def all-to-meta*)
apply (*rule hn-refine-cons*[*rotated*])
apply *rprems*
apply (*sep-auto simp*: *pure-def*)
apply (*sep-auto simp*: *pure-def*)
apply (*sep-auto simp*: *pure-def*)
done

lemma [*def-pat-rules*]: *Network.bfs2-op*\$*c*\$*s*\$*t* $\equiv$ *UNPROTECT bfs2-op* **by** *simp*
sepref-register *PR-CONST bfs2-op* *i-ps* $\Rightarrow$ *capacity-impl i-mtx* $\Rightarrow$ *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*)
unfolding *edka5-tabulate-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref-keep*

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

thm *edka-imp-tabulate.refine*

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*
apply (*rule*)
apply (*rule hn-refine-preI*)
apply (*clarsimp simp*: *uncurry-def hn-list-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-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*)
unfolding *edka5-run-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref-keep*

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*]:
($uncurry$ ($edka\text{-}imp\text{-}run\ s\ t\ N$), $uncurry$ ($PR\text{-}CONST\ edka5\text{-}run$))
$\in$ ($is\text{-}mtx\ N$)^d $*_a$ ($is\text{-}ps$)^k $\to_a$ $is\text{-}rflow\ N$
apply *rule*
apply (*clarsimp simp*: *uncurry-def hn-list-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* $\equiv$ *UNPROTECT edka5-run*
by *simp*
sepref-register *PR-CONST edka5-run* *capacity-impl i-mtx* $\Rightarrow$ *i-ps* $\Rightarrow$ *i-rflow nres*

schematic-lemma *edka-imp-impl*:
notes [*sepref-opt-simps*] = *heap-WHILET-def*
fixes *ps* :: *node* $\Rightarrow$ *node list* **and** *cf* :: *capacity-impl graph*
notes [*id-rules*] =
itypeI[*Pure.of ps TYPE*(*node* $\Rightarrow$ *node list*)]
notes [*sepref-import-param*] = *IdI*[*of ps*]
shows *hn-refine* (*emp*) (*?c*::*?'c Heap*) *?*Γ *?R* (*edka5 ps*)
unfolding *edka5-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref-keep*

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*

context *Network-Impl* **begin**

Correctness theorem of the final implementation

theorem *edka-imp-correct*:
assumes *VN*: *Graph.V c* $\subseteq$ {*0..*<*N*}
assumes *ABS-PS*: *is-pred-succ ps c*
shows <*emp*> *edka-imp c s t N ps* <λ*fi*. $\exists_A$*f*. *is-rflow N f fi* $*$ $\uparrow$(*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 ps ≤ SPEC isMaxFlow .
    from hn-refine-ref[OF this edka-imp-refine]
    show ?thesis
      by (simp add: hn-refine-def)
  qed
 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 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-lemma [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*)

schematic-lemma [*code*]: *bfs-impl-0 t u l = ?foo*
apply (*subst bfs-impl.code*)
apply (*rewrite* **in** ⌑ *insert-stat-inner-c*)
by (*rule refl*)

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* (*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* (*N*,*fi*) ⇒ ∃$_A$*f*. *Network-Impl.is-rflow* (*ln-α el*) *N f fi* ∗ ↑(*Network.isMaxFlow* (*ln-α el*) *s t f*)
∗ ↑(*ln-invar el* ∧ *Network* (*ln-α el*) *s t* ∧ *Graph.V* (*ln-α el*) ⊆ {*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 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


definition get-flow :: capacity-impl graph ⇒ nat ⇒ Graph.node ⇒ capacity-impl
mtx ⇒ capacity-impl Heap where
  get-flow c N s fi ≡ do {
    imp-nfoldli ([0..<N]) (λ-. return True) (λv cap. do {
      let csv = c (s,v);
      cfsv ← mtx-get N fi (s,v);
      let fsv = csv − cfsv;
      return (cap + fsv)
    }) 0
  }


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 get-flow
  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.