

Flow Networks and the Min-Cut-Max-Flow Theorem

Peter Lammich and S. Reza Sefidgar

June 14, 2026

Abstract

We present a formalization of flow networks and the Min-Cut-Max-Flow theorem. 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.

Contents

1	Introduction	3
2	Flows, Cuts, and Networks	3
2.1	Definitions	3
2.1.1	Flows	3
2.1.2	Cuts	4
2.1.3	Networks	4
2.1.4	Networks with Flows and Cuts	6
2.2	Properties	7
2.2.1	Flows	7
2.2.2	Networks	9
2.2.3	Networks with Flow	9
3	Residual Graph	11
3.1	Definition	11
3.2	Properties	12
4	Augmenting Flows	18
4.1	Augmentation of a Flow	18
4.2	Augmentation yields Valid Flow	19
4.2.1	Capacity Constraint	19
4.2.2	Conservation Constraint	20
4.3	Value of the Augmented Flow	22
5	Augmenting Paths	24
5.1	Definitions	24
5.2	Augmenting Flow is Valid Flow	25
5.3	Value of Augmenting Flow is Residual Capacity	27
6	The Ford-Fulkerson Theorem	27
6.1	Net Flow	28
6.2	Ford-Fulkerson Theorem	29
6.3	Corollaries	32

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 [3] describes a class of algorithms to solve the maximum flow problem. It is based on a corollary of the Min-Cut-Max-Flow theorem [3, 2], which states that a flow is maximal iff there exists no augmenting path.

In this chapter, we present a formalization of flow networks and prove the Min-Cut-Max-Flow theorem, closely following the textbook presentation of Cormen et al. [1]. We have used the Isar [4] proof language to develop human-readable proofs that are accessible even to non-Isabelle experts.

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 preflow on a graph is a labeling of the edges with values from a linearly ordered integral domain, such that:

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

non-deficiency constraint for all nodes except s and t , the incoming flow greater or equal to the outgoing flow.

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

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

```
assumes capacity-const:  $\forall e. 0 \leq f e \wedge f e \leq c e$ 
```

```
assumes no-deficient-nodes:  $\forall v \in V - \{s, t\}.$ 
```

```
( $\sum_{e \in \text{outgoing } v. f e} \leq (\sum_{e \in \text{incoming } v. f e}$ )
```

```
begin
```

end

An *s-t flow* on a graph is a preflow that has no active nodes except source and sink, where a node is *active* iff it has more incoming flow than outgoing flow.

```
locale Flow = Preflow c s t f
  for c :: 'capacity::linordered-idom graph
  and s t :: node
  and f +
  assumes no-active-nodes:
     $\forall v \in V - \{s,t\}. (\sum e \in \text{outgoing } v. f e) \geq (\sum e \in \text{incoming } v. f e)$ 
begin
```

For a flow, inflow equals outflow for all nodes except sink and source. This is called *conservation*.

```
lemma conservation-const:
   $\forall v \in V - \{s, t\}. (\sum e \in \text{incoming } v. f e) = (\sum e \in \text{outgoing } v. f e)$ 
  using no-deficient-nodes no-active-nodes
  by force
```

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

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

```
locale Finite-Preflow = Preflow c s t f + Finite-Graph c
  for c :: 'capacity::linordered-idom graph and s t f
```

```
locale Finite-Flow = Flow c s t f + Finite-Preflow c s t f
  for c :: 'capacity::linordered-idom graph and s t f
```

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. The other partition is implicitly given by the remaining nodes.

```
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.
- There are 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.

Notes on the formalization

- We encode the graph by a mapping c , such that $c(u,v)$ is the capacity of edge (u,v) , or 0 , if there is no edge from u to v . Thus, in the formalization below, we only demand that $c(u,v) \geq 0$ for all u and v .
- We only demand the set of nodes reachable from the source to be finite. Together with the constraint that all nodes lay on a path from the source, this implies that the graph is finite.

```

locale Network = Graph c for c :: 'capacity::linordered-idom graph +
fixes s t :: node
assumes s-node[simp, intro!]: s ∈ V
assumes t-node[simp, intro!]: t ∈ V
assumes s-not-t[simp, intro!]: s ≠ t

assumes cap-non-negative: ∀ u v. c (u, v) ≥ 0
assumes no-incoming-s: ∀ u. (u, s) ∉ E
assumes no-outgoing-t: ∀ u. (t, u) ∉ E
assumes no-parallel-edge: ∀ u v. (u, v) ∈ E ⟶ (v, u) ∉ E
assumes nodes-on-st-path: ∀ v ∈ V. connected s v ∧ connected v t
assumes finite-reachable: finite (reachableNodes s)
begin

```

Edges have positive capacity

```

lemma edge-cap-positive: (u,v)∈E ⟹ c (u,v) > 0
unfolding E-def using cap-non-negative[THEN spec2, of u v] by simp

```

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

```

lemma reachable-is-V[simp]: reachableNodes s = V
proof
show V ⊆ reachableNodes s
unfolding reachableNodes-def using s-node nodes-on-st-path
by auto
qed (simp add: reachable-ss-V)

```

Thus, the network is actually a finite graph.

```

sublocale Finite-Graph
apply unfold-locales
using reachable-is-V finite-reachable by auto

```

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*

lemma *adjacent-not-self*[*simp, intro!*]: $v \notin \text{adjacent-nodes } v$
unfolding *adjacent-nodes-def* **using** *no-self-loop*
by *auto*

A flow is maximal, if it has a maximal value

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

definition *is-max-flow-val* $fv \equiv \exists f. \text{isMaxFlow } f \wedge fv = \text{Flow.val } c \ s \ f$

lemma *t-not-s*[*simp*]: $t \neq s$ **using** *s-not-t* **by** *blast*

The excess of a node is the difference between incoming and outgoing flow.

definition *excess* :: $\text{'capacity flow} \Rightarrow \text{node} \Rightarrow \text{'capacity}$ **where**
 $\text{excess } f \ v \equiv (\sum e \in \text{incoming } v. f \ e) - (\sum e \in \text{outgoing } v. f \ e)$

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 *NPreflow* = *Network* $c \ s \ t$ + *Preflow* $c \ s \ t \ f$
for c :: $\text{'capacity::linordered-idom graph}$ **and** $s \ t \ f$
begin

end

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

lemma (**in** *Network*) *isMaxFlow-alt*:
 $\text{isMaxFlow } f \longleftrightarrow \text{NFlow } c \ s \ t \ f \wedge$
 $(\forall f'. \text{NFlow } c \ s \ t \ f' \longrightarrow \text{Flow.val } c \ s \ f' \leq \text{Flow.val } c \ s \ f)$
unfolding *isMaxFlow-def*
by (*auto simp: NFlow-def Flow-def NPreflow-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 :: $\text{'capacity::linordered-idom graph}$ **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 *Preflow*
begin

Only edges are labeled with non-zero flows

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

lemma *f-non-negative*: $0 \leq f e$
using *capacity-const* **by** (*cases e*) *auto*

lemma *sum-f-non-negative*: $sum f X \geq 0$ **using** *capacity-const*
by (*auto simp: sum-nonneg f-non-negative*)

end — *Preflow*

context *Flow*
begin

We provide a useful equivalent formulation of the conservation constraint.

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

The value of the flow is bounded by the capacity of the outgoing edges of the source node

lemma *val-bounded*:

$-(\sum e \in \text{incoming } s. c e) \leq \text{val}$
 $\text{val} \leq (\sum e \in \text{outgoing } s. c e)$
proof –
have
 $\text{sum } f (\text{outgoing } s) \leq \text{sum } c (\text{outgoing } s)$
 $\text{sum } f (\text{incoming } s) \leq \text{sum } c (\text{incoming } s)$
using *capacity-const* **by** (*auto intro!*: *sum-mono*)
thus $-(\sum e \in \text{incoming } s. c e) \leq \text{val} \quad \text{val} \leq (\sum e \in \text{outgoing } s. c e)$
using *sum-f-non-negative*[*of incoming s*]
using *sum-f-non-negative*[*of outgoing s*]
unfolding *val-def* **by** *auto*
qed

end — Flow

Introduce a flow via the conservation constraint

lemma (*in Graph*) *intro-Flow*:
assumes *cap*: $\forall e. 0 \leq f e \wedge f e \leq c e$
assumes *cons*: $\forall v \in V - \{s, t\}.$
 $(\sum e \in \text{incoming } v. f e) = (\sum e \in \text{outgoing } v. f e)$
shows *Flow c s t f*
using *assms* **by** *unfold-locales auto*

context *Finite-Preflow*
begin

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* \Rightarrow '*capacity*'
assumes $u \in V$
shows $(\sum e \in \text{outgoing } u. f e) = (\sum v \in V. f (u, v))$
apply (*subst sum-outgoing-alt*)
using *assms capacity-const*
by *auto*

lemma *sum-incoming-alt-flow*:
fixes *g* :: *edge* \Rightarrow '*capacity*'
assumes $u \in V$
shows $(\sum e \in \text{incoming } u. f e) = (\sum v \in V. f (v, u))$
apply (*subst sum-incoming-alt*)
using *assms capacity-const*
by *auto*
end — Finite Preflow

2.2.2 Networks

context *Network*

begin

lemmas [*simp*] = *no-incoming-s no-outgoing-t*

lemma *incoming-s-empty*[*simp*]: *incoming s = {}*
unfolding *incoming-def* **using** *no-incoming-s* **by** *auto*

lemma *outgoing-t-empty*[*simp*]: *outgoing t = {}*
unfolding *outgoing-def* **using** *no-outgoing-t* **by** *auto*

lemma *cap-positive*: $e \in E \implies c\ e > 0$
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*)

lemma *card-V-ge2*: $\text{card } V \geq 2$

proof –

have $2 = \text{card } \{s,t\}$ **by** *auto*

also have $\{s,t\} \subseteq V$ **by** *auto*

hence $\text{card } \{s,t\} \leq \text{card } V$ **by** (*rule-tac card-mono*) *auto*

finally show *?thesis* .

qed

lemma *zero-is-flow*: $\text{Flow } c\ s\ t\ (\lambda\ \cdot\ 0)$
using *cap-non-negative* **by** *unfold-locales auto*

lemma *max-flow-val-unique*:

$[[\text{is-max-flow-val } fv1; \text{is-max-flow-val } fv2]] \implies fv1 = fv2$

unfolding *is-max-flow-val-def isMaxFlow-def*

by (*auto simp: antisym*)

end — *Network*

2.2.3 Networks with Flow

context *NPreflow*

begin

sublocale *Finite-Preflow* **by** *unfold-locales*

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

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

proof (*rule ccontr*)

assume $\neg(\forall e \in \text{incoming } s. f\ e = 0)$

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

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

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

lemma *excess-non-negative*: $\forall v \in V - \{s,t\}. excess\ f\ v \geq 0$
unfolding *excess-def* **using** *no-deficient-nodes* **by** *auto*

lemma *excess-nodes-only*: $excess\ f\ v > 0 \implies v \in V$
unfolding *excess-def incoming-def outgoing-def V-def*
using *sum.not-neutral-contains-not-neutral* **by** *fastforce*

lemma *excess-non-negative'*: $\forall v \in V - \{s\}. excess\ f\ v \geq 0$
proof –
have $excess\ f\ t \geq 0$ **unfolding** *excess-def outgoing-def*
by (*auto simp: capacity-const sum-nonneg*)
thus *?thesis* **using** *excess-non-negative* **by** *blast*
qed

lemma *excess-s-non-pos*: $excess\ f\ s \leq 0$
unfolding *excess-def*
by (*simp add: capacity-const sum-nonneg*)

end — Network with preflow

context *NFlow* **begin**
sublocale *Finite-Preflow* **by** *unfold-locales*

There is no outflow from the sink in a network. 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*)

end

end — Theory

3 Residual Graph

```
theory Residual-Graph
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  $\Rightarrow$  - flow  $\Rightarrow$  - 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
```

```
context Network begin
```

```
abbreviation cf-of  $\equiv$  residualGraph c
abbreviation cfE-of f  $\equiv$  Graph.E (cf-of f)
```

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

```
lemma cfE-of-ss-invE: cfE-of cf  $\subseteq$  E  $\cup$  E-1
  unfolding residualGraph-def Graph.E-def
  by auto
```

```
lemma cfE-of-ss-VxV: cfE-of f  $\subseteq$  V  $\times$  V
  unfolding V-def
  unfolding residualGraph-def Graph.E-def
  by auto
```

```
lemma cfE-of-finite[simp, intro!]: finite (cfE-of f)
  using finite-subset[OF cfE-of-ss-VxV] by auto
```

```
lemma cf-no-self-loop: (u, u)  $\notin$  cfE-of f
proof
  assume a1: (u, u)  $\in$  cfE-of f
  have (u, u)  $\notin$  E
```

```

    using no-parallel-edge by blast
  then show False
    using a1 unfolding Graph.E-def residualGraph-def by fastforce
qed

```

end

Let's fix a network with a preflow f on it

```
context NPreflow
```

```
begin
```

We abbreviate the residual graph by cf .

```

abbreviation cf ≡ residualGraph c f
sublocale cf: Graph cf .
lemmas cf-def = residualGraph-def[of c f]

```

3.2 Properties

```
lemmas cfE-ss-invE = cfE-of-ss-invE[of f]
```

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  $\langle (v, u) \in E \rangle$  by (auto)
    then have  $cf(v, u) \neq 0$  using  $\langle (v, u) \in E \rangle$  unfolding E-def by auto
    thus ?thesis unfolding Graph.E-def by auto
  next
    assume  $f(v, u) \neq 0$ 
    then have  $cf(u, v) = f(v, u)$  unfolding residualGraph-def
      using  $\langle (v, u) \in E \rangle$  no-parallel-edge by auto
    then have  $cf(u, v) \neq 0$  using  $\langle f(v, u) \neq 0 \rangle$  by auto
    thus ?thesis unfolding Graph.E-def by auto
  qed
} ultimately show  $u \in cf.V$  unfolding cf.V-def by auto
qed
next
  show Graph.V  $cf \subseteq V$  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:

```

sublocale cf: Finite-Graph cf
  by unfold-locales 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  $(u, v) \in E$ 
    then have  $cf(u, v) = c(u, v) - f(u, v)$  unfolding cf-def by auto
    hence  $cf(u, v) \geq 0$ 
      using capacity-const cap-non-negative by auto
  } moreover {
    assume  $(v, u) \in E$ 
    then have  $cf(u, v) = f(v, u)$ 
      using no-parallel-edge unfolding cf-def by auto
    hence  $cf(u, v) \geq 0$ 
      using capacity-const by auto
  } moreover {
    assume  $(u, v) \notin E \quad (v, u) \notin E$ 
    hence  $cf(u, v) \geq 0$  unfolding residualGraph-def by simp
  } ultimately show  $cf(u, v) \geq 0$  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 ∈ cf.E ⇒ cf e > 0
proof –
  assume e ∈ cf.E
  hence cf e ≠ 0 unfolding cf.E-def by auto
  thus ?thesis using resE-nonNegative by (meson eq-iff not-le)
qed

```

```

lemma reverse-flow: Preflow cf s t f' ⇒ ∀ (u, v) ∈ E. f' (v, u) ≤ f (u, v)

```

```

proof –
  assume asm: Preflow cf s t f'
  then interpret f': Preflow cf s t f' .

  {
    fix u v
    assume (u, v) ∈ E

    then have cf (v, u) = f (u, v)
      unfolding residualGraph-def using no-parallel-edge by auto
    moreover have f' (v, u) ≤ cf (v, u) using f'.capacity-const by auto
    ultimately have f' (v, u) ≤ f (u, v) by metis
  }
  thus ?thesis by auto
qed

```

```

definition (in Network) flow-of-cf cf e ≡ (if (e ∈ E) then c e - cf e else 0)

```

```

lemma (in NPreflow) E-ss-cfinvE: E ⊆ Graph.E cf ∪ (Graph.E cf)-1
  unfolding residualGraph-def Graph.E-def
  apply (clarsimp)
  using no-parallel-edge
  unfolding E-def
  apply simp
  done

```

Nodes with positive excess must have an outgoing edge in the residual graph.

Intuitively: The excess flow must come from somewhere.

```

lemma active-has-cf-outgoing: excess f u > 0 ⇒ cf.outgoing u ≠ {}

```

unfolding *excess-def*
proof –
assume $0 < \text{sum } f \text{ (incoming } u) - \text{sum } f \text{ (outgoing } u)$
hence $0 < \text{sum } f \text{ (incoming } u)$
by (*metis diff-gt-0-iff-gt linorder-neqE-linordered-idom linorder-not-le sum-f-non-negative*)
with *f-non-negative* **obtain** $e \text{ where } e \in \text{incoming } u \quad f \ e > 0$
by (*meson not-le sum-nonpos*)
then obtain $v \text{ where } (v,u) \in E \quad f \ (v,u) > 0$ **unfolding** *incoming-def* **by** *auto*
hence $cf \ (u,v) > 0$ **unfolding** *residualGraph-def* **by** *auto*
thus *?thesis* **unfolding** *cf.outgoing-def cf.E-def* **by** *fastforce*
qed

end — Network with preflow

locale *RPreGraph* — Locale that characterizes a residual graph of a network
 $= \text{Network} +$
fixes *cf*
assumes *EX-RPG*: $\exists f. \text{NPreflow } c \ s \ t \ f \wedge cf = \text{residualGraph } c \ f$
begin

lemma *this-loc-rpg*: *RPreGraph* $c \ s \ t \ cf$
by *unfold-locales*

definition $f \equiv \text{flow-of-cf } cf$

lemma *f-unique*:
assumes *NPreflow* $c \ s \ t \ f'$
assumes *A*: $cf = \text{residualGraph } c \ f'$
shows $f' = f$

proof –
interpret f' : *NPreflow* $c \ s \ t \ f'$ **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-NPreflow*: *NPreflow* $c \ s \ t \ (\text{flow-of-cf } cf)$
apply (*fold f-def*)
using *EX-RPG f-unique* **by** *metis*

```

sublocale  $f$ :  $NPreflow\ c\ s\ t\ f$  unfolding  $f$ -def by (rule is-NPreflow)

lemma  $rg$ -is-cf[simp]:  $residualGraph\ c\ f = cf$ 
using EX-RPG  $f$ -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  $E$ -ss-cfinvE:  $E \subseteq cf.E \cup cf.E^{-1}$ 
using  $f.E$ -ss-cfinvE by simp

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

lemma  $resE$ -nonNegative:  $cf\ e \geq 0$ 
using  $f.resE$ -nonNegative by auto

end

context  $NPreflow$  begin
lemma  $is$ -RPreGraph:  $RPreGraph\ c\ s\ t\ cf$ 
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 NPreflow)
  flow-of-cf (residualGraph c f) = f
  by (rule fo-rg-inv)

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
  sublocale RPreGraph
  proof
    from EX-RG obtain f where
      NFlow c s t f and [simp]: cf = residualGraph c f by auto
    then interpret NFlow c s t f by simp

    show  $\exists f. NPreflow\ c\ s\ t\ f \wedge cf = residualGraph\ c\ f$ 
    apply (rule exI[where x=f])
    apply simp
    by unfold-locales
  qed

lemma this-loc: RGraph c s t cf
  by unfold-locales
lemma this-loc-rpg: RPreGraph c s t cf
  by unfold-locales

lemma is-NFlow: NFlow c s t (flow-of-cf cf)
  using EX-RG f-unique is-NPreflow NFlow.axioms(1)
  apply (fold f-def) by force

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

The value of the flow can be computed from the residual graph.
lemma val-by-cf:  $val = (\sum (u,v) \in outgoing\ s. cf\ (v,u))$ 
proof —
  have f (s,v) = cf (v,s) for v
  unfolding cf-def by auto
  thus ?thesis

```

```

    unfolding val-alt outgoing-def
    by (auto intro!: sum.cong)
qed

end — Network with Flow

lemma (in RPreGraph) maxflow-imp-rgraph:
  assumes isMaxFlow (flow-of-cf cf)
  shows RGraph c s t cf
proof —
  from assms interpret Flow c s t f
  unfolding isMaxFlow-def by (simp add: f-def)

  interpret NFlow c s t f by unfold-locales

  show ?thesis
  apply unfold-locales
  apply (rule exI[of - f])
  apply (simp add: NFlow-axioms)
  done
qed

end — Theory

```

4 Augmenting Flows

```

theory Augmenting-Flow
imports Residual-Graph
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  $\Rightarrow$  '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 al.:

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

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 $f'\text{-flow}: Flow\ cf\ s\ t\ f'$

begin

interpretation $f': Flow\ cf\ s\ t\ f'$ **by** (*rule* $f'\text{-flow}$)

4.2.1 Capacity Constraint

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

lemma *augment-flow-presv-cap*:

shows $0 \leq (f \uparrow f')(u, v) \wedge (f \uparrow f')(u, v) \leq c(u, v)$

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 $cf(v, u) \geq f'(v, u)$ **using** $f'.capacity\ const$ **by** *auto*

finally have $f'(v, u) \leq f(u, v)$.

have $(f \uparrow f')(u, v) = f(u, v) + f'(u, v) - f'(v, u)$

by (*auto simp: augment-def*)

also have $\dots \geq f(u, v) + f'(u, v) - f(u, v)$

using $\langle f'(v, u) \leq f(u, v) \rangle$ **by** *auto*

also have $\dots = f'(u, v)$ **by** *auto*

also have $\dots \geq 0$ **using** $f'.capacity\ const$ **by** *auto*

finally show $(f \uparrow f')(u, v) \geq 0$.

have $(f \uparrow f')(u, v) = f(u, v) + f'(u, v) - f'(v, u)$

by (*auto simp: augment-def*)

also have $\dots \leq f(u, v) + f'(u, v)$ **using** $f'.capacity\ const$ **by** *auto*

also have $\dots \leq f(u, v) + cf(u, v)$ **using** $f'.capacity\ const$ **by** *auto*

also have $\dots = f(u, v) + c(u, v) - f(u, v)$

by (*auto simp: residualGraph-def*)

also have $\dots = c(u,v)$ **by** *auto*
finally show $(f \uparrow f')(u, v) \leq c(u, v)$.
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.

Note that the term $E \text{ `` } \{u\}$ characterizes the successor nodes of u , and $E^{-1} \text{ `` } \{u\}$ characterizes the predecessor nodes of u .

private lemma *split-rflow-outgoing*:

$$(\sum_{v \in cf.E \text{ `` } \{u\}}. f'(u,v)) = (\sum_{v \in E \text{ `` } \{u\}}. f'(u,v)) + (\sum_{v \in E^{-1} \text{ `` } \{u\}}. f'(u,v))$$

(**is** ?LHS = ?RHS)

proof –

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

have ?LHS = $(\sum_{v \in E \text{ `` } \{u\} \cup E^{-1} \text{ `` } \{u\}}. f'(u,v))$

apply (*rule sum.mono-neutral-left*)

using *cfE-ss-invE*

by (*auto intro: finite-Image*)

also have $\dots = ?RHS$

apply (*subst sum.union-disjoint[OF - - DJ]*)

by (*auto intro: finite-Image*)

finally show ?LHS = ?RHS .

qed

private lemma *split-rflow-incoming*:

$$(\sum_{v \in cf.E^{-1} \text{ `` } \{u\}}. f'(v,u)) = (\sum_{v \in E \text{ `` } \{u\}}. f'(v,u)) + (\sum_{v \in E^{-1} \text{ `` } \{u\}}. f'(v,u))$$

(**is** ?LHS = ?RHS)

proof –

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

have ?LHS = $(\sum_{v \in E \text{ `` } \{u\} \cup E^{-1} \text{ `` } \{u\}}. f'(v,u))$

apply (*rule sum.mono-neutral-left*)

using *cfE-ss-invE*

by (*auto intro: finite-Image*)

also have $\dots = ?RHS$

apply (*subst sum.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 $u :: \text{node}$

assumes $U\text{-ASM}: u \in V - \{s, t\}$

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 $(\sum v \in E^{\leftarrow\{u\}}. f'(u, v)) - (\sum v \in E^{\leftarrow\{u\}}. f'(v, u))$
 $= (\sum v \in E^{-1\leftarrow\{u\}}. f'(v, u)) - (\sum v \in E^{-1\leftarrow\{u\}}. f'(u, v))$
(is ?LHS = ?RHS is ?A - ?B = ?RHS)

proof –

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

have $?A = (\sum v \in cf.E^{\leftarrow\{u\}}. f'(u, v)) - (\sum v \in E^{-1\leftarrow\{u\}}. f'(u, v))$

by (*simp add: split-rflow-outgoing*)

also have $(\sum v \in cf.E^{\leftarrow\{u\}}. f'(u, v)) = (\sum v \in cf.E^{-1\leftarrow\{u\}}. f'(v, u))$

using $U\text{-ASM}$

by (*simp add: f'.conservation-const-pointwise*)

finally have $?A = (\sum v \in cf.E^{-1\leftarrow\{u\}}. f'(v, u)) - (\sum v \in E^{-1\leftarrow\{u\}}. f'(u, v))$

by *simp*

moreover

have $?B = (\sum v \in cf.E^{-1\leftarrow\{u\}}. f'(v, u)) - (\sum v \in E^{-1\leftarrow\{u\}}. f'(v, u))$

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 $(\sum e \in \text{outgoing } u. \text{augment } f' e) = (\sum e \in \text{incoming } u. \text{augment } f' e)$

(is ?LHS = ?RHS)

proof –

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

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

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

have $?LHS = (\sum v \in ?Vo. \text{augment } f'(u, v))$

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: sum-subtractf sum.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: sum-subtractf sum.distrib*)
also have ...
 $= (\sum_{v \in ?Vi} \text{augment } f'(v, u))$
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. [1] 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 (*rule-tac intro-Flow*) *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* .

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 *sum-simp-setup[simp]* =

```

sum-outgoing-alt[OF capacity-const] s-node
sum-incoming-alt[OF capacity-const]
cf.sum-outgoing-alt[OF f'.capacity-const]
cf.sum-incoming-alt[OF f'.capacity-const]
sum-outgoing-alt[OF f''.capacity-const]
sum-incoming-alt[OF f''.capacity-const]
sum-subtractf sum.distrib

```

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.

```

have aux1: f'(u,v) = 0 if (u,v)∉E (v,u)∉E for u v
proof –
  from that cfE-ss-invE have (u,v)∉cf.E by auto
  thus f'(u,v) = 0 by auto
qed

```

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 sum.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 f''.val = val + f'.val .
qed

```

Note, there is also an automatic proof. When creating the above explicit proof, this automatic one has been used to extract meaningful subgoals, abusing Isabelle as a term rewriter.

```

lemma Flow.val c s (f↑f') = val + Flow.val cf s f'
proof –
  interpret f'': Flow c s t f↑f' using augment-flow-presv .

```

```

have aux1: f'(u,v) = 0 if A: (u,v)∉E (v,u)∉E for u v
proof –
  from A cfE-ss-invE have (u,v)∉cf.E by auto
  thus f'(u,v) = 0 by auto
qed

```

```

show ?thesis
  unfolding val-def f'.val-def f''.val-def
  apply (simp del:
    add:
      sum-outgoing-alt[OF capacity-const] s-node
      sum-incoming-alt[OF capacity-const]
      sum-outgoing-alt[OF f''.capacity-const]
      sum-incoming-alt[OF f''.capacity-const]

```

```

    cf.sum-outgoing-alt[OF f'.capacity-const]
    cf.sum-incoming-alt[OF f'.capacity-const]
    sum-subtractf[symmetric] sum.distrib[symmetric]
  )
  apply (rule sum.cong)
  apply (auto simp: augment-def no-parallel-edge aux1)
done
qed

```

end — Augmenting flow
end — Network flow

end — Theory

5 Augmenting Paths

```

theory Augmenting-Path
imports Residual-Graph
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 preflow f on it.

```

context NPreflow
begin

```

5.1 Definitions

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

```

definition isAugmentingPath :: path  $\Rightarrow$  bool
where isAugmentingPath p  $\equiv$  cf.isSimplePath s p t

```

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

```

definition resCap :: path  $\Rightarrow$  'capacity
where resCap p  $\equiv$  Min {cf e | e. e  $\in$  set p}

```

```

lemma resCap-alt: resCap p = Min (cf`set p)
  — Useful characterization for finiteness arguments
  unfolding resCap-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  $\Rightarrow$  'capacity flow

```

where $augmentingFlow\ p \equiv \lambda(u, v).$
if $(u, v) \in (set\ p)$ **then**
 $resCap\ p$
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 $resCap-gzero-aux: cf.isPath\ s\ p\ t \implies 0 < resCap\ p$
proof –
assume $PATH: cf.isPath\ s\ p\ t$
hence $set\ p \neq \{\}$ **using** $s-not-t$ **by** $(auto)$
moreover **have** $\forall e \in set\ p. cf\ e > 0$
using $cf.isPath-edgeset[OF\ PATH]$ $resE-positive$ **by** $(auto)$
ultimately show $?thesis$ **unfolding** $resCap-alt$ **by** $(auto)$
qed

lemma $resCap-gzero: isAugmentingPath\ p \implies 0 < resCap\ p$
using $resCap-gzero-aux[of\ p]$
by $(auto\ simp: isAugmentingPath-def\ cf.isSimplePath-def)$

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

lemma $sum-augmenting-alt:$
assumes $finite\ A$
shows $(\sum e \in A. (augmentingFlow\ p)\ e)$
 $= resCap\ p * of-nat\ (card\ (A \cap set\ p))$
proof –
have $(\sum e \in A. (augmentingFlow\ p)\ e) = sum\ (\lambda-. resCap\ p)\ (A \cap set\ p)$
apply $(subst\ sum.inter-restrict)$
apply $(auto\ simp: augmentingFlow-def\ assms)$
done
thus $?thesis$ **by** $auto$
qed

lemma $augFlow-resFlow: isAugmentingPath\ p \implies Flow\ cf\ s\ t\ (augmentingFlow\ p)$
proof $(rule\ cf.intro-Flow; intro\ allI\ ballI)$
assume $AUG: isAugmentingPath\ p$
hence $SPATH: cf.isSimplePath\ s\ p\ t$ **by** $(simp\ add: isAugmentingPath-def)$
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 resCap p ≤ cf e unfolding resCap-alt by auto
  moreover have (augmentingFlow p) e = resCap p
    unfolding augmentingFlow-def using ⟨e ∈ set p⟩ by auto
  moreover have 0 < resCap p using resCap-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: sum-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*:

isAugmentingPath p \implies *Flow.val cf s (augmentingFlow p) = resCap p*

proof –

assume *AUG*: *isAugmentingPath p*

with *augFlow-resFlow* **interpret** *f*: *Flow cf s t augmentingFlow p* .

note *AUG*

hence *SPATH*: *cf.isSimplePath s p t* **by** (*simp add: isAugmentingPath-def*)

hence *PATH*: *cf.isPath s p t* **by** (*simp add: cf.isSimplePath-def*)

then obtain *v p'* **where** $p=(s,v)\#p'$ $(s,v)\in cf.E$

using *s-not-t* **by** (*cases p*) *auto*

hence *cf.outgoing s* \cap *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* \cap *set p* = $\{\}$ **using** *SPATH no-incoming-s*

by (*auto*)

simp: cf.incoming-def $\langle p=(s,v)\#p' \rangle$ *in-set-conv-decomp[where xs=p]*

simp: cf.isSimplePath-append cf.isSimplePath-cons)

ultimately show *?thesis*

unfolding *f.val-def*

by (*auto simp: sum-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-idom graph **and** *s t f k*
begin

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

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* $\equiv (\sum e \in \text{outgoing}' k. f e) - (\sum e \in \text{incoming}' k. f e)$

We can show that the net flow equals the value of the flow. Note: Cormen et al. [1] 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* = $\{(u, v). u \in k \wedge v \in k \wedge (u, v) \in E\}$
let *?AOG* = $\{(u, v). u \in k \wedge (u, v) \in E\}$
let *?AIN* = $\{(v, u) \mid u v. u \in k \wedge (v, u) \in E\}$
let *?SOG* = $\lambda u. (\sum e \in \text{outgoing } u. f e)$
let *?SIN* = $\lambda u. (\sum e \in \text{incoming } u. f e)$
let *?SOG'* = $(\sum e \in \text{outgoing}' k. f e)$
let *?SIN'* = $(\sum e \in \text{incoming}' k. f e)$

Some setup to make finiteness reasoning implicit

note $[[\text{simproc } \text{finite-Collect}]]$

have

netFlow = *?SOG'* + $(\sum e \in ?LCL. f e) - (?SIN' + (\sum e \in ?LCL. f e))$
(is - = *?SAOG* **-** *?SAIN)*

using *netFlow-def* **by** *auto*

also have *?SAOG* = $(\sum y \in k - \{s\}. ?SOG y) + ?SOG s$

proof –

have *?SAOG* = $(\sum e \in (\text{outgoing}' k \cup ?LCL). f e)$

by *(rule sum.union-disjoint[symmetric]) (auto simp: outgoing'-def)*

also have *outgoing' k* $\cup ?LCL = (\bigcup y \in k - \{s\}. \text{outgoing } y) \cup \text{outgoing } s$

by *(auto simp: outgoing-def outgoing'-def s-in-cut)*

also have $(\sum e \in (\bigcup (\text{outgoing}' (k - \{s\})) \cup \text{outgoing } s). f e)$

= $(\sum e \in (\bigcup (\text{outgoing}' (k - \{s\}))). f e) + (\sum e \in \text{outgoing } s. f e)$

by *(rule sum.union-disjoint)*

(auto simp: outgoing-def intro: finite-Image)

also have $(\sum e \in (\bigcup (\text{outgoing}' (k - \{s\}))). f e)$

= $(\sum y \in k - \{s\}. ?SOG y)$

by *(rule sum.UNION-disjoint)*

(auto simp: outgoing-def intro: finite-Image)

finally show *?thesis* .
qed
also have $?SAIN = (\sum y \in k - \{s\}. ?SIN y) + ?SIN s$
proof –
have $?SAIN = (\sum e \in (incoming' k \cup ?LCL). f e)$
by (*rule sum.union-disjoint[symmetric]*) (*auto simp: incoming'-def*)
also have $incoming' k \cup ?LCL = (\bigcup y \in k - \{s\}. incoming y) \cup incoming s$
by (*auto simp: incoming-def incoming'-def s-in-cut*)
also have $(\sum e \in (\bigcup (incoming' (k - \{s\})) \cup incoming s). f e)$
 $= (\sum e \in (\bigcup (incoming' (k - \{s\}))). f e) + (\sum e \in incoming s. f e)$
by (*rule sum.union-disjoint*)
(auto simp: incoming-def intro: finite-Image)
also have $(\sum e \in (\bigcup (incoming' (k - \{s\}))). f e)$
 $= (\sum y \in k - \{s\}. ?SIN y)$
by (*rule sum.UNION-disjoint*)
(auto simp: incoming-def intro: finite-Image)
finally show *?thesis* .
qed
finally have $netFlow =$
 $((\sum y \in k - \{s\}. ?SOG y) + ?SOG s)$
 $- ((\sum y \in k - \{s\}. ?SIN y) + ?SIN s)$
(is netFlow = ?R) .
also have $?R = ?SOG s - ?SIN s$
proof –
have $(\bigwedge u. u \in k - \{s\} \implies ?SOG u = ?SIN u)$
using *conservation-const cut-ss-V t-ni-cut* **by force**
thus *?thesis* **by auto**
qed
finally 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 ≤ cap*

proof –

have $(\sum e \in outgoing' k. f e) \leq (\sum e \in outgoing' k. c e)$ **(is ?L ≤ ?R)**
using *capacity-const* **by** (*metis sum-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 sum-nonneg*)
ultimately show *?thesis* **by simp**

qed

end — Cut

6.2 Ford-Fulkerson Theorem

context *NFlow* **begin**

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

lemma *fofu-I-II*: $isMaxFlow\ f \implies \neg (\exists\ p.\ isAugmentingPath\ p)$

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*: $\neg \neg (\exists\ p.\ isAugmentingPath\ p)$

then obtain *p* **where** *obt*: $isAugmentingPath\ p$ **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*

resCap-gzero isAugmentingPath-def cf.isSimplePath-def **by** *auto*

have $NFlow\ c\ s\ t$ (*augment (augmentingFlow p)*)

using *fct1 augment-flow-presv Network-axioms*

unfolding *Flow-def NFlow-def NPreFlow-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*:

$\neg (\exists\ p.\ isAugmentingPath\ p) \implies \exists\ k'.\ NCut\ c\ s\ t\ k' \wedge val = NCut.cap\ c\ k'$

proof (*intro exI conjI*)

let $?S = cf.reachableNodes\ s$

assume *asm*: $\neg (\exists\ p.\ isAugmentingPath\ p)$

hence $t \notin ?S$

unfolding *isAugmentingPath-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 $(u,v) \in outgoing'\ ?S$

hence $(u,v) \in E\ u \in ?S\ v \notin ?S$

by (*auto simp: outgoing'-def*)

assume $f\ (u,v) \neq c\ (u,v)$

hence $f\ (u,v) < c\ (u,v)$

using *capacity-const* **by** (*metis (no-types) eq-iff not-le*)

hence $cf(u, v) \neq 0$
unfolding *residualGraph-def* **using** $\langle (u, v) \in E \rangle$ **by** *auto*
hence $(u, v) \in cf.E$ **unfolding** *cf.E-def* **by** *simp*
hence $v \in ?S$ **using** $\langle u \in ?S \rangle$ **by** (*auto intro: cf.reachableNodes-append-edge*)
thus *False* **using** $\langle v \notin ?S \rangle$ **by** *auto*
qed
hence $(\sum e \in outgoing' ?S. f e) = cap$
unfolding *cap-def* **by** *auto*
moreover
have $\forall (u, v) \in incoming' ?S. f(u, v) = 0$
proof (*rule ballI, rule ccontr, clarify*) — Proof by contradiction
fix $u v$
assume $(u, v) \in incoming' ?S$
hence $(u, v) \in E \quad u \notin ?S \quad v \in ?S$ **by** (*auto simp: incoming'-def*)
hence $(v, u) \notin E$ **using** *no-parallel-edge* **by** *auto*

assume $f(u, v) \neq 0$
hence $cf(v, u) \neq 0$
unfolding *residualGraph-def* **using** $\langle (u, v) \in E \rangle \langle (v, u) \notin E \rangle$ **by** *auto*
hence $(v, u) \in cf.E$ **unfolding** *cf.E-def* **by** *simp*
hence $u \in ?S$ **using** $\langle v \in ?S \rangle$ *cf.reachableNodes-append-edge* **by** *auto*
thus *False* **using** $\langle u \notin ?S \rangle$ **by** *auto*
qed
hence $(\sum e \in incoming' ?S. f e) = 0$
unfolding *cap-def* **by** *auto*
ultimately show $val = cap$
unfolding *flow-value[symmetric] netFlow-def* **by** *simp*
qed

lemma *fofu-III-I*:
 $\exists k. NCut\ c\ s\ t\ k \wedge val = NCut.cap\ c\ k \implies isMaxFlow\ f$
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 $fc': Flow\ c\ s\ t\ f'$.
interpret $fc': NFlowCut\ c\ s\ t\ f'\ k$ **by** *intro-locales*

have $fc'.val \leq cap$ **using** *fc'.weak-duality* .
also note $\langle val = cap \rangle$ *[symmetric]*
finally have $fc'.val \leq val$.
}
thus $isMaxFlow\ f$ **unfolding** *isMaxFlow-def*

by *simp unfold-locales*
qed

Finally we can state the Ford-Fulkerson theorem:

theorem *ford-fulkerson*: **shows**
 $isMaxFlow\ f \longleftrightarrow$
 $\neg\ Ex\ isAugmentingPath\ \mathbf{and}\ \neg\ Ex\ isAugmentingPath \longleftrightarrow$
 $(\exists\ k.\ NCut\ c\ s\ t\ k \wedge\ val = NCut.cap\ c\ k)$
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.

corollary *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.

corollary *noAugPath-iff-maxFlow*: $(\nexists\ p.\ isAugmentingPath\ p) \longleftrightarrow isMaxFlow\ f$
using *ford-fulkerson* **by** *blast*

end — Network with flow

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

corollary (in *Network*) *maxFlow-minCut*: $[[isMaxFlow\ f; isMinCut\ c\ s\ t\ k]]$
 $\implies Flow.val\ c\ s\ f = NCut.cap\ c\ k$

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* $\langle isMaxFlow\ f \rangle$

obtain *k'* **where** *NCut c s t k'* **and** *val = NCut.cap c k'*

by *blast*

thus *val = cap*

using $\langle isMinCut\ c\ s\ t\ k \rangle$ *weak-duality*

unfolding *isMinCut-def* **by** *auto*

qed

end — Theory

References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [2] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *IEEE Transactions on Information Theory*, 2(4):117–119, dec 1956.
- [3] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.
- [4] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184. Springer, 1999.