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

Peter Lammich and S. Reza Sefidgar

June 11, 2019

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.
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 ⇒ 'capacity

**locale** Preflow = Graph c for c :: 'capacity::linordered-idom graph +

- fixes \( s \) \( t \) :: node
- fixes \( f \) :: 'capacity flow

- **assumes** capacity-const: \( \forall e. \ 0 \leq f e \land f e \leq c e \)
- **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
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** \( \text{Flow} = \text{Preflow} \ c \ s \ t \ f \)  
*for* \( c :: \text{capacity::linordered-idom graph} \)  
*and* \( s \ t :: \text{node} \)  
*and* \( f :: \text{non-active-nodes} \):  
\[ \forall v \in V - \{s,t\}, (\sum e \in \text{outgoing} \ v \ f e) \geq (\sum e \in \text{incoming} \ v \ f e) \]

**begin**

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

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

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

**definition** \( \text{val} :: \text{capacity} \)  
*where* \( \text{val} \equiv (\sum e \in \text{outgoing} \ s \ f e) - (\sum e \in \text{incoming} \ s \ f e) \)

**end**

**locale** \( \text{Finite-Preflow} = \text{Preflow} \ c \ s \ t \ f + \text{Finite-Graph} \ c \)  
*for* \( c :: \text{capacity::linordered-idom graph} \)  
*and* \( s \ t \ f \)

**locale** \( \text{Finite-Flow} = \text{Flow} \ c \ s \ t \ f + \text{Finite-Preflow} \ c \ s \ t \ f \)  
*for* \( c :: \text{capacity::linordered-idom graph} \)  
*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** \( \text{cut} = \text{node set} \)

**locale** \( \text{Cut} = \text{Graph} + \)  
*fixes* \( k :: \text{cut} \)  
*assumes* \( \text{cut-ss-V:} \ k \subseteq V \)

### 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 $\in$ V
assumes t-node[simp, intro!]: t $\in$ V
assumes s-not-t[simp, intro!]: s $\neq$ t
assumes cap-non-negative: $\forall$ u v. c (u, v) $\geq$ 0
assumes no-incoming-s: $\forall$ u. (u, s) $\notin$ E
assumes no-outgoing-t: $\forall$ u. (t, u) $\notin$ E
assumes no-parallel-edge: $\forall$ u v. (u, v) $\in$ E $\rightarrow$ (v, u) $\notin$ E
assumes nodes-on-st-path: $\forall$ v $\in$ V. connected s v $\land$ connected v t
assumes finite-reachable: finite (reachableNodes s)
begin

Edges have positive capacity

lemma edge-cap-positive: (u,v)$\in$E $\Rightarrow$ 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 $\subseteq$ 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
Our assumptions imply that there are no self loops

\[
\text{lemma no-self-loop: } \forall u. (u, u) \notin E
\]

using \texttt{no-self-loop} by auto

\[
\text{lemma adjacent-not-self[simp, intro!]: } v \notin \text{adjacent-nodes } v
\]

unfolding adjacent-nodes-def using \texttt{no-self-loop}

by auto

A flow is maximal, if it has a maximal value

\[
\text{definition isMaxFlow ::= } \text{flow } \Rightarrow \text{bool}
\]

\[
\text{where isMaxFlow } f \equiv \text{Flow } c \text{ s } f \land (
(\forall f'. \text{Flow } c \text{ s } f' \rightarrow \text{Flow.val } c \text{ s } f' \leq \text{Flow.val } c \text{ s } f)
)\]

\[
\text{definition is-max-flow-val } \text{fe } \equiv \exists f. \text{isMaxFlow } f \land \text{fv } = \text{Flow.val } c \text{ s } f
\]

\[
\text{lemma t-not-s[simp]: } t \neq s \text{ using } \texttt{s-not-t by blast}
\]

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

\[
\text{definition excess ::= 'capacity flow } \Rightarrow \text{node } \Rightarrow \text{'capacity where }
\]

\[
\text{excess } f \ v \equiv (\sum e \in \text{incoming } v \ . \ f \ e) - (\sum e \in \text{outgoing } v \ . \ f \ e)
\]

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

\[
\text{locale NPreflow } = \text{Network } c \text{ s } t + \text{Preflow } c \text{ s } t \ f
\]

\[
\text{for } c :: 'capacity::linordered-idom graph \text{ and } s \ t \ f
\]

begin

end

\[
\text{locale NFlow } = \text{NPreflow } c \text{ s } t \ f + \text{Flow } c \text{ s } t \ f
\]

\[
\text{for } c :: 'capacity::linordered-idom graph \text{ and } s \ t \ f
\]

lemma (in Network) isMaxFlow-alt:

\[
isMaxFlow f \leftarrow \text{NFlow } c \text{ s } t \ f \land
(\forall f'. \text{NFlow } c \text{ s } t \ f' \rightarrow \text{Flow.val } c \text{ s } f' \leq \text{Flow.val } c \text{ s } f)
\]

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

\[
\text{locale NCut } = \text{Network } c \text{ s } t + \text{Cut } c \ k
\]
for \( c :: 'capacity::linordered-idom\) graph and \( s\ t\ k\) +
assumes \( s\text{-in-cut}: s \in k\)
assumes \( t\text{-ni-cut}: t \notin k\)
begin

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

definition \( \text{cap} :: 'capacity\)
where \( \text{cap} \equiv (\sum e \in '\text{outgoing}\ k\). c\ e)\)
end

A minimum cut is a cut with minimum capacity.

definition \( \text{isMinCut} :: -\ graph \Rightarrow \text{nat} \Rightarrow \text{cut} \Rightarrow \text{bool}\)
where \( \text{isMinCut} c\ s\ t\ k \equiv \text{NCut}\ c\ s\ t\ k \wedge\)
\( (\forall k'.\ \text{NCut}\ c\ s\ t\ k' \longrightarrow \text{NCut}.\text{cap}\ c\ k \leq \text{NCut}.\text{cap}\ c\ k')\)

2.2 Properties

2.2.1 Flows

context \( \text{Preflow}\)
begin

Only edges are labeled with non-zero flows

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

lemma \( \text{f-non-negative}: 0 \leq f\ e\)
using \( \text{capacity-const}\) by \( (\text{cases}\ e)\) \( \text{auto}\)

lemma \( \text{sum-f-non-negative}: \sum f\ X \geq 0\) using \( \text{capacity-const}\)
by \( (\text{auto simp: sum-nonneg f-non-negative})\)
end — \( \text{Preflow}\)

context \( \text{Flow}\)
begin

We provide a useful equivalent formulation of the conservation constraint.

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

The value of the flow is bounded by the capacity of the outgoing edges of the source node
lemma val-bounded:
\[-(\sum_{e \in \text{incoming } s.} c_e) \leq \text{val} \leq (\sum_{e \in \text{outgoing } s.} c_e)\]

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 \[-(\sum_{e \in \text{incoming } s.} c_e) \leq \text{val} \leq (\sum_{e \in \text{outgoing } s.} c_e)\]
  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 \([simp] = no\text{-}incoming\text{-}s no\text{-}outgoing\text{-}t\)

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

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

lemma cap\text{\textendash}positive: \(e \in E \Rightarrow c e > 0\)
  unfolding E\text{-}def using cap\text{-}non\text{-}negative le\text{-}neq\text{-}trans by fastforce

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

lemma card\text{-}V\text{\textendash}ge\text{\textendash}2: \(\text{card V} \geq 2\)
  proof
    have 2 = card \(\{s, t\}\) by auto
    also have \(\{s, t\} \subseteq V\) by auto
    hence card \(\{s, t\}\) \(\leq\) card V by (rule-tac card\text{-}mono) auto
    finally show \(?thesis\).
   qed

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

lemma max\text{\textendash}flow\text{\textendash}val\text{\textendash}unique:
  \([\text{is\text{-}max\text{-}flow\text{-}val fv1}; \text{is\text{-}max\text{-}flow\text{-}val fv2}\] \(\Rightarrow\) \(fv1 = fv2\)
  unfolding is\text{-}max\text{\textendash}flow\text{-}val\text{\textendash}def isMaxFlow\text{\textendash}def
  by (auto simp: antisym)

end — Network

2.2.3 Networks with Flow

context NPreflow

begin

sublocale Finite\text{-}Preflow by unfold\text{\textendash}locales

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

lemma no\text{-}inflow\text{-}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: \( e \in \text{incoming } s \land f_e \neq 0 \) 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: \( \forall e \in \text{outgoing } t. \ f_e = 0 \)
proof (rule ccontr)
  assume \( \neg (\forall e \in \text{outgoing } t. \ f_e = 0) \)
  then obtain \( e \) where obt1: \( e \in \text{outgoing } t \land f_e \neq 0 \) 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: \( \forall v \in V - \{s,t\}. \ \text{excess } f_v \geq 0 \)
  unfolding excess-def using no-deficient-nodes by auto

lemma excess-nodes-only: \( \text{excess } f_v > 0 \implies v \in V \)
  unfolding excess-def incoming-def outgoing-def V-def
  using sum.not-neutral-contains-not-neutral by fastforce

lemma excess-non-negative': \( \forall v \in V - \{s\}. \ \text{excess } f_v \geq 0 \)
proof
  have \( \text{excess } f_t \geq 0 \) unfolding excess-def outgoing-def
    by (auto simp: capacity-const sum-nonneg)
  thus \( \text{thesis} \) using excess-non-negative by blast
qed

lemma excess-s-non-pos: \( \text{excess } f_s \leq 0 \)
  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: \( \text{val} = (\sum e \in \text{outgoing } s. \ f_e) \)
  unfolding val-def by (auto simp: no-inflow-s)

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

context Network begin

abbreviation cf-of ≡ residualGraph c
abbreviation cfE-of f ≡ 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 ⊆ E ∪ E⁻¹
  unfolding residualGraph-def Graph.E-def
  by auto

lemma cfE-of-ss-VxV: cfE-of f ⊆ V × 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) ∉ cfE-of f
proof
  assume a1: (u, u) ∈ cfE-of f
  have (u, u) ∉ 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 $cf \equiv \text{residualGraph } c f$
sublocale $cf$:
Graph $cf$.
lemmas $cf$-def $\equiv \text{residualGraph-def }[\text{of } c f]$

3.2 Properties

lemmas $cfE$-ss-invE $\equiv cfE$-of-ss-invE $[\text{of } f]$

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

lemma $\text{resV-netV }[\text{simp}]: cf.V = V$
proof
show $V \subseteq \text{Graph.V } cf$
proof
fix $u$
assume $u \in V$
then obtain $v$ where $(u, v) \in E \lor (v, u) \in E$ unfolding $V$-def by auto

moreover {
assume $(u, v) \in E$
then have $(u, v) \in \text{Graph.E } cf \lor (v, u) \in \text{Graph.E } cf$
proof (cases)
assume $f(u, v) = 0$
then have $cf(u, v) = c(u, v)$
unfolding residualGraph-def using $(u, v) \in E$ by (auto simp:)
then have $cf(u, v) \neq 0$ using $(u, v) \in E$ unfolding $E$-def by auto
thus $\text{thesis unfolding Graph.E-def by auto}$
next
assume $f(u, v) \neq 0$
then have $cf(v, u) = f(u, v)$ unfolding residualGraph-def
using $(u, v) \in E$ no-parallel-edge by auto
then have $cf(v, u) \neq 0$ unfolding $f(u, v) \neq 0$ by auto
thus $\text{thesis unfolding Graph.E-def by auto}$
qed
}
moreover {
assume $(v, u) \in E$
then have $(v, u) \in \text{Graph.E } cf \lor (u, v) \in \text{Graph.E } cf$
proof (cases)
assume \( f(v, u) = 0 \)
then have \( cf(v, u) = c(v, u) \)
  unfolding residualGraph-def using \( \langle v, u \rangle \in E \) by (auto)
then have \( cf(v, u) \neq 0 \) using \( \langle v, u \rangle \in E \) unfolding E-def by auto
thus \( \text{thesis unfolding Graph.E-def by auto} \)
next
assume \( f(v, u) \neq 0 \)
then have \( cf(u, v) = f(v, u) \)
  unfolding residualGraph-def
  using \( \langle v, u \rangle \in E \) no-parallel-edge by auto
then have \( cf(u, v) \neq 0 \) using \( f(v, u) \neq 0 \) by auto
thus \( \text{thesis unfolding Graph.E-def by auto} \)
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 \( \text{resE-nonNegative:} \ cf \geq 0 \)
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 \( cf(u,v) = f(v,u) \)
    unfolding no-parallel-edge cf-def by auto
  hence \( cf(u,v) \geq 0 \)
    using capacity-const by auto
  \}
  moreover \{
  assume \( (u,v) \notin E \) \( (v,u) \notin E \)
  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** $\text{cf } e \geq 0$

- **Apply** \(\text{cases } e\)
- **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** \(\text{resE-positive: } e \in \text{cf } E \implies \text{cf } e > 0\)

**Proof**

- Assume \(e \in \text{cf } E\)
- Hence \(e \neq 0\) unfolding \(\text{cf } E\)-def by auto
- Thus ?thesis using \(\text{resE-nonNegative}\) by (meson eq-iff not-le)

**Qed**

**Lemma** reverse-flow: \(\text{Preflow cf } s \ t f' \implies \forall (u, v) \in E. \ f'(v, u) \leq f(u, v)\)

**Proof**

- Assume \(\text{asm: } \text{Preflow cf } s \ t f'\)
  - Then interpret \(f': \text{Preflow cf } s \ t f'\).

  \[
  \begin{align*}
  &\{ \\
  &\quad \text{Fix } u \ v \\
  &\quad \text{Assume } (u, v) \in E \\
  &\quad \text{Then have } \text{cf} (v, u) = f(u, v) \\
  &\quad \text{Unfolding residualGraph-def using no-parallel-edge by auto} \\
  &\quad \text{Moreover have } f'(v, u) \leq \text{cf} (v, u) \text{ using } f'.\text{capacity-const by auto} \\
  &\quad \text{Ultimately have } f'(v, u) \leq f(u, v) \text{ by meson} \\
  &\}\text{thus ?thesis by auto}
  \]

**Qed**

**Definition** (in Network) \(\text{flow-of-cf cf e } \equiv (\text{if } (e \in E) \text{ then } c \ e - \text{cf e else 0})\)

**Lemma** (in NPreflow) \(\text{E-ss-cfinvE: } E \subseteq \text{Graph}.E \cup (\text{Graph}.E \text{ cf})^{-1}\)

- **Unfolding** residualGraph-def \(\text{Graph}.E\)-def
- **Apply** (clarsimp)
- **Using** no-parallel-edge
- **Unfolding** \(\text{E-def}\)
- **Apply** (simp add:)
- **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: \(\text{excess } f u > 0 \implies \text{cf.outgoing } u \neq \{}\)
unfolding excess-def
proof –
  assume $0 < \sum f \text{ (incoming } u \text{)} - \sum f \text{ (outgoing } u \text{)}$
  hence $0 < \sum f \text{ (incoming } u \text{)}$
    by (metis diff-gt-0-iff-gt linorder-neqE-linordered-idom linorder-not-le
        sum-f-non-negative)
with f-non-negative obtain e where e\in incoming u f e > 0
  by (meson not-le sum-nonpos)
then obtain v where (v, u)\in E 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
= Network +
  fixes cf
  assumes EX-RPG: $\exists f. \text{NPreflow } c\ s\ t\ f \land cf = \text{residualGraph } c\ f$
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 = 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 (flow-of-cf cf)
    apply (fold f-def)
    using EX-RPG f-unique by metis
sublocale \( f: \) NPreflow \( c \ s \ t \ \text{unfolding} \ f\text{-def} \ \text{by} \ (\text{rule is-NPreflow})

lemma \( rg\text{-is-cf}[\text{simp}]: \) residualGraph \( c \ f = cf \)
  using EX-RPG \( f\text{-unique} \ \text{by} \ \text{auto} \)

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

sublocale \( cf: \) Graph \( cf \).

lemma \( \text{resV-netV}[\text{simp}]: \) \( cf.V = V \)
  using \( f.\text{resV-netV} \ \text{by} \ \text{simp} \)

sublocale \( cf: \) Finite-Graph \( cf \)
apply unfold-locales
apply simp
done

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

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

lemma \( \text{resE-nonNegative:} \) \( cf.e \geq 0 \)
  using \( f.\text{resE-nonNegative} \ \text{by} \ \text{auto} \)

end

context NPreflow begin

lemma \( \text{is-RPreGraph}: \) RPreGraph \( c \ s \ t \ \text{cf} \)
apply unfold-locales
apply (rule \( \text{exI[where} \ x=f]\))
apply (safe; unfold-locales)
done

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

end
lemma (in NPreflow) flow-of-cf (residualGraph c f) = f
  by (rule fa-rg-inv)

locale RGraph — Locale that characterizes a residual graph of a network
  = Network +
fixes cf
assumes EX-RG: ∃f. NFlow c s t f ∧ 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 ∃f. NPreflow c s t f ∧ 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 = (∑(u,v)∈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

custom 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 $f'$ ≡ λ(u, v).
  if $(u, v) \in E$ then
    $f (u, v) + f' (u, v) - f' (v, u)$
  else
We define a syntax similar to Cormen et al.:

abbreviation (input) augment-syntax (infix ↑ 55)
where \( f \uparrow f' \equiv NFlow.\text{augment} \ f \ f' \)

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 \( f' :: '\text{capacity flow} \)
- assumes \( f'-flow \colon \text{Flow} \ c \ s \ t \ f' \)

**begin**

**interpretation** \( f' \colon \text{Flow} \ c \ s \ t \ f' \) 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 \( 0 \leq (f \uparrow f')(u,v) \land (f \uparrow f')(u,v) \leq c(u,v) \)

**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 \( f(u,v) + f'(u,v) - f'(v,u) \leq f(u,v) \)
    - using \( f'(v,u) \leq f(u,v) \) by auto
  - also have \( f'(u,v) \) by auto
  - also have \( \geq 0 \) 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 \( f(u,v) + f'(u,v) \leq f(u,v) \) using \( f'.capacity-const \) by auto
    - also have \( f(u,v) + cf(u,v) \leq f(u,v) \) using \( f'.capacity-const \) by auto
    - also have \( f(u,v) + c(u,v) - f(u,v) \)
      - by (auto simp: residualGraph-def)
also have \ldots = 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 \(E''\{u\}\) characterizes the successor nodes of \(u\), and \(E^{-1}\{u\}\) characterizes the predecessor nodes of \(u\).

**private lemma split-rflow-outgoing:**

\[
\left(\sum_{v \in cf.E''\{u\}}. f'(u,v)\right) = \left(\sum_{v \in E''\{u\}}. f'(u,v)\right) + \left(\sum_{v \in E^{-1}\{u\}}. f'(v,u)\right)
\]

(is \(\text{LHS} = \text{RHS}\))

**proof** --

from no-parallel-edge have DJ: \(E''\{u\} \cap E^{-1}\{u\} = \{\}\) by auto

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

apply (rule sum.mono-neutral-left)

using cfE-ss-invE

by (auto intro: finite-Image)

also have \ldots = \(\text{RHS}\)

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

by (auto intro: finite-Image)

finally show \(\text{LHS} = \text{RHS}\).

qed

**private lemma split-rflow-incoming:**

\[
\left(\sum_{v \in cf.E^{-1}\{u\}}. f'(v,u)\right) = \left(\sum_{v \in E^{-1}\{u\}}. f'(v,u)\right) + \left(\sum_{v \in E''\{u\}}. f'(v,u)\right)
\]

(is \(\text{LHS} = \text{RHS}\))

**proof** --

from no-parallel-edge have DJ: \(E''\{u\} \cap E^{-1}\{u\} = \{\}\) by auto

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

apply (rule sum.mono-neutral-left)

using cfE-ss-invE

by (auto intro: finite-Image)

also have \ldots = \(\text{RHS}\)

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

by (auto intro: finite-Image)

finally show \(\text{LHS} = \text{RHS}\).

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 ∈ 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 (∑v ∈ E''{{u}}. f'(u,v)) − (∑v ∈ E''{{u}}. f'(v,u))
= (∑v ∈ E''{u}. f'(v,u)) − (∑v ∈ E''{u}. f'(u,v))

proof −

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

have ?A = (∑v ∈ cf.E''{{u}}. f'(u,v)) − (∑v ∈ E''{u}. f'(u,v))
by (simp add: split-rflow-outgoing)
also have (∑v ∈ cf.E''{u}. f'(u,v)) = (∑v ∈ cf.E''{u}. f'(v,u))
using U-ASM
by (simp add: f'.conservation-const-pointwise)
finally have ?A = (∑v ∈ cf.E''{u}. f'(v,u)) − (∑v ∈ E''{u}. f'(u,v))
by simp
moreover
have ?B = (∑v ∈ cf.E''{u}. f'(v,u)) − (∑v ∈ E''{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 (∑e ∈ outgoing u. augment f' e) = (∑e ∈ 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 ?Vj = E'′′{u}

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

have ?LHS = (∑v ∈ ?Vo. augment f' (u,v))
by (auto simp: sum-outgoing-pointwise)
also have \[ \sum_{v \in \mathcal{V}_o} f(u,v) + f'(u,v) - f'(v,u) \]
by (auto simp: augment-def)
also have \[ \sum_{v \in \mathcal{V}_o} f(u,v) \]
by (auto simp: sum-subtractf sum.distrib)
also have \[ \sum_{v \in \mathcal{V}_o} f'(u,v) \]
by (auto simp: conservation-const-pointwise[OF U-ASM] flow-summation-aux)
also have \[ \sum_{v \in \mathcal{V}_i} f'(v,u) \]
by (auto simp: augment-def)
also have \[ \sum_{v \in \mathcal{V}_i} f'(v,u) \]
by (auto simp: sum-subtractf sum.distrib)
also have \[ \sum_{v \in \mathcal{V}_i} augment f'(v,u) \]
by (auto simp: augment-def)
also have \[ \sum_{v \in \mathcal{V}_i} f'(v,u) \]
by (auto simp: sum-subtractf sum.distrib)
also have \[ ?RHS \]
by (auto simp: sum-incoming-pointwise)
finally show \[ ?LHS = ?RHS \].

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.

\textbf{corollary} augment-flow-presv: \textit{Flow c s t (f + f')} using augment-flow-presv-cap augment-flow-presv-con by (rule-tac intro-Flow) auto

\section{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.

\textbf{lemma} augment-flow-value: \textit{Flow.val c s (f + f') = val + Flow.val c s f'}
\textbf{proof} –
interpret f'': \textit{Flow c s 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.

\textbf{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 \( (u,v) \notin E \) (\( v,u \)\( E \) for \( u,v \)
proof –
  from that cfE-ss-invE have \( (u,v) \notin cf.E \) by auto
  thus \( f'(u,v) = 0 \) by auto
qed

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

have \( f''\cdot\text{val} = (\sum u \in V. \text{augment } f' (s, u) - \text{augment } f' (u, s)) \)
unfolding \( f''\cdot\text{val-def} \) by simp
also have \( = (\sum u \in V. f (s, u) - f (u, s) + (f' (s, u) - f' (u, s))) \)
  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 \( \ldots = \text{val + Flow.val cf s f' \ldots} \)
  unfolding \( \text{val-def } f'.\text{val-def} \) by simp
finally show \( f''\cdot\text{val} = \text{val + f'}\cdot\text{val} \).
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''\cdot: Flow c s t f↑f' using augment-flow-presv .
  have aux1: \( f'(u,v) = 0 \) if \( A: (u,v) \notin E \) (\( v,u \)\( E \) for \( u,v \)
proof –
  from \( A \) cfE-ss-invE have \( (u,v) \notin cf.E \) by auto
  thus \( f'(u,v) = 0 \) by auto
qed

show \( ?\text{thesis} \)
unfolding \( \text{val-def } f'.\text{val-def} \) f''.\text{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]
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.

5.1 Definitions

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

\[
\text{definition isAugmentingPath} :: \text{path} \Rightarrow \text{bool}
\]

\[
\text{where} \quad \text{isAugmentingPath} \; p \equiv \text{isSimplePath} \; s \; p \; t
\]

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

\[
\text{definition resCap} :: \text{path} \Rightarrow '\text{capacity}
\]

\[
\text{where} \quad \text{resCap} \; p \equiv \text{Min} \; \{ \text{cf} \; e \mid e \in \text{set} \; p \}
\]

\[
\text{lemma resCap-alt} : \text{resCap} \; p = \text{Min} \; (\text{cf} \; \text{set} \; p)
\]

— Useful characterization for finiteness arguments

\[
\text{unfolding resCap-def \; apply (rule arg-cong}\{\text{where} \; f = \text{Min}\}) \text{ by auto}
\]

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

\[
\text{definition augmentingFlow} :: \text{path} \Rightarrow '\text{capacity} \; \text{flow}
\]
where \( \text{augmentingFlow} \ p \equiv \lambda (u, v) \).
if \((u, v) \in (\text{set} \ p)\) 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** \( \text{resCap-gzero-aux} \): \(\text{cf}.\text{isPath} \ s \ p \ t \Rightarrow 0 < \text{resCap} \ p\)

**proof**

1. **assume** \(\text{PATH}: \text{cf}.\text{isPath} \ s \ p \ t\)
2. **hence** \(\text{set} \ p \neq \{}\) using \(s\neq t\) by (auto)
3. **moreover have** \(\forall e \in \text{set} \ p. \ cf \ e > 0\)
4. **using** \(\text{cf}.\text{isPath}-\text{edgeset}[OF \ \text{PATH}]\) resE-positive by (auto)
5. **ultimately show** ?thesis unfolding resCap-alt by (auto)

**qed**

**lemma** \( \text{resCap-gzero} \): \(\text{isAugmentingPath} \ p \Rightarrow 0 < \text{resCap} \ p\)

**using** \(\text{resCap-gzero-aux}[of \ p]\)

by (auto simp: \(\text{isAugmentingPath-def} \ cf.\text{isSimplePath-def}\))

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

**lemma** \( \text{sum-augmenting-alt} \):

**assumes** finite \( A \)

**shows** \((\sum e \in A. (\text{augmentingFlow} \ p) \ e) = \text{resCap} \ p * \text{of-nat} (\text{card} (A \cap \text{set} \ p))\)

**proof**

1. **have** \((\sum e \in A. (\text{augmentingFlow} \ p) \ e) = \text{sum} (\lambda -. \text{resCap} \ p) (A \cap \text{set} \ p)\)
2. **apply** (subst \(\text{sum-inter-restrict}\))
3. **apply** (auto simp: \(\text{augmentingFlow-def} \ \text{assms}\))
4. **done**
5. **thus** ?thesis by auto

**qed**

**lemma** \( \text{augFlow-resFlow} \): \(\text{isAugmentingPath} \ p \Rightarrow \text{Flow} \ cf \ s \ t \ (\text{augmentingFlow} \ p)\)

**proof** (rule \(\text{cf}.\text{intro-Flow}; \text{intro allI ballI}\))

1. **assume** \(\text{AUG}: \text{isAugmentingPath} \ p\)
2. **hence** \(\text{SPATH}: \text{cf}.\text{isSimplePath} \ s \ p \ t\) by (simp add: \(\text{isAugmentingPath-def}\))
3. **hence** \(\text{PATH}: \text{cf}.\text{isPath} \ s \ p \ t\) by (simp add: \(\text{cf}.\text{isSimplePath-def}\))

\[
\]
We first show the capacity constraint

\[ 0 \leq (\text{augmentingFlow } p) e \land (\text{augmentingFlow } p) e \leq \text{cf } e \]

**proof** cases

assume \( e \in \text{set } p \)

hence \( \text{resCap } p \leq \text{cf } e \)

unfolding \( \text{resCap-alt} \) by auto

moreover have \( (\text{augmentingFlow } p) e = \text{resCap } p \)

unfolding \( \text{augmentingFlow-def} \) using \( e \in \text{set } p \) by auto

moreover have \( 0 < \text{resCap } p \) using \( \text{resCap-gzero} \) by simp

ultimately show \( \text{thesis} \) by auto

next

assume \( e \not\in \text{set } p \)

hence \( (\text{augmentingFlow } p) e = 0 \)

unfolding \( \text{augmentingFlow-def} \) by auto

thus \( \text{thesis} \) using \( \text{resE-nonNegative} \) by auto

qed

}\)

Next, we show the conservation constraint

\[ \text{fix } v \]

\[ \text{assume } \text{asm-s: } v \in \text{Graph.} V \text{ cf } - \{s, t\} \]

have \( \text{card } (\text{Graph.incoming cf } v \cap \text{set } p) = \text{card } (\text{Graph.outgoing cf } v \cap \text{set } p) \)

**proof** (\text{cases})

assume \( v \in \text{set } (\text{cf.} \text{pathVertices-fwd s p}) \)

from \( \text{cf.split-path-at-vertex}[\text{OF this PATH}] \) obtain \( p1 \) \( p2 \) where

\( \text{P-FMT: } \text{p=p1@p2} \)

and \( 1: \text{cf.isPath s p1 v} \)

and \( 2: \text{cf.isPath v p2 t} \)

from \( 1 \) obtain \( p1' \) \( u1 \) where [simp]: \( p1'=p1@[u1,v] \)

using \( \text{asm-s} \) by \( \text{(cases p1 rule: rev-cases) (auto simp: split-path-simps)} \)

from \( 2 \) obtain \( p2' \) \( u2 \) where [simp]: \( p2=(v,u2)#p2' \)

using \( \text{asm-s} \) by \( \text{(cases p2) (auto)} \)

from

\( \text{cf.isSPath-sg-outgoing}[\text{OF SPATH}, \text{of v u2}] \)

\( \text{cf.isSPath-sg-incoming}[\text{OF SPATH}, \text{of u1 v}] \)

\( \text{cf.isPath-edgeset}[\text{OF PATH}] \)

have \( \text{cf.outgoing } v \cap \text{set } p = \{(v,u2)\} \)

\( \text{cf.incoming } v \cap \text{set } p = \{(u1,v)\} \)

by \( \text{(fastforce simp: P-FMT cf.outgoing-def cf.incoming-def)} \)

thus \( \text{thesis} \) by auto

next

assume \( v \in \text{set } (\text{cf.pathVertices-fed s p}) \)

then have \( \forall u. (u,v) \not\in \text{set } p \land (v,u) \not\in \text{set } p \)

by \( \text{(auto dest: cf.pathVertices-edge[OF PATH])} \)

hence \( \text{cf.incoming } v \cap \text{set } p = \{\} \)

\( \text{cf.outgoing } v \cap \text{set } p = \{\} \)

by \( \text{(auto simp: cf.incoming-def cf.outgoing-def)} \)

thus \( \text{thesis} \) by auto

26
\[ \text{qed} \]

\[ \text{thus} \ (\sum_{e \in \text{Graph}, \text{incoming} \ cf \ v. \ (\text{augmentingFlow} \ p) \ e}) = \ (\sum_{e \in \text{Graph}, \text{outgoing} \ cf \ v. \ (\text{augmentingFlow} \ p) \ e}) \]

\[ \text{by} \ \text{(auto simp: sum-augmenting-alt)} \]

\[ \text{qed} \]

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

\text{lemma} \ \text{augFlow-val}: 
\[ \text{isAugmentingPath} \ p \implies \text{Flow.val cf s (augmentingFlow} \ p) = \text{resCap} \ p \]

\text{proof} – 
\[ \text{assume AUG: isAugmentingPath} \ p \]
\[ \text{with} \ \text{augFlow-resFlow} \ \text{interpret} \ f: \text{Flow cf s t augmentingFlow} \ p. \]

\text{note} \ AUG 
\[ \text{hence} \ \text{SPATH}: \ cf.\text{isSimplePath} \ s \ p \ t \ \text{by} \ \text{(simp add: isAugmentingPath-def)} \]
\[ \text{hence} \ \text{PATH}: \ cf.\text{isPath} \ s \ p \ t \ \text{by} \ \text{(simp add: cf.isSimplePath-def)} \]
\[ \text{then obtain} \ v \ p' \ \text{where} \ p=(s,v)\#p' \ (s,v)\in cf.E \]
\[ \text{using} \ s\text{-not-t by (cases p) auto} \]
\[ \text{hence} \ cf.\text{outgoing} \ s \cap \text{set} \ p = \{(s,v)\} \]
\[ \text{using} \ cf.\text{isPath-sq-outgoing}[\text{OF} \ \text{SPATH}, \ \text{of} \ s \ v] \]
\[ \text{using} \ cf.\text{isPath-edgeset}[\text{OF} \ \text{PATH}] \]
\[ \text{by (fastforce simp: cf.outgoing-def)} \]
\[ \text{moreover have} \ cf.\text{incoming} \ s \cap \text{set} \ p = \{\} \ \text{using} \ \text{SPATH no-incoming-s} \]
\[ \text{by (auto simp: cf.incoming-def ;p=(s,v)\#p', in-set-conv-decomp[where xs=p'] simp: cf.isSimplePath-append cf.isSimplePath-cons]} \]
\[ \text{ultimately show} \ \text{?thesis} \]
\[ \text{unfolding} \ f.\text{val-def} \]
\[ \text{by (auto simp: sum-augmenting-alt)} \]

\[ \text{qed} \]

end — Network with flow
end — Theory

6 The Ford-Fulkerson Theorem

\text{theory} \ \text{Ford-Fulkerson}
\text{imports} \ \text{Augmenting-Flow Augmenting-Path}
\text{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 ≡ (∑ e ∈ outgoing k. f e) − (∑ e ∈ 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 ∈ k ∧ v ∈ k ∧ (u, v) ∈ E}
let ?AOG = {(u, v). u ∈ k ∧ (u, v) ∈ E}
let ?AIN = {(v, u). u ∈ k ∧ (v, u) ∈ E}
let ?SOG = λ u. (∑ e ∈ outgoing u. f e)
let ?SIN = λ u. (∑ e ∈ incoming u. f e)
let ?SOG' = (∑ e ∈ outgoing k. f e)
let ?SIN' = (∑ e ∈ incoming k. f e)

Some setup to make finiteness reasoning implicit

note [[simproc finite-Collect]]

have netFlow = ?SOG' + (∑ e ∈ ?LCL. f e) − (?SIN' + (∑ e ∈ ?LCL. f e))
(is - = ?SAOG - ?SAIN)
using netFlow-def by auto
also have ?SAOG = (∑ y ∈ k − {s}. ?SOG y) + ?SOG s
proof
have ?SAOG = (∑ e∈(outgoing' k ∪ ?LCL). f e)
  by (rule sum.union-disjoint[symmetric]) (auto simp: outgoing'-def)
also have outgoing' k ∪ ?LCL = (∪ y∈k−{s}. outgoing y) ∪ outgoing s
  by (auto simp: outgoing-def outgoing'-def s-in-cut)
also have (∑ e∈(∪ (outgoing' (k − {s}))) ∪ outgoing s). f e)
  = (∑ e∈(∪ (outgoing' (k − {s}))). f e) + (∑ e∈outgoing s. f e)
  by (rule sum.union-disjoint)
(ast simp: outgoing-def intro: finite-Image)
also have (∑ e∈(∪ (outgoing' (k − {s}))). f e)
  = (∑ y ∈ k − {s}. ?SOG y)
  by (rule sum.UNION-disjoint)
  (auto simp: outgoing-def intro: finite-Image)

28
finally show \( ?\text{thesis} \).

qed

also have \( ?SAIN = (\sum y \in k - \{s\}. \ ?\text{SIN} y) + \ ?\text{SIN} s \)

proof –

have \( ?SAIN = (\sum e \in (\text{incoming}' k \cup \ ?\text{LCL}). \ f e) \)

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

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

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

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

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

by (rule sum.union-disjoint)

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

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

\( = (\sum y \in k - \{s\}. \ ?\text{SIN} y) \)

by (rule sum.UNION-disjoint)

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

finally show \( ?\text{thesis} \).

qed

finally have \( \text{netFlow} = ((\sum y \in k - \{s\}. \ ?\text{SOG} y) + \ ?\text{SOG} s) \)

\( - (\sum y \in k - \{s\}. \ ?\text{SIN} y) + \ ?\text{SIN} s \)

(is \( \text{netFlow} = \ ?R \)).

also have \( ?R = \ ?\text{SOG} s - \ ?\text{SIN} s \)

proof –

have \( (\forall u. \ u \in k - \{s\} \implies \ ?\text{SOG} u = \ ?\text{SIN} u) \)

using conservation-const cut-ss-V t-ni-cut by force

thus \( ?\text{thesis} \) by auto

qed

finally show \( ?\text{thesis unfolding val-def by simp} \)

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: \( \text{val} \leq \text{cap} \)

proof –

have \( (\sum e \in \text{outgoing}' k. \ f e) \leq (\sum e \in \text{outgoing}' k. \ c e) \) (is \( \text{cap} \leq \ ?R \))

using capacity-const by (metis sum-mono)

then have \( (\sum e \in \text{outgoing}' k. \ f e) \leq \text{cap unfolding cap-def by simp} \)

moreover have \( \text{val} \leq (\sum e \in \text{outgoing}' k. \ f e) \) using netFlow-def

by (simp add: capacity-const flow-value sum-nonneg)

ultimately show \( ?\text{thesis by simp} \)

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 $f \implies \neg (\exists \ p. \ isAugmentingPath p)$

unfolding isMaxFlow-alt

proof (rule ccontr)

assume asm: NFlow $c \ s \ t \ f$

\wedge (\forall f'. \ NFlow c s t f' \implies Flow.val c s f' \leq Flow.val c s f)

assume asm-c: \neg (\exists \ p. \ isAugmentingPath p)

then obtain $p$ where obt: isAugmentingPath $p$ by blast

have fct1: Flow $c \ s \ t$ (augmentingFlow $p$) using obt augFlow-resFlow by auto

have fct2: Flow.val $c \ s$ (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 $?S = cfreachableNodes s$

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

hence $t \notin ?S$

unfolding isAugmentingPath-def cfreachableNodes-def cf.connected-def

by (auto dest: cf.isPath-pathLE)

then show CUT: NCut $c \ s \ t \ ?S$

proof unfold-locales

show GraphreachableNodes $cf \ s \subseteq V$

using cfreachable-ss-V s-node resV-netV by auto

show $s \in GraphreachableNodes cf \ s$

unfolding GraphreachableNodes-def Graph.connected-def

by (metis Graph.isPath.simps(1) mem-Collect-eq)

qed

then interpret NCut $c \ s \ t \ ?S$.

interpret NFlowCut $c \ s \ t \ ?S$ 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 $(u,v) \notin E \ u \notin ?S \ v \notin ?S$

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 $(u,v) \in E$ by auto

hence $(u, v) \in cf.E$ unfolding cf-def by simp

hence $v \in ?S$ using $(u \in ?S)$ by (auto intro: cf.reachableNodes-append-edge)

thus False using $(v \notin ?S)$ 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 $(u,v) \in E$ $(v,u) \notin E$ by auto

hence $u \in ?S$ using $(v \in ?S)$ cf.reachableNodes-append-edge by auto

thus False using $(u \notin ?S)$ 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:

$\exists k. \ NCut c s t k \land val = NCut.cap c k \Rightarrow isMaxFlow f$

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 $(val = cap)[symmetric]$
  finally have $fc'.val \leq val$.
\}

thus $isMaxFlow f$ unfolding $isMaxFlow-def$
Finally we can state the Ford-Fulkerson theorem:

**Theorem ford-fulkerson**: shows

\[
\text{isMaxFlow } f \iff \\
\neg \exists x \text{ isAugmentingPath} \text{ and } \neg \exists x \text{ isAugmentingPath} \iff \\
(\exists k. \text{NCut } c \, s \, t \, k \land \text{val } = \text{NCut.cap } c \, k)
\]

**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**:  
\[
(\sum e \in \text{incoming } t . \, f \, e) = (\sum e \in \text{outgoing } s . \, f \, e)
\]

**Proof** –

We choose a cut between the sink and all other nodes

\[
\text{let } ?K = V - \{t\}
\]

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' \(\, ?K = \text{incoming } t\)

**And** incoming' \(\, ?K = \{\}\)

**Using** no-self-loop no-outgoing-t

**Unfolding** outgoing'-def incoming-def incoming'-def outgoing-def V-def

**By** auto

hence \(\sum e \in \text{incoming } t . \, f \, e) = \text{netFlow}\)** unfolding netFlow-def by auto

**Also have** netFlow = val **by (rule flow-value)**

**Also have** val = \(\sum e \in \text{outgoing } s . \, f \, e) **by (auto simp: val-alt)**

**Finally show** \(?\text{thesis}\)

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**: \((\exists p. \text{isAugmentingPath } p) \iff \text{isMaxFlow } f\)

**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**: \( [\text{isMaxFlow } f; \text{isMinCut } c s t k] \Rightarrow \text{Flow.val } c s f = \text{NCut.cap } c k \)

**Proof**

- Assume \( \text{isMaxFlow } f \) and \( \text{isMinCut } c s t k \)
- Then interpret \( \text{Flow } c s t f + \text{NCut } c s t k \)
  - Unfolding \( \text{isMaxFlow-def} \) \( \text{isMinCut-def} \) by \( \text{simp-all} \)
  - Interpret \( \text{NFlowCut } c s t f k \) by \( \text{intro-locales} \)

From \( \text{ford-fulkerson } (\text{isMaxFlow } f) \)

Obtain \( k' \) where \( \text{NCut } c s t k' \) and \( \text{val } = \text{NCut.cap } c k' \)

By \( \text{blast} \)

Thus \( \text{val } = \text{cap} \)

Using \( (\text{isMinCut } c s t k) \) \( \text{weak-duality} \)

Unfolding \( \text{isMinCut-def} \) by \( \text{auto} \)

Qed

end — Theory

**References**


