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

Peter Lammich and S. Reza Seifidgar

September 13, 2023

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 ⇒ 'capacity

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

assumes capacity-const: ∀ e. 0 ≤ f e ∧ f e ≤ c e
assumes no-deficient-nodes: ∀ v ∈ V − {s,t}.
(∑ e∈outgoing v. f e) ≤ (∑ e∈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.

```plaintext
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.

```plaintext
lemma conservation-const:
  \forall v \in V - \{s, t\}. (∑ e ∈ outgoing v. f e) = (∑ e ∈ 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.

```plaintext
definition val :: 'capacity
  where val ≡ (∑ e ∈ outgoing s. f e) - (∑ e ∈ incoming s. f e)
end
```

```plaintext
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.

```plaintext
type-synonym cut = node set

locale Cut = Graph +
  fixes k :: cut
  assumes cut-ss-V: k ⊆ 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.

\[
\text{locale Network = Graph c | for } c :: 'capacity::linordered-idom graph +
\]
\[
\text{fixes } s \ t :: \text{node}
\]
\[
\text{assumes } s\text{-node[simp, intro!]}: s \in V
\]
\[
\text{assumes } t\text{-node[simp, intro!]}: t \in V
\]
\[
\text{assumes } s\text{-not-t[simp, intro!]}: s \neq t
\]
\[
\text{assumes } \text{cap-non-negative}: \forall u \ v. \ c(u, v) \geq 0
\]
\[
\text{assumes } \text{no-incoming-s}: \forall u. \ (u, s) \notin E
\]
\[
\text{assumes } \text{no-outgoing-t}: \forall u. \ (t, u) \notin E
\]
\[
\text{assumes } \text{no-parallel-edge}: \forall u \ v. \ (u, v) \in E \rightarrow (v, u) \notin E
\]
\[
\text{assumes } \text{nodes-on-st-path}: \forall v \in V. \ \text{connected s v} \land \text{connected v t}
\]
\[
\text{assumes } \text{finite-reachable}: \text{finite} \ (\text{reachableNodes } s)
\]
\[
\text{begin}
\]

Edges have positive capacity

\[
\text{lemma edge-cap-positive: } (u,v) \in E \implies c(u,v) > 0
\]
\[
\text{unfolding E-def using cap-non-negative}\[THEN\ spec2, \ of \ u \ v\] \text{by simp}
\]

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

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

Thus, the network is actually a finite graph.

\[
\text{sublocale Finite-Graph}
\]
\[
\text{apply unfold-locales}
\]
\[
\text{using reachable-is-V finite-reachable by auto}
\]
Our assumptions imply that there are no self loops

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

using no-parallel-edge by auto

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

unfolding adjacent-nodes-def using no-self-loop by auto

A flow is maximal, if it has a maximal value

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

where \( \text{isMaxFlow } f \equiv \text{Flow } c s t f \land (\forall f'. \text{Flow } c s t f' \rightarrow \text{Flow} \cdot \text{val } c s f' \leq \text{Flow} \cdot \text{val } c s f) \)

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

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

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

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

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

\section{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.

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

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

begin

\end

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

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

\[ \text{lemma (in Network) isMaxFlow-alt:} \]

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

\[ \text{unfolding isMaxFlow-def by (auto simp: NFlow-def Flow-def NPreflow-def) intro-locales} \]

A cut in a network separates the source from the sink.

\[ \text{locale } \text{NCut } = \text{Network } c s t + \text{Cut } c k \]

for \( c :: '\text{capacity}::\text{linordered-idom graph and } s t k + \)
\textbf{assumes} \ s-in-cut: \ s \in k \\
\textbf{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.

\textbf{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.

\textbf{definition} \ \text{isMinCut} :: - graph \Rightarrow nat \Rightarrow nat \Rightarrow cut \Rightarrow bool \\
where \ \text{isMinCut} c s t k \equiv \text{NCut} c s t k \land \forall k'. \text{NCut} c s t k' \rightarrow \text{NCut}.cap \ c \ k \leq \text{NCut}.cap \ c \ k' 

\subsection*{2.2 Properties}

\subsubsection*{2.2.1 Flows}

\textbf{context} \ \text{Preflow} \\
\begin{itemize}
\item Only edges are labeled with non-zero flows
\item \textbf{lemma} \ zero-flow-simp[simp]: \\
\quad \forall \ (u,v) \notin E \Rightarrow f(u,v) = 0 \\
\quad \text{by (metis capacity-const eq-iff zero-cap-simp)} \\
\item \textbf{lemma} \ f-non-negative: \ 0 \leq f \ e \\
\quad \text{using} \ \text{capacity-const} \ \text{by (cases e) auto} \\
\item \textbf{lemma} \ \text{sum-f-non-negative: sum f X} \geq 0 \ \text{using} \ \text{capacity-const} \\
\quad \text{by (auto simp: sum-nonneg f-non-negative)}
\end{itemize}

\textbf{end} \ — \ \text{Preflow} \\

\textbf{context} \ \text{Flow} \\
\begin{itemize}
\item We provide a useful equivalent formulation of the conservation constraint.
\item \textbf{lemma} \ \text{conservation-const-pointwise:} \\
\quad \textbf{assumes} \ w \in V - \{s,t\} \\
\quad \textbf{shows} \ \sum v \in E^{-1}\{u\}. \ f \ (u,v) = (\sum v \in E^{-1}\{u\}. \ f \ (v,u)) \\
\quad \textbf{using} \ \text{conservation-const} \ \text{assms} \\
\quad \text{by (auto simp: sum-incoming-pointwise sum-outgoing-pointwise)}
\end{itemize}

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

\textbf{lemma} \ \text{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 \( -(\sum_{e \in \text{incoming } s} c_e) \leq \text{val} \leq (\sum_{e \in \text{outgoing } s} c_e) \)
  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 \( \text{cap} : \forall e. \ 0 \leq f_e \land f_e \leq c_e \)
- assumes \( \text{cons} : \forall v \in V - \{s, t\}. \)
  \( (\sum e \in \text{incoming } v \ f_e) = (\sum e \in \text{outgoing } v \ f_e) \)
- shows \( \text{Flow } c \ s \ t \ f \)
  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 \( g :: \text{edge } \Rightarrow '\text{capacity} \)
- 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 \( g :: \text{edge } \Rightarrow '\text{capacity} \)
- 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 ∈ E ⇒ c e > 0
unfolding E-def using cap-non-negative le-neq-trans by fastforce

lemma V-not-empty: V ≠ {} using s-node by auto
lemma E-not-empty: E ≠ {} using V-not-empty by (auto simp: V-def)

lemma card-V-ge2: card V ≥ 2
proof
  have 2 = card {s, t} by auto
  also have {s, t} ⊆ V by auto
  hence card {s, t} ≤ card V by (rule-tac card-mono) auto
  finally show ?thesis .
qed

lemma zero-is-flow: Flow c s t (λ-. 0)
using cap-non-negative by unfold-locales auto

lemma max-flow-val-unique:
  [is-max-flow-val fv1; is-max-flow-val fv2] ⇒ 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: ∀ e ∈ incoming s. f e = 0 (is ?thesis)
proof (rule ccontr)
  assume ¬(∀ e ∈ 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 (simp: capacity-const sum-nonneg)
  thus ?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

class $NFlow$
begin
  locale 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: $val = (\sum_{e \in \text{outgoing } s} f e)$
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:

\[
\text{definition residualGraph :: - graph } \Rightarrow \text{ - flow } \Rightarrow \text{ - graph}
\]

where \( \text{residualGraph c f} \equiv \lambda (u, v). \)

\[
\begin{align*}
\text{if } (u, v) & \in \text{Graph.E c then} \\
& c (u, v) - f (u, v) \\
\text{else if } (v, u) & \in \text{Graph.E c then} \\
& f (v, u) \\
\text{else} & 0
\end{align*}
\]

context Network begin

abbreviation \( \text{cf-of} \equiv \text{residualGraph c} \)

abbreviation \( \text{cfE-of} f \equiv \text{Graph.E (cf-of f)} \)

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

lemma \( \text{cfE-of-ss-invE}: \text{cfE-of c} \subseteq E \cup E^{-1} \)

unfolding residualGraph-def Graph.E-def

by auto

lemma \( \text{cfE-of-ss-VxV}: \text{cfE-of f} \subseteq V \times V \)

unfolding V-def

unfolding residualGraph-def Graph.E-def

by auto

lemma \( \text{cfE-of-finite[simp, intro!]}: \text{finite (cfE-of f)} \)

using finite-subset[OF cfE-of-ss-VxV] by auto

lemma \( \text{cf-no-self-loop}: (u, u) \notin \text{cfE-of f} \)

proof

assume \( a1: (u, u) \in \text{cfE-of f} \)

have \( (u, u) \notin E \)

end — Theory
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 \( cf\text{-def} = \text{residualGraph-def}[of } c f \)

3.2 Properties

lemmas \( cf\text{-E-ss-invE} = cf\text{-E-of-ss-invE}[of } f \)

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 \( V\text{-def} \) 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 \( E\text{-def} \) 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 \( cf(v, u) \neq 0 \)
using \((u, v) \neq 0\) 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

} 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: \( cf(e) \geq 0 \)
proof (cases e; simp)
  fix \( u, v \)
  
  \{ assume \( \langle u, v \rangle \in E \)
      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 \( \langle v, u \rangle \in E \)
      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 \( \langle u, v \rangle \notin E \quad \langle v, u \rangle \notin E \)
      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** $cf \ e \geq 0$

apply (cases $e$)

unfolding residualGraph-def

using no-parallel-edge capacity-const cap-positive

by auto

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

**corollary** $resE$-positive: $e \in cf.E \implies cf \ e > 0$

**proof** –

assume $e \in cf.E$

hence $cf \ e \neq 0$ unfolding $cf.E$-def by auto

thus ?thesis using resE-nonNegative by (meson eq-iff not-le)

qed

**lemma** reverse-flow: $Preflow \ cf \ s \ t \ f' \implies \forall (u, v) \in E. f'(v, u) \leq f(u, v)$

**proof** –

assume $asm: Preflow \ cf \ s \ t \ f'$

then interpret $f'$: $Preflow \ cf \ s \ t \ f'$.

{}

fix $u \ v$

assume $(u, v) \in E$

then have $cf \ (v, u) = f(u, v)$

unfolding residualGraph-def using no-parallel-edge by auto

moreover have $f'(v, u) \leq cf \ (v, u)$ using $f'.capacity-const$ by auto

ultimately have $f'(v, u) \leq f(u, v)$ by metis

} thus ?thesis by auto

qed

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

**lemma** (in NPreflow) $E$-ss-cfineE: $E \subseteq Graph.E \cup (Graph.E \ cf)^{-1}$

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 $f \ u > 0 \implies cf.outgoing \ u \neq \{\}$
unfolding excess-def

proof –
assume $0 < \sum f \text{ (incoming } u) - \sum f \text{ (outgoing } u)$
hence $0 < \sum f \text{ (incoming } u)$
  by (metis diff_gt_0_iff_gt linorder_neqE linordered_idom linorder_not_le
  sum-f_non-negative)
with $f$-non-negative obtain $e$ where $e \in$ incoming $u \quad f \ e > 0$
  by (meson not_le sum_nonpos)
then obtain $v$ 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
= 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 $\text{NPreflow } c \ s \ t \ f'$
  assumes $A: cf = \text{residualGraph } c \ f'$
  shows $f' = f$
proof –
interpret $f': \text{NPreflow } c \ s \ t \ f'$ by fact

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

lemma is-NPreflow: $\text{NPreflow } c \ s \ t \ (\text{flow-of-cf } cf)$
  apply (fold $f\text{-def}$)
  using $\text{EX-RPG } f$-unique by metis
sublocale f: NPreflow c s t 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 ⊆ cf.E ∪ cf.E⁻¹
  using f.E-ss-cfinvE by simp

lemma cfE-ss-invE: cf.E ⊆ E ∪ E⁻¹
  using f.cfE-ss-invE by simp

lemma resE-nonNegative: cf e ≥ 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 simp 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: ∃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

class 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 $(u, v) ∈ E$ then $f \cdot (u, v) + f' \cdot (u, v) − f' \cdot (v, u)$
  else

We define a syntax similar to Cormen et al.:

```
abbreviation (input) augment-syntax (infix ↑ 55)
  where f↑f′ :: NFlow.augment c f f′
```

such that we can write $f ↑ f′$ for the flow $f$ augmented by $f′$.

### 4.2 Augmentation yields Valid Flow

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

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

**begin**

**interpretation** $f′$: Flow cf s t f′ by (rule $f′$-flow)

#### 4.2.1 Capacity Constraint

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

**lemma** augment-flow-presv-cap:
- shows $0 ≤ (f ↑ f′)(u,v) ∧ (f ↑ f′)(u,v) ≤ c(u,v)
- proof (cases (u,v)∈E; rule conjI)
  - assume [simp]: (u,v)∈E
  - hence $f(u,v) = cf(v,u)$
    - using no-parallel-edge by (auto simp: residualGraph-def)
  - also have $cf(v,u) ≥ f′(v,u)$ using $f′$.capacity-const by auto
  - finally have $f′(v,u) ≤ f(u,v)$.

- have $(f ↑ f′)(u,v) = f(u,v) + f′(u,v) − f′(v,u)$
  - by (auto simp: augment-def)
- also have $f(u,v) ≥ f′(v,u)$ using $f′$.capacity-const by auto
- also have $f′(v,u) ≤ f(u,v)$ by auto
- also have $f′(u,v) ≥ 0$ using $f′$.capacity-const by auto
- finally show $(f ↑ f′)(u,v) ≥ 0$.

- have $(f ↑ f′)(u,v) = f(u,v) + f′(u,v) − f′(v,u)$
  - by (auto simp: augment-def)
- also have $f(u,v) + f′(u,v) ≥ f′.capacity-const$ by auto
- also have $f′(v,u) + cf(u,v) ≥ f′.capacity-const$ by auto
- also have $f′(u,v) + c(a,v) − f(u,v)$
  - by (auto simp: residualGraph-def)
also have \( \ldots = c(u,v) \) by \textit{auto}

finally show \((f^\dagger f')(u, v) \leq c(u, v)\).
\texttt{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 \( E^\dagger\{u\} \) characterizes the successor nodes of \( u \), and \( E^{-1}\dagger\{u\} \) characterizes the predecessor nodes of \( u \).

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

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

(is \(?LHS = \?RHS\))

**proof** –

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

have \(?LHS = (\sum v \in E^{-1}\dagger\{u\} \cup E^{-1}\dagger\{u\}). f'(u,v))\)

apply (rule sum.mono-neutral-left)

using \( \text{cfE-ss-invE} \)

by (auto intro: finite-Image)

also have \( \ldots = \?RHS \)

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

by (auto intro: finite-Image)

finally show \(?LHS = \?RHS\) .
\texttt{qed}

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

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

(is \(?LHS = \?RHS\))

**proof** –

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

have \(?LHS = (\sum v \in E^{-1}\dagger\{u\} \cup E^{-1}\dagger\{u\}). f'(v,u))\)

apply (rule sum.mono-neutral-left)

using \( \text{cfE-ss-invE} \)

by (auto intro: finite-Image)

also have \( \ldots = \?RHS \)

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

by (auto intro: finite-Image)

finally show \(?LHS = \?RHS\) .
\texttt{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-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 $(\sum_{v \in E'' \setminus \{u\}}. f'(u,v)) - (\sum_{v \in E' \setminus \{u\}}. f'(v,u)) = (\sum_{v \in E' \setminus \{u\}}. f'(v,u)) - (\sum_{v \in E'' \setminus \{u\}}. f'(u,v))$

proof –

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

have ?A = $(\sum_{v \in cf. E'' \setminus \{u\}}. f'(u,v)) - (\sum_{v \in E' \setminus \{u\}}. f'(v,u))$  by (simp add: split-rflow-outgoing)
also have $(\sum_{v \in cf. E'' \setminus \{u\}}. f'(u,v)) = (\sum_{v \in cf. E' \setminus \{u\}}. f'(v,u))$
using U-ASM
by (simp add: f'.conservation-const-pointwise)
finally have ?A = $(\sum_{v \in cf. E' \setminus \{u\}}. f'(v,u)) - (\sum_{v \in E'' \setminus \{u\}}. f'(u,v))$
by simp
moreover
have ?B = $(\sum_{v \in cf. E' \setminus \{u\}}. f'(v,u)) - (\sum_{v \in E'' \setminus \{u\}}. f'(v,u))$
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 $?Vo = E'' \{u\} \text{ let } ?Vi = E' \{u\}$

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 ... 
  \[ \sum_{v \in ?V_0} f(u,v) + f'(u,v) - f'(v,u) \]
  by (auto simp: augment-def)

also have ... 
  \[ \sum_{v \in ?V_0} f(u,v) + \sum_{v \in ?V_0} f'(u,v) - \sum_{v \in ?V_0} f'(v,u) \]
  by (auto simp: sum-subtractf sum.distrib)

also have ... 
  \[ \sum_{v \in ?V_i} f(v,u) + \sum_{v \in ?V_i} f'(v,u) - \sum_{v \in ?V_i} f'(u,v) \]
  by (auto simp: conservation-const-pointwise[OF U-ASM] flow-summation-aux)

also have ... 
  \[ \sum_{v \in ?V_i} f(v,u) + f'(v,u) - f'(u,v) \]
  by (auto simp: sum-subtractf sum.distrib)

also have ... 
  \[ \sum_{v \in ?V_i} f'(v,u) \]
  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↑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↑f') = val + Flow.val cf s f'

**proof** –
  interpret f": Flow c s t f↑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] =
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.

\[
\text{have } \text{aux1}: f'(u,v) = 0 \text{ if } (u,v) \notin E \text{ and } (v,u) \notin E \text{ for } u v
\]

\[
\text{proof - from that cfE-ss-invE have } (u,v) \notin E \text{ by auto}
\]

\[
\text{thus } f'(u,v) = 0 \text{ by auto}
\]

\[
\text{qed}
\]

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

\[
\text{have } f''.\text{val} = (\sum u \in V. \text{augment } f' (s, u) - \text{augment } f' (u, s))
\]

\[
\text{unfolding } f''.\text{val-def by simp}
\]

\[
\text{also have } \ldots = (\sum u \in V. f (s, u) - f (u, s) + (f' (s, u) - f' (u, s)))
\]

\[
\text{— Note that this is the crucial step of the proof, which Cormen et al. leave as an exercise.}
\]

\[
\text{by (rule sum.cong) (auto simp: augment-def no-parallel-edge aux1)}
\]

\[
\text{also have } \ldots = \text{val + Flow.val cf s f'}
\]

\[
\text{unfolding } \text{val-def f'.val-def by simp}
\]

\[
\text{finally show } f''.\text{val} = \text{val + f'.val}.
\]

\[
\text{qed}
\]

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.

\[
\text{lemma } \text{Flow.val c s (f'f')} = \text{val + Flow.val cf s f'}
\]

\[
\text{proof - interpret } f'': \text{Flow c s t f'f' using augment-flow-presv}.
\]

\[
\text{have aux1: } f'(u,v) = 0 \text{ if } A: (u,v) \notin E \text{ and } (v,u) \notin E \text{ for } u v
\]

\[
\text{proof - from } A \text{ cfE-ss-invE have } (u,v) \notin E \text{ by auto}
\]

\[
\text{thus } f'(u,v) = 0 \text{ by auto}
\]

\[
\text{qed}
\]

\[
\text{show } \text{thesis}
\]

\[
\text{unfolding } \text{val-def f'.val-def f''.val-def}
\]

\[
\]

\[
23
\]
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:

**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 \( p \equiv Min \ (cf . set \ p) \)

— Useful characterization for finiteness arguments

**unfolding** resCap-def **apply** (rule arg-cong[where \( f \equiv Min \)]) 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 \( \text{augmentingFlow} \; p \equiv \lambda(u, v). \)
if \((u, v) \in (\text{set}\; p)\) then
\(\text{resCap} \; p\)
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**
- assume \(\text{PATH}::\; \text{cf}.\; \text{isPath} \; s \; p \; t\)
  hence \(\text{set} \; p \neq \{\}\) using \(\text{s-not-t}\) by (auto)
- moreover have \(\forall e \in \text{set} \; p.\; \text{cf} \; e > 0\)
  using \(\text{cf}.\; \text{isPath-edgeset}[[\text{OF} \; \text{PATH}]]\) \(\text{resE-positive}\) by (auto)
- ultimately show \(\text{thesis}\) unfolding \(\text{resCap-alt}\) by (auto)

**qed**

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

**proof**
- using \(\text{resCap-gzero-aux}[\text{of} \; p]\)
  by (auto simp: \(\text{isAugmentingPath-def}\; \text{cf}.\; \text{isSimplePath-def}\))

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

**lemma** \(\text{sum-augmenting-alt}::\; \text{assumes finite} \; A\)

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

**proof**
- have \(\sum\; e \in\; A.\; (\text{augmentingFlow} \; p) \; e = \text{sum} \; (\lambda - \; \text{resCap} \; p) \; (A \cap \text{set} \; p)\)
  apply (subst \text{sam.inter-restrict})
  apply (auto simp: \text{augmentingFlow-def} \text{assms})
  done
  thus \(\text{thesis}\) 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}; intro \text{allI} \; \text{ballI})
- assume \(\text{AUG}::\; \text{isAugmentingPath} \; p\)
  hence \(\text{SPATH}::\; \text{cf}.\; \text{isSimplePath} \; s \; p \; t\) by (simp add: \text{isAugmentingPath-def})
  hence \(\text{PATH}::\; \text{cf}.\; \text{isPath} \; s \; p \; t\) by (simp add: \text{cf}.\; \text{isSimplePath-def})

\{
We first show the capacity constraint

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

**proof** cases

- assume \( e \in \text{set } p \)
  - hence \( \text{resCap } p \leq cf e \)
    - 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 \( \text{resCap-gzero}[OF AUG] \) by simp
  - ultimately show \( \text{thesis} \) by auto

**next**

- assume \( e \notin \text{set } p \)
  - 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

**fix** \( v \)

**assume** \( \text{asm-s: } v \in \text{Graph}.V cf \ - \ \{s, t\} \)

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

**proof** (cases)

- assume \( v \in \text{set } (\text{cf}.\text{pathVertices-fwd } s p) \)
  - from \( \text{cf}.\text{split-path-at-vertex}[OF this PATH] \) obtain \( p1 p2 \) where
    \( \text{P-FMT}: p=p1 \oplus p2 \)
    - 1: \( \text{cf}.\text{isPath } s p1 v \)
    - 2: \( \text{cf}.\text{isPath } v p2 t \)
  - from 1 obtain \( p1' u1 \) where
    \( \text{simp}: p1'=p1' \oplus [(u1,v)] \)
    - using \( \text{asm-s} \) by (auto dest: \( \text{cf}.\text{pathVertices-edge} [OF PATH] \))
  - from 2 obtain \( p2' u2 \) where
    \( \text{simp}: p2'=(v,u2)#p2' \)
    - using \( \text{asm-s} \) by (cases p2) (auto)
  - from
    - \( \text{cf}.\text{isSPath-sg-outgoing}[OF SPATH, of } v u2 \)
    - \( \text{cf}.\text{isSPath-sg-incoming}[OF SPATH, of } u1 v \)
    - \( \text{cf}.\text{Path-edgeset}[OF PATH] \)
    - have \( \text{cf}.\text{outgoing } v \cap \text{set } p = \{(v,u2)\} \)
    - \( \text{cf}.\text{incoming } v \cap \text{set } p = \{(u1,v)\} \)
    - by (fastforce simp: \( \text{P-FMT} \) \( \text{cf}.\text{outgoing-def} \) \( \text{cf}.\text{incoming-def} \))
    - thus \( \text{thesis} \) by auto
  - **next**
    - assume \( v \notin \text{set } (\text{cf}.\text{pathVertices-fwd } s p) \)
    - then have \( \forall u. (u,v) \notin \text{set } p \land (v,u) \notin \text{set } p \)
      - by (auto dest: \( \text{cf}.\text{pathVertices-edge}[OF PATH] \))
    - hence \( \text{cf}.\text{incoming } v \cap \text{set } p = \{} \)
    - \( \text{cf}.\text{outgoing } v \cap \text{set } p = \{} \)
      - by (auto simp: \( \text{cf}.\text{incoming-def} \) \( \text{cf}.\text{outgoing-def} \))
    - thus \( \text{thesis} \) by auto

26
qed
thus \((\sum e \in \text{Graph}.\text{incoming} \; cf \; v. \; (\text{augmentingFlow} \; p) \; e) = \\
(\sum e \in \text{Graph}.\text{outgoing} \; cf \; v. \; (\text{augmentingFlow} \; p) \; e)\)
by \((\text{auto simp: sum-augmenting-alt})\)
\}
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** \(\text{augFlow-val}:
\)
\[\text{isAugmentingPath} \; p \implies \text{Flow}.\text{val} \; cf \; s \; (\text{augmentingFlow} \; p) = \text{resCap} \; p\]
**proof** –
assume \(\text{AUG}: \text{isAugmentingPath} \; p\)
with \(\text{augFlow-resFlow} \;
interpret \; f: \text{Flow} \; cf \; s \; t \; \text{augmentingFlow} \; p\).

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

end — Network with flow
end — Theory

6 The Ford-Fulkerson Theorem

theory \(\text{Ford-Fulkerson}\)
imports \(\text{Augmenting-Flow} \; \text{Augmenting-Path}\)
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

27
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 v. u ∈ k ∧ (u, v) ∈ E}
let ?AIN = {(v, u) | u v. 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 ("
also 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)
(auto 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)
finally show \(\text{thesis} \).

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 \(\text{?L} \leq \text{?R}\))

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: \( \text{isMaxFlow } f \implies \neg (\exists \ p. \ \text{isAugmentingPath } p) \)

**Unfolding** isMaxFlow-alt

**Proof** (rule ccontr)

- **Assume** \( \text{asm: } NFlow \ c \ s \ t \ f \wedge (\forall f'. \ NFlow \ c \ s \ t \ f' \rightarrow \text{Flow.val } c \ s \ f' \leq \text{Flow.val } c \ s \ f) \)
- **Assume** \( \text{asm-c: } \neg \neg (\exists \ p. \ \text{isAugmentingPath } p) \)

Then obtain \( p \) where \( \text{obt: } \text{isAugmentingPath } p \) by blast

- **Have** \( \text{fct1: } \text{Flow.val } c \ s \ f \rightarrow \text{Flow.val } c \ s \ f' \leq \text{Flow.val } c \ s \ f \)
- **Have** \( \text{fct2: } \text{Flow.val } c \ s \ f \rightarrow \text{Flow.val } c \ s \ f' \leq \text{Flow.val } c \ s \ f \)

Then interpret \( \text{NFlowCut } c \ s \ t \ f \ ?S \) by intro-locales

- **Show** \( \forall (u,v) \in \text{outgoing}' ?S. \ f (u,v) = c (u,v) \)

**Proof** (rule ballI, rule ccontr, clarify) — Proof by contradiction

- **Fix** \( u \ v \)
- **Assume** \( (u,v) \in \text{outgoing}' ?S \)
- **Hence** \( (u,v) \in E \wedge u \in ?S \wedge v \notin ?S \)
  - 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)

**Lemma** fofu-II-III: \( \neg (\exists \ p. \ \text{isAugmentingPath } p) \implies \exists k^' . \ NCut \ c \ s \ t \ k^' \wedge \text{val } = \text{NCut.cap } c \ k^' \)

**Proof** (intro exI conjI)

- **Let** \( ?S = c.f \text{reachableNodes } s \)
- **Assume** \( \text{asm: } \neg (\exists \ p. \ \text{isAugmentingPath } p) \)
- **Hence** \( t \notin ?S \)

Unfolding isAugmentingPath-def c.f reachableNodes-def c.f connected-def by (auto dest: c.f isSPath-pathLE)

Then show \( \text{CUT: } \text{NFlowcut } c \ s \ t \ ?S \)

**Proof** unfold-locales

- **Show** \( \text{GraphreachableNodes } c f s \subseteq V \)
  - using c.f reachable-ss-s-node resV-netV by auto
- **Show** \( s \in \text{GraphreachableNodes } c f s \)
  - unfolding GraphreachableNodes-def Graph connected-def by (metis Graph isPath.simps(1) mem Collect eq)

**Qed**

**Then interpret** \( \text{NFlowcut } c \ s \ t \ ?S \).

**Interpret** \( \text{NFlowcut } c \ s \ t \ f \ ?S \) by intro-locales

- **Have** \( \forall (u,v) \in \text{outgoing}' ?S. \ f (u,v) = c (u,v) \)

**Proof** (rule ballI, rule ccontr, clarify) — Proof by contradiction

- **Fix** \( u \ v \)
- **Assume** \( (u,v) \in \text{outgoing}' ?S \)
- **Hence** \( (u,v) \in E \wedge u \in ?S \wedge v \notin ?S \)
  - 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.E-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 $(\sum e \in outgoing' ?S. f e) = \text{cap}$

unfolding cap-def by auto

moreover

have $\forall (u,v) \in \text{incoming'} ?S. (u,v) = 0$

proof (rule ballI, rule ccontr, clarify) — Proof by contradiction

fix $u$ $v$

assume $(u,v) \in \text{incoming'} ?S$

hence $(u,v) \in E$ $u \notin ?S$ $v \in ?S$ by (auto simp: incoming'-def)

hence $(v,u) \notin E$ using no-parallel-edge by auto

assume $f (u,v) \neq 0$

hence $cf(u,v) \neq 0$

unfolding residualGraph-def using $(u,v) \in E$ $(v,u) \notin E'$ by auto

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

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 $\text{val} = \text{cap}$

unfolding flow-value[symmetric] netFlow-def by simp

qed

lemma fofu-III-I:

$\exists k. \text{NCut c s t k} \land \text{val} = \text{NCut.cap c k} \implies \text{isMaxFlow f}$

proof clarify

fix $k$

assume $\text{NCut c s t k}$

then interpret $\text{NCut c s t k}$.

interpret $\text{NFlowCut c s t f k}$ by intro-locales

assume $\text{val} = \text{cap}$

{

fix $f'$

assume $\text{Flow c s t f'}$

then interpret $\text{fc'}$.

interpret $\text{fc'}$.

have $\text{fc'}.\text{val} \leq \text{cap}$ using $\text{fc'}.\text{weak-duality}$.

also note $(\text{val} = \text{cap})[\text{symmetric}]$

finally have $\text{fc'}.\text{val} \leq \text{val}$.

}

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

**theorem ford-fulkerson**: shows
\[ \text{isMaxFlow } f \iff \neg \exists x \text{ isAugmentingPath} \land \neg \exists x \text{ isAugmentingPath} \iff \exists k. \text{NCut } c s t k \land \text{val } = \text{NCut.cap } c k \]
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

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

\[ \text{interpret } \text{NFlowCut } c s t f ?K \]

\[ \text{using } s\text{-node } s\text{-not-t by unfold-locales auto} \]

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.

\[ \text{have } \text{outgoing'} ?K = \text{incoming } t \]
\[ \text{and } \text{incoming'} ?K = \{ \} \]

\[ \text{using } \text{no-self-loop } \text{no-outgoing-t} \]

\[ \text{unfolding } \text{outgoing'-def } \text{incoming-def } \text{incoming'-def } \text{outgoing-def } V\text{-def} \]

\[ \text{by auto} \]

\[ \text{hence } (\sum_{e \in \text{incoming } t} f e) = \text{netFlow} \text{ unfolding } \text{netFlow-def by auto} \]

\[ \text{also have } \text{netFlow } = \text{val by } (\text{rule flow-value}) \]

\[ \text{also have } \text{val } = (\sum_{e \in \text{outgoing } s} f e) \text{ by } (\text{auto simp: val-alt}) \]

\[ \text{finally show } ?\text{thesis} . \]

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

**proof** –

\begin{align*}
\text{assume } & \text{isMaxFlow } f \quad \text{isMinCut } c s t k \\
\text{then interpret } & \text{Flow } c s t f + \text{NCut } c s t k \\
\quad & \text{unfolding } \text{isMaxFlow-def isMinCut-def by simp-all} \\
\text{interpret } & \text{NFlowCut } c s t f k \text{ by intro-locals} \\
\text{from } & \text{ford-fulkerson } (\text{isMaxFlow } f) \\
\text{obtain } & k' \text{ where } \text{NCut } c s t k' \text{ and val } = \text{NCut.cap } c k' \\
\quad & \text{by blast} \\
\text{thus } & \text{val } = \text{cap} \\
\quad & \text{using } (\text{isMinCut } c s t k) \text{, weak-duality} \\
\quad & \text{unfolding } \text{isMinCut-def by auto} \\
\text{qed}
\end{align*}

end — Theory

**References**


