

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

Peter Lammich and S. Reza Sefidgar

June 14, 2026

## **Abstract**

We present a formalization of flow networks and the Min-Cut-Max-Flow theorem. Our formal proof closely follows a standard textbook proof, and is accessible even without being an expert in Isabelle/HOL— the interactive theorem prover used for the formalization.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Flows, Cuts, and Networks</b>	<b>3</b>
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 . . . . .	8
2.2.3	Networks with Flow . . . . .	9
<b>3</b>	<b>Residual Graph</b>	<b>10</b>
3.1	Definition . . . . .	10
3.2	Properties . . . . .	11
<b>4</b>	<b>Augmenting Flows</b>	<b>14</b>
4.1	Augmentation of a Flow . . . . .	14
4.2	Augmentation yields Valid Flow . . . . .	14
4.2.1	Capacity Constraint . . . . .	15
4.3	Value of the Augmented Flow . . . . .	16
<b>5</b>	<b>Augmenting Paths</b>	<b>16</b>
5.1	Definitions . . . . .	16
5.2	Augmenting Flow is Valid Flow . . . . .	17
5.3	Value of Augmenting Flow is Residual Capacity . . . . .	18
<b>6</b>	<b>The Ford-Fulkerson Theorem</b>	<b>18</b>
6.1	Net Flow . . . . .	18
6.2	Ford-Fulkerson Theorem . . . . .	19
6.3	Corollaries . . . . .	19

# 1 Introduction

Computing the maximum flow of a network is an important problem in graph theory. Many other problems, like maximum-bipartite-matching, edge-disjoint-paths, circulation-demand, as well as various scheduling and resource allocating problems can be reduced to it. The Ford-Fulkerson method [3] describes a class of algorithms to solve the maximum flow problem. It is based on a corollary of the Min-Cut-Max-Flow theorem [3, 2], which states that a flow is maximal iff there exists no augmenting path.

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

## 2 Flows, Cuts, and Networks

```
theory Network
imports Graph
begin
```

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

### 2.1 Definitions

#### 2.1.1 Flows

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

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

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

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

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

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

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

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

```
begin
```

**end**

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

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

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

```
lemma conservation-const:
   $\forall v \in V - \{s, t\}. (\sum e \in \text{incoming } v. f e) = (\sum e \in \text{outgoing } v. f e)$ 
  <proof>
```

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

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

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

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

### 2.1.2 Cuts

A *cut* is a partitioning of the nodes into two sets. We define it by just specifying one of the partitions. The other partition is implicitly given by the remaining nodes.

```
type-synonym cut = node set
```

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

### 2.1.3 Networks

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

- The source has no incoming edges, and the sink has no outgoing edges.

- There are no parallel edges, i.e., for any edge, the reverse edge must not be in the network.
- Every node must lay on a path from the source to the sink.

Notes on the formalization

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

```

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

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

```

Edges have positive capacity

**lemma** edge-cap-positive:  $(u,v) \in E \implies c(u,v) > 0$   
 ⟨proof⟩

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

**lemma** reachable-is-V[simp]:  $\text{reachableNodes } s = V$   
 ⟨proof⟩

Thus, the network is actually a finite graph.

**sublocale** Finite-Graph  
 ⟨proof⟩

Our assumptions imply that there are no self loops

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

**lemma** adjacent-not-self[simp, intro!]:  $v \notin \text{adjacent-nodes } v$   
 ⟨proof⟩

A flow is maximal, if it has a maximal value

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

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

**lemma**  $t-not-s[simp]: t \neq s$  *<proof>*

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

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

**end**

#### 2.1.4 Networks with Flows and Cuts

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

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

**end**

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

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

A cut in a network separates the source from the sink

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

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

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

A minimum cut is a cut with minimum capacity.

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

## 2.2 Properties

### 2.2.1 Flows

**context**  $Preflow$   
**begin**

Only edges are labeled with non-zero flows

**lemma**  $zero-flow-simp[simp]$ :  
 $(u,v) \notin E \implies f(u,v) = 0$   
 $\langle proof \rangle$

**lemma**  $f-non-negative: 0 \leq f\ e$   
 $\langle proof \rangle$

**lemma**  $sum-f-non-negative: sum\ f\ X \geq 0 \langle proof \rangle$

**end** — Preflow

**context**  $Flow$   
**begin**

We provide a useful equivalent formulation of the conservation constraint.

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

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

**lemma**  $val-bounded$ :  
 $-(\sum e \in incoming\ s. c\ e) \leq val$   
 $val \leq (\sum e \in outgoing\ s. c\ e)$   
 $\langle proof \rangle$

**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**  $cons: \forall v \in V - \{s, t\}.$   
 $(\sum e \in incoming\ v. f\ e) = (\sum e \in outgoing\ v. f\ e)$

**shows** *Flow c s t f*  
*<proof>*

**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**  $(\sum_{e \in \text{outgoing } u} f e) = (\sum_{v \in V} f (u, v))$   
*<proof>*

**lemma** *sum-incoming-alt-flow:*  
**fixes**  $g :: \text{edge} \Rightarrow \text{'capacity}$   
**assumes**  $u \in V$   
**shows**  $(\sum_{e \in \text{incoming } u} f e) = (\sum_{v \in V} f (v, u))$   
*<proof>*  
**end** — Finite Preflow

## 2.2.2 Networks

**context** *Network*  
**begin**

**lemmas**  $[\text{simp}] = \text{no-incoming-s no-outgoing-t}$

**lemma** *incoming-s-empty[simp]: incoming s = {}*  
*<proof>*

**lemma** *outgoing-t-empty[simp]: outgoing t = {}*  
*<proof>*

**lemma** *cap-positive: e ∈ E ⇒ c e > 0*  
*<proof>*

**lemma** *V-not-empty: V ≠ {} <proof>*

**lemma** *E-not-empty: E ≠ {} <proof>*

**lemma** *card-V-ge2: card V ≥ 2*  
*<proof>*

**lemma** *zero-is-flow: Flow c s t (λ-. 0)*  
*<proof>*

**lemma** *max-flow-val-unique:*

$\llbracket is-max-flow-val\ f v1; is-max-flow-val\ f v2 \rrbracket \implies f v1 = f v2$   
 ⟨proof⟩

**end** — Network

### 2.2.3 Networks with Flow

**context** *NPreflow*

**begin**

**sublocale** *Finite-Preflow* ⟨proof⟩

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

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

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

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

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

**lemma** *excess-non-negative*:  $\forall v \in V - \{s,t\}. excess\ f\ v \geq 0$   
 ⟨proof⟩

**lemma** *excess-nodes-only*:  $excess\ f\ v > 0 \implies v \in V$   
 ⟨proof⟩

**lemma** *excess-non-negative'*:  $\forall v \in V - \{s\}. excess\ f\ v \geq 0$   
 ⟨proof⟩

**lemma** *excess-s-non-pos*:  $excess\ f\ s \leq 0$   
 ⟨proof⟩

**end** — Network with preflow

**context** *NFlow* **begin**

**sublocale** *Finite-Preflow* ⟨proof⟩

There is no outflow from the sink in a network. Thus, we can simplify the definition of the value:

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

**end**

**end** — Theory

### 3 Residual Graph

**theory** *Residual-Graph*  
**imports** *Network*  
**begin**

In this theory, we define the residual graph.

#### 3.1 Definition

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

**definition** *residualGraph* :: *- graph*  $\Rightarrow$  *- flow*  $\Rightarrow$  *- graph*

**where** *residualGraph* *c* *f*  $\equiv \lambda(u, v).$

*if*  $(u, v) \in \text{Graph}.E$  *c* *then*

$c(u, v) - f(u, v)$

*else if*  $(v, u) \in \text{Graph}.E$  *c* *then*

$f(v, u)$

*else*

$0$

**context** *Network* **begin**

**abbreviation** *cf-of*  $\equiv \text{residualGraph } c$

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

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

**lemma** *cfE-of-ss-invE*:  $cfE\text{-of } cf \subseteq E \cup E^{-1}$

*<proof>*

**lemma** *cfE-of-ss-VxV*:  $cfE\text{-of } f \subseteq V \times V$

*<proof>*

**lemma** *cfE-of-finite[simp, intro!]*: *finite* (*cfE-of* *f*)

*<proof>*

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

*<proof>*

**end**

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

**context** *NPreflow*  
**begin**

We abbreviate the residual graph by *cf*.

**abbreviation**  $cf \equiv residualGraph\ c\ f$   
**sublocale**  $cf: Graph\ cf\ \langle proof \rangle$   
**lemmas**  $cf-def = residualGraph-def[of\ c\ f]$

### 3.2 Properties

**lemmas**  $cfE-ss-invE = cfE-of-ss-invE[of\ f]$

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

**lemma**  $resV-netV[simp]: cf.V = V$   
 $\langle proof \rangle$

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

**lemma**  $cf.V = V$   
 $\langle proof \rangle$

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

**sublocale**  $cf: Finite-Graph\ cf$   
 $\langle proof \rangle$

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

**lemma**  $resE-nonNegative: cf\ e \geq 0$   
 $\langle proof \rangle$

Again, there is an automatic proof

**lemma**  $cf\ e \geq 0$   
 $\langle proof \rangle$

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

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

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

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

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

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*:  $\text{excess } f \ u > 0 \implies \text{cf.outgoing } u \neq \{\}$   
 $\langle \text{proof} \rangle$

**end** — Network with preflow

**locale** *RPreGraph* — Locale that characterizes a residual graph of a network  
 $= \text{Network} +$

**fixes** *cf*

**assumes** *EX-RPG*:  $\exists f. \text{NPreflow } c \ s \ t \ f \wedge \text{cf} = \text{residualGraph } c \ f$

**begin**

**lemma** *this-loc-rpg*:  $\text{RPreGraph } c \ s \ t \ \text{cf}$   
 $\langle \text{proof} \rangle$

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

**lemma** *f-unique*:

**assumes** *NPreflow*  $c \ s \ t \ f'$

**assumes** *A*:  $\text{cf} = \text{residualGraph } c \ f'$

**shows**  $f' = f$

$\langle \text{proof} \rangle$

**lemma** *is-NPreflow*:  $\text{NPreflow } c \ s \ t \ (\text{flow-of-cf } \text{cf})$   
 $\langle \text{proof} \rangle$

**sublocale**  $f: \text{NPreflow } c \ s \ t \ f \ \langle \text{proof} \rangle$

**lemma** *rg-is-cf[simp]*:  $\text{residualGraph } c \ f = \text{cf}$   
 $\langle \text{proof} \rangle$

**lemma** *rg-fo-inv[simp]*:  $\text{residualGraph } c \ (\text{flow-of-cf } \text{cf}) = \text{cf}$   
 $\langle \text{proof} \rangle$

**sublocale**  $\text{cf}: \text{Graph } \text{cf} \ \langle \text{proof} \rangle$

**lemma** *resV-netV[simp]*:  $\text{cf}.V = V$   
 $\langle \text{proof} \rangle$

**sublocale**  $\text{cf}: \text{Finite-Graph } \text{cf}$   
 $\langle \text{proof} \rangle$

**lemma** *E-ss-cfinvE*:  $E \subseteq \text{cf}.E \cup \text{cf}.E^{-1}$   
 $\langle \text{proof} \rangle$

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

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

**end**

**context** *NPreFlow* **begin**

**lemma** *is-RPreGraph*:  $RPreGraph\ c\ s\ t\ cf$   
 ⟨*proof*⟩

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

**end**

**lemma** (**in** *NPreFlow*)  
*flow-of-cf* (*residualGraph*  $c\ f$ ) =  $f$   
 ⟨*proof*⟩

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

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

**lemma** *this-loc-rpg*:  $RPreGraph\ c\ s\ t\ cf$   
 ⟨*proof*⟩

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

**sublocale**  $f: NFlow\ c\ s\ t\ f$  ⟨*proof*⟩

**end**

**context** *NFlow* **begin**

**lemma** *is-RGraph*:  $RGraph\ c\ s\ t\ cf$   
 ⟨*proof*⟩

The value of the flow can be computed from the residual graph.

**lemma** *val-by-cf*:  $val = (\sum (u,v) \in outgoing \ s. \ cf \ (v,u))$   
*<proof>*

**end** — Network with Flow

**lemma** (**in** *RPreGraph*) *maxflow-imp-rgraph*:  
  **assumes** *isMaxFlow* (*flow-of-cf* *cf*)  
  **shows** *RGraph* *c s t cf*  
*<proof>*

**end** — Theory

## 4 Augmenting Flows

**theory** *Augmenting-Flow*  
**imports** *Residual-Graph*  
**begin**

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

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

**context** *NFlow*  
**begin**

### 4.1 Augmentation of a Flow

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

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

We define a syntax similar to Cormen et al.:

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

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

### 4.2 Augmentation yields Valid Flow

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

conservation constraints:

**context**

— Let the *residual flow*  $f'$  be a flow in the residual graph

**fixes**  $f' :: \text{'capacity flow}$

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

**begin**

**interpretation**  $f': \text{Flow } cf\ s\ t\ f' \langle \text{proof} \rangle$

#### 4.2.1 Capacity Constraint

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

**lemma** *augment-flow-presv-cap*:

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

$\langle \text{proof} \rangle$  **lemma** *split-rflow-incoming*:

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

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

$\langle \text{proof} \rangle$

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

**context**

**fixes**  $u :: \text{node}$

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

**begin**

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

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

**private lemma** *flow-summation-aux*:

**shows**  $(\sum_{v \in E\{u\}} f'(u, v)) - (\sum_{v \in E\{u\}} f'(v, u))$   
 $= (\sum_{v \in E^{-1}\{u\}} f'(v, u)) - (\sum_{v \in E^{-1}\{u\}} f'(u, v))$   
 (**is** ?LHS = ?RHS **is** ?A - ?B = ?RHS)

$\langle \text{proof} \rangle$

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)

$\langle \text{proof} \rangle$

Note that we tried to follow the proof presented by Cormen et al. [1] as closely as possible. Unfortunately, this proof generalizes the summation to all nodes immediately, rendering the first equation invalid. Trying to fix this

error, we encountered that the step that uses the conservation constraints on the augmenting flow is more subtle as indicated in the original proof. Thus, we moved this argument to an auxiliary lemma.

**end** —  $u$  is node

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

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

### 4.3 Value of the Augmented Flow

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

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

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**  $Flow.val\ c\ s\ (f\uparrow f') = val + Flow.val\ cf\ s\ f'$   
*<proof>*

**end** — Augmenting flow

**end** — Network flow

**end** — Theory

## 5 Augmenting Paths

**theory** *Augmenting-Path*  
**imports** *Residual-Graph*  
**begin**

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

We fix a network with a preflow  $f$  on it.

**context** *NPreflow*  
**begin**

### 5.1 Definitions

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

**definition** *isAugmentingPath* ::  $path \Rightarrow bool$

**where**  $isAugmentingPath\ p \equiv cf.isSimplePath\ s\ p\ t$

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

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

**lemma**  $resCap-alt: resCap\ p = Min\ (cf'set\ p)$   
 — Useful characterization for finiteness arguments  
 $\langle proof \rangle$

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

**definition**  $augmentingFlow :: path \Rightarrow 'capacity\ flow$   
**where**  $augmentingFlow\ p \equiv \lambda(u, v).$   
    $if\ (u, v) \in (set\ p)\ then$   
      $resCap\ p$   
    $else$   
      $0$

## 5.2 Augmenting Flow is Valid Flow

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

We start with some auxiliary lemmas.

The residual capacity of an augmenting path is always positive.

**lemma**  $resCap-gzero-aux: cf.isPath\ s\ p\ t \implies 0 < resCap\ p$   
 $\langle proof \rangle$

**lemma**  $resCap-gzero: isAugmentingPath\ p \implies 0 < resCap\ p$   
 $\langle proof \rangle$

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

**lemma**  $sum-augmenting-alt:$   
**assumes**  $finite\ A$   
**shows**  $(\sum\ e \in A.\ (augmentingFlow\ p)\ e)$   
    $= resCap\ p * of-nat\ (card\ (A \cap set\ p))$   
 $\langle proof \rangle$

**lemma**  $augFlow-resFlow: isAugmentingPath\ p \implies Flow\ cf\ s\ t\ (augmentingFlow\ p)$   
 $\langle proof \rangle$

### 5.3 Value of Augmenting Flow is Residual Capacity

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

**lemma** *augFlow-val*:

*isAugmentingPath p*  $\implies$  *Flow.val cf s (augmentingFlow p) = resCap p*  
*<proof>*

**end** — Network with flow

**end** — Theory

## 6 The Ford-Fulkerson Theorem

**theory** *Ford-Fulkerson*

**imports** *Augmenting-Flow Augmenting-Path*

**begin**

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

We fix a network with a flow and a cut

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

**lemma** *finite-k[simp, intro]: finite k*  
*<proof>*

### 6.1 Net Flow

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

**definition** *netFlow :: 'capacity*

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

We can show that the net flow equals the value of the flow. Note: Cormen et al. [1] present a whole page full of summation calculations for this proof, and our formal proof also looks quite complicated.

**lemma** *flow-value: netFlow = val*  
*<proof>*

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

**corollary** *weak-duality: val  $\leq$  cap*  
*<proof>*

**end** — Cut

## 6.2 Ford-Fulkerson Theorem

**context** *NFlow* **begin**

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

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

**lemma** *fofu-II-III*:

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

**lemma** *fofu-III-I*:

$\exists k. NCut\ c\ s\ t\ k \wedge val = NCut.cap\ c\ k \implies isMaxFlow f$   
*<proof>*

Finally we can state the Ford-Fulkerson theorem:

**theorem** *ford-fulkerson*: **shows**

$isMaxFlow f \longleftrightarrow$

$\neg \exists p. isAugmentingPath\ p$  **and**  $\neg \exists p. isAugmentingPath\ p \longleftrightarrow$

$(\exists k. NCut\ c\ s\ t\ k \wedge val = NCut.cap\ c\ k)$

*<proof>*

## 6.3 Corollaries

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

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

**corollary** *inflow-t-outflow-s*:

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

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

**corollary** *noAugPath-iff-maxFlow*:  $(\nexists p. isAugmentingPath\ p) \longleftrightarrow isMaxFlow f$   
*<proof>*

**end** — Network with flow

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

**corollary** (**in** *Network*) *maxFlow-minCut*:  $\llbracket isMaxFlow\ f; isMinCut\ c\ s\ t\ k \rrbracket$

$\implies \text{Flow.val } c \ s \ f = \text{NCut.cap } c \ k$   
*<proof>*

**end** — Theory

## References

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