Formalizing Push-Relabel Algorithms

Peter Lammich and S. Reza Sefidgar

March 17, 2025

Abstract

We present a formalization of push-relabel algorithms for computing the maximum flow in a network. We start with Goldberg's et al. generic push-relabel algorithm, for which we show correctness and the time complexity bound of $O(V^2 E)$. We then derive the relabel-to-front and FIFO implementation. Using stepwise refinement techniques, we derive an efficient verified implementation.

Our formal proof of the abstract algorithms 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

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 practically most efficient algorithms to solve the maximum flow problem are push-relabel algorithms [3]. In this entry, we present a formalization of Goldberg's et al. generic push-relabel algorithm [5], and two instances: The relabel-to-front algorithm [4] and the FIFO push-relabel algorithm [5]. Using stepwise refinement techniques [9, 1, 2], we derive efficient verified implementations. Moreover, we show that the generic push-relabel algorithm has a time complexity of $O(V^2E)$.

This entry re-uses and extends theory developed for our formalization of the Edmonds-Karp maximum flow algorithm [6, 7].

While there exists another formalization of the Ford-Fulkerson method in Mizar [8], we are, to the best of our knowledge, the first that verify a polynomial maximum flow algorithm, prove a polynomial complexity bound, or provide a verified executable implementation.

2 Generic Push Relabel Algorithm

theory *Generic-Push-Relabel*
imports
  Flow-Networks.Fofu-Abs-Base
  Flow-Networks.Ford-Fulkerson
begin

2.1 Labeling

The central idea of the push-relabel algorithm is to add natural number labels l : *node* ⇒ *nat* to each node, and maintain the invariant that for all edges (u,v) in the residual graph, we have $l\ u \leq l\ v + 1$.

type-synonym *labeling* = *node* ⇒ *nat*

locale *Labeling* = *NPreflow* +
  fixes l :: *labeling*
  assumes *valid*: $(u,v) \in cf.E \Longrightarrow l(u) \leq l(v) + 1$
  assumes *lab-src*[*simp*]: $l\ s = card\ V$
  assumes *lab-sink*[*simp*]: $l\ t = 0$
begin

Generalizing validity to paths

lemma *gen-valid*: $l(u) \leq l(x) + length\ p$ **if** *cf.isPath u p x*
  ⟨*proof*⟩

In a valid labeling, there cannot be an augmenting path [Cormen 26.17]. The proof works by contradiction, using the validity constraint to show that any augmenting path would be too long for a simple path.

theorem *no-augmenting-path*: $\neg$*isAugmentingPath p*
⟨*proof*⟩

The idea of push relabel algorithms is to maintain a valid labeling, and, ultimately, arrive at a valid flow, i.e., no nodes have excess flow. We then immediately get that the flow is maximal:

corollary *no-excess-imp-maxflow*:
assumes $\forall\, u \in V - \{s,t\}.$ *excess f u* $= 0$
shows *isMaxFlow f*
⟨*proof*⟩

end — Labeling

2.2 Basic Operations

The operations of the push relabel algorithm are local operations on single nodes and edges.

2.2.1 Augmentation of Edges

context *Network*
begin

We define a function to augment a single edge in the residual graph.

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

lemma *augment-edge-zero*[*simp*]: *augment-edge f e 0* $= f$
⟨*proof*⟩

lemma *augment-edge-same*[*simp*]: $e \in E \Longrightarrow$ *augment-edge f e* $\Delta\ e = f\ e + \Delta$
⟨*proof*⟩

lemma *augment-edge-other*[*simp*]:$[\![e \in E;\ e' \neq e\ ]\!] \Longrightarrow$ *augment-edge f e* $\Delta\ e' = f\ e'$

⟨*proof*⟩

lemma *augment-edge-rev-same*[*simp*]:
$(v,u) \in E \Longrightarrow$ *augment-edge f* $(u,v)\ \Delta\ (v,u) = f\ (v,u) - \Delta$
⟨*proof*⟩

lemma *augment-edge-rev-other*[*simp*]:
$[\![(u,v)\notin E;\ e'\neq(v,u)]\!] \Longrightarrow$ *augment-edge* $f\ (u,v)\ \Delta\ e' = f\ e'$
⟨*proof*⟩

lemma *augment-edge-cf*[*simp*]: $(u,v)\in E\cup E^{-1} \Longrightarrow$
cf-of (*augment-edge* $f\ (u,v)\ \Delta$)
= (*cf-of* f)((u,v) := *cf-of* $f\ (u,v) - \Delta$, (v,u) := *cf-of* $f\ (v,u) + \Delta$)
⟨*proof*⟩

lemma *augment-edge-cf'*: $(u,v)\in$ *cfE-of* $f \Longrightarrow$
cf-of (*augment-edge* $f\ (u,v)\ \Delta$)
= (*cf-of* f)((u,v) := *cf-of* $f\ (u,v) - \Delta$, (v,u) := *cf-of* $f\ (v,u) + \Delta$)
⟨*proof*⟩

The effect of augmenting an edge on the residual graph

definition (**in** −) *augment-edge-cf* :: - *flow* ⇒ - **where**
augment-edge-cf cf
≡ $\lambda(u,v)\ \Delta.$ (*cf*)((u,v) := *cf* $(u,v) - \Delta$, (v,u) := *cf* $(v,u) + \Delta$)

lemma *cf-of-augment-edge*:
assumes *A*: $(u,v)\in$ *cfE-of* f
shows *cf-of* (*augment-edge* $f\ (u,v)\ \Delta$) = *augment-edge-cf* (*cf-of* f) $(u,v)\ \Delta$
⟨*proof*⟩

lemma *cfE-augment-ss*:
assumes *EDGE*: $(u,v)\in$ *cfE-of* f
shows *cfE-of* (*augment-edge* $f\ (u,v)\ \Delta$) $\subseteq$ *insert* (v,u) (*cfE-of* f)
⟨*proof*⟩

end — Network

context *NPreflow* **begin**

Augmenting an edge (u,v) with a flow Δ that does not exceed the available edge capacity, nor the available excess flow on the source node, preserves the preflow property.

lemma *augment-edge-preflow-preserve*: $[\![0\leq\Delta;\ \Delta \leq cf\ (u,v);\ \Delta \leq$ *excess* $f\ u]\!]$
$\Longrightarrow$ *Preflow* $c\ s\ t$ (*augment-edge* $f\ (u,v)\ \Delta$)
⟨*proof*⟩
end — Network with Preflow

2.2.2 Push Operation

context *Network*
begin

The push operation pushes as much flow as possible flow from an active node over an admissible edge.

A node is called *active* if it has positive excess, and an edge (u,v) of the residual graph is called admissible, if $l\ u = l\ v + 1$.

definition *push-precond* :: *'capacity flow* ⇒ *labeling* ⇒ *edge* ⇒ *bool*
  where *push-precond f l*
    $\equiv \lambda(u,v).\ excess\ f\ u > 0 \wedge (u,v) \in cfE\text{-}of\ f \wedge l\ u = l\ v + 1$

The maximum possible flow is determined by the available excess flow at the source node and the available capacity of the edge.

definition *push-effect* :: *'capacity flow* ⇒ *edge* ⇒ *'capacity flow*
  where *push-effect f*
    $\equiv \lambda(u,v).\ augment\text{-}edge\ f\ (u,v)\ (min\ (excess\ f\ u)\ (cf\text{-}of\ f\ (u,v)))$

lemma *push-precondI*[*intro?*]:
  $[\![excess\ f\ u > 0;\ (u,v) \in cfE\text{-}of\ f;\ l\ u = l\ v + 1]\!] \Longrightarrow push\text{-}precond\ f\ l\ (u,v)$
  ⟨*proof*⟩

2.2.3 Relabel Operation

An active node (not the sink) without any outgoing admissible edges can be relabeled.

definition *relabel-precond* :: *'capacity flow* ⇒ *labeling* ⇒ *node* ⇒ *bool*
  where *relabel-precond f l u*
    $\equiv u \neq t \wedge excess\ f\ u > 0 \wedge (\forall\ v.\ (u,v) \in cfE\text{-}of\ f \longrightarrow l\ u \neq l\ v + 1)$

The new label is computed from the neighbour's labels, to be the minimum value that will create an outgoing admissible edge.

definition *relabel-effect* :: *'capacity flow* ⇒ *labeling* ⇒ *node* ⇒ *labeling*
  where *relabel-effect f l u*
    $\equiv l(\ u := Min\ \{\ l\ v \mid v.\ (u,v) \in cfE\text{-}of\ f\ \} + 1\ )$

2.2.4 Initialization

The initial preflow exhausts all outgoing edges of the source node.

definition *pp-init-f* $\equiv \lambda(u,v).\ if\ (u=s)\ then\ c\ (u,v)\ else\ 0$

The initial labeling labels the source with $|V|$, and all other nodes with 0.

definition *pp-init-l* $\equiv (\lambda x.\ 0)(s := card\ V)$

end — Network

2.3 Abstract Correctness

We formalize the abstract correctness argument of the algorithm. It consists of two parts:

1. Execution of push and relabel operations maintain a valid labeling
2. If no push or relabel operations can be executed, the preflow is actually a flow.

This section corresponds to the proof of [Cormen 26.18].

2.3.1 Maintenance of Invariants

context *Network*
begin

lemma *pp-init-invar*: *Labeling c s t pp-init-f pp-init-l*
⟨*proof*⟩

lemma *pp-init-f-preflow*: *NPreflow c s t pp-init-f*
⟨*proof*⟩

end — Network

context *Labeling*
begin

Push operations preserve a valid labeling [Cormen 26.16].

theorem *push-pres-Labeling*:
assumes *push-precond f l e*
shows *Labeling c s t* (*push-effect f e*) *l*
⟨*proof*⟩

lemma *finite-min-cf-outgoing*[*simp*, *intro*!]: *finite* {*l v* |*v*. (*u*, *v*) ∈ *cf*.*E*}
⟨*proof*⟩

Relabel operations preserve a valid labeling [Cormen 26.16]. Moreover, they increase the label of the relabeled node [Cormen 26.15].

theorem
assumes *PRE*: *relabel-precond f l u*
shows *relabel-increase-u*: *relabel-effect f l u u* > *l u* (**is** *?G1*)
and *relabel-pres-Labeling*: *Labeling c s t f* (*relabel-effect f l u*) (**is** *?G2*)
⟨*proof*⟩

lemma *relabel-preserve-other*: *u*≠*v* ⟹ *relabel-effect f l u v* = *l v*
⟨*proof*⟩

2.3.2 Maxflow on Termination

If no push or relabel operations can be performed any more, we have arrived at a maximal flow.

theorem *push-relabel-term-imp-maxflow*:

assumes *no-push*: $\forall\,(u,v) \in cf.E.\ \neg push\text{-}precond\ f\ l\ (u,v)$
assumes *no-relabel*: $\forall\,u.\ \neg relabel\text{-}precond\ f\ l\ u$
shows *isMaxFlow f*
⟨*proof*⟩

end — Labeling

2.4 Convenience Lemmas

We define a locale to reflect the effect of a push operation

locale *push-effect-locale* = *Labeling* +
fixes *u v*
assumes *PRE*: *push-precond f l (u,v)*
begin
abbreviation $f' \equiv$ *push-effect f (u,v)*
sublocale l': *Labeling c s t f' l*
⟨*proof*⟩

lemma *uv-cf-edge*[*simp*, *intro*!]: $(u,v) \in cf.E$
⟨*proof*⟩
lemma *excess-u-pos*: *excess f u* > *0*
⟨*proof*⟩
lemma *l-u-eq*[*simp*]: $l\ u = l\ v + 1$
⟨*proof*⟩

lemma *uv-edge-cases*:
obtains (*par*) $(u,v) \in E$ $(v,u) \notin E$
| (*rev*) $(v,u) \in E$ $(u,v) \notin E$
⟨*proof*⟩

lemma *uv-nodes*[*simp*, *intro*!]: $u \in V$ $v \in V$
⟨*proof*⟩

lemma *uv-not-eq*[*simp*]: $u \neq v$ $v \neq u$
⟨*proof*⟩

definition Δ = *min* (*excess f u*) (*cf-of f (u,v)*)

lemma Δ-*positive*: $\Delta > 0$
⟨*proof*⟩

lemma f'-*alt*: f' = *augment-edge f (u,v)* Δ
⟨*proof*⟩

lemma cf'-*alt*: $l'.cf$ = *augment-edge-cf cf (u,v)* Δ
⟨*proof*⟩

lemma *excess'-u*[*simp*]: *excess* f' *u* = *excess f u* $- \Delta$
⟨*proof*⟩

lemma *excess′-v*[*simp*]: *excess f′ v* = *excess f v* + Δ
⟨*proof*⟩

lemma *excess′-other*[*simp*]:
assumes $x \neq u$ $x \neq v$
shows *excess f′ x* = *excess f x*
⟨*proof*⟩

lemma *excess′-if*:
excess f′ x = (
if x=u then excess f u − Δ
else if x=v then excess f v + Δ
else excess f x)
⟨*proof*⟩

end — Push Effect Locale

2.5 Complexity

Next, we analyze the complexity of the generic push relabel algorithm. We will show that it has a complexity of $O(V^2E)$ basic operations. Here, we often trade precise estimation of constant factors for simplicity of the proof.

2.5.1 Auxiliary Lemmas

context *Network*
begin

lemma *cardE-nz-aux*[*simp*, *intro*!]:
card E ≠ *0* *card E* ≥ *Suc 0* *card E* > *0*
⟨*proof*⟩

The number of nodes can be estimated by the number of edges. This estimation is done in various places to get smoother bounds.

lemma *card-V-est-E*: *card V* ≤ *2* ∗ *card E*
⟨*proof*⟩

end

2.5.2 Height Bound

A crucial idea of estimating the complexity is the insight that no label will exceed $2|V|-1$ during the algorithm.

We define a locale that states this invariant, and show that the algorithm maintains it. The corresponds to the proof of [Cormen 26.20].

locale *Height-Bounded-Labeling* = *Labeling* +
  assumes *height-bound*: $\forall u \in V.\ l\ u \leq 2 * card\ V - 1$
begin
  lemma *height-bound'*: $u \in V \Longrightarrow l\ u \leq 2 * card\ V - 1$
    ⟨*proof*⟩
end

lemma (**in** *Network*) *pp-init-height-bound*:
  Height-Bounded-Labeling c s t pp-init-f pp-init-l
⟨*proof*⟩

context *Height-Bounded-Labeling*
begin

As push does not change the labeling, it trivially preserves the height bound.

lemma *push-pres-height-bound*:
  assumes *push-precond f l e*
  shows *Height-Bounded-Labeling c s t* (*push-effect f e*) *l*
⟨*proof*⟩

In a valid labeling, any active node has a (simple) path to the source node in the residual graph [Cormen 26.19].

lemma (**in** *Labeling*) *excess-imp-source-path*:
  assumes *excess f u* > *0*
  obtains *p* **where** *cf.isSimplePath u p s*
⟨*proof*⟩

Relabel operations preserve the height bound [Cormen 26.20].

lemma *relabel-pres-height-bound*:
  assumes *relabel-precond f l u*
  shows *Height-Bounded-Labeling c s t f* (*relabel-effect f l u*)
⟨*proof*⟩

Thus, the total number of relabel operations is bounded by $O(V^2)$ [Cormen 26.21].

We express this bound by defining a measure function, and show that it is decreased by relabel operations.

definition (**in** *Network*) *sum-heights-measure* $l \equiv \sum v \in V.\ 2 * card\ V - l\ v$

corollary *relabel-measure*:
  assumes *relabel-precond f l u*
  shows *sum-heights-measure* (*relabel-effect f l u*) < *sum-heights-measure l*
⟨*proof*⟩
end — Height Bounded Labeling

lemma (**in** *Network*) *sum-height-measure-is-OV2*:
  sum-heights-measure $l \leq 2 * (card\ V)^2$
  ⟨*proof*⟩

2.5.3 Formulation of the Abstract Algorithm

We give a simple relational characterization of the abstract algorithm as a labeled transition system, where the labels indicate the type of operation (push or relabel) that have been executed.

context *Network*
begin

datatype *pr-operation* = *is-PUSH*: *PUSH* | *is-RELABEL*: *RELABEL*
inductive-set *pr-algo-lts*
:: (('*capacity flow*×*labeling*) × *pr-operation* × ('*capacity flow*×*labeling*)) *set*
where
push: ⟦*push-precond f l e*⟧
⟹ ((*f*,*l*),*PUSH*,(*push-effect f e*,*l*))∈*pr-algo-lts*
| *relabel*: ⟦*relabel-precond f l u*⟧
⟹ ((*f*,*l*),*RELABEL*,(*f*,*relabel-effect f l u*))∈*pr-algo-lts*

end — Network

We show invariant maintenance and correctness on termination

lemma (**in** *Height-Bounded-Labeling*) *pr-algo-maintains-hb-labeling*:
assumes ((*f*,*l*),*a*,(*f*′,*l*′)) ∈ *pr-algo-lts*
shows *Height-Bounded-Labeling c s t f*′ *l*′
⟨*proof*⟩

lemma (**in** *Height-Bounded-Labeling*) *pr-algo-term-maxflow*:
assumes (*f*,*l*)∉*Domain pr-algo-lts*
shows *isMaxFlow f*
⟨*proof*⟩

2.5.4 Saturating and Non-Saturating Push Operations

context *Network*
begin

For complexity estimation, it is distinguished whether a push operation saturates the edge or not.

definition *sat-push-precond* :: '*capacity flow* ⇒ *labeling* ⇒ *edge* ⇒ *bool*
where *sat-push-precond f l*
≡ λ(*u*,*v*). *excess f u* > *0*
∧ *excess f u* ≥ *cf-of f* (*u*,*v*)
∧ (*u*,*v*)∈*cfE-of f*
∧ *l u* = *l v* + *1*

definition *nonsat-push-precond* :: '*capacity flow* ⇒ *labeling* ⇒ *edge* ⇒ *bool*
where *nonsat-push-precond f l*
≡ λ(*u*,*v*). *excess f u* > *0*

∧ *excess f u* < *cf-of f* (*u*,*v*)
∧ (*u*,*v*)∈*cfE-of f*
∧ *l u* = *l v* + *1*

lemma *push-precond-eq-sat-or-nonsat*:
push-precond f l e ⟷ *sat-push-precond f l e* ∨ *nonsat-push-precond f l e*
⟨*proof*⟩

lemma *sat-nonsat-push-disj*:
sat-push-precond f l e ⟹ ¬*nonsat-push-precond f l e*
nonsat-push-precond f l e ⟹ ¬*sat-push-precond f l e*
⟨*proof*⟩

lemma *sat-push-alt*: *sat-push-precond f l e*
⟹ *push-effect f e* = *augment-edge f e* (*cf-of f e*)
⟨*proof*⟩

lemma *nonsat-push-alt*: *nonsat-push-precond f l* (*u*,*v*)
⟹ *push-effect f* (*u*,*v*) = *augment-edge f* (*u*,*v*) (*excess f u*)
⟨*proof*⟩

end — Network

context *push-effect-locale*
begin
lemma *nonsat-push-*Δ: *nonsat-push-precond f l* (*u*,*v*) ⟹ Δ = *excess f u*
⟨*proof*⟩
lemma *sat-push-*Δ: *sat-push-precond f l* (*u*,*v*) ⟹ Δ = *cf* (*u*,*v*)
⟨*proof*⟩

end

2.5.5 Refined Labeled Transition System

context *Network*
begin

For simpler reasoning, we make explicit the different push operations, and integrate the invariant into the LTS

datatype *pr-operation'* =
is-RELABEL': *RELABEL'*
| *is-NONSAT-PUSH'*: *NONSAT-PUSH'*
| *is-SAT-PUSH'*: *SAT-PUSH' edge*

inductive-set *pr-algo-lts'* **where**
nonsat-push': ⟦*Height-Bounded-Labeling c s t f l*; *nonsat-push-precond f l e*⟧
⟹ ((*f*,*l*),*NONSAT-PUSH'*,(*push-effect f e*,*l*))∈*pr-algo-lts'*
| *sat-push'*: ⟦*Height-Bounded-Labeling c s t f l*; *sat-push-precond f l e*⟧
⟹ ((*f*,*l*),*SAT-PUSH' e*,(*push-effect f e*,*l*))∈*pr-algo-lts'*

| *relabel'*: ⟦*Height-Bounded-Labeling c s t f l*; *relabel-precond f l u* ⟧
  $\Longrightarrow$ ((*f*,*l*),*RELABEL'*,(*f*,*relabel-effect f l u*))∈*pr-algo-lts'*

fun *project-operation* **where**
  project-operation RELABEL' = *RELABEL*
| *project-operation NONSAT-PUSH'* = *PUSH*
| *project-operation* (*SAT-PUSH'* -) = *PUSH*

lemma *is-RELABEL-project-conv*[*simp*]:
  is-RELABEL ∘ *project-operation* = *is-RELABEL'*
  ⟨*proof*⟩

lemma *is-PUSH-project-conv*[*simp*]:
  is-PUSH ∘ *project-operation* = (λ*x*. *is-SAT-PUSH' x* ∨ *is-NONSAT-PUSH' x*)
  ⟨*proof*⟩

end — Network

context *Height-Bounded-Labeling*
begin
lemma (**in** *Height-Bounded-Labeling*) *xfer-run*:
  assumes ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts*
  obtains *p'* **where** ((*f*,*l*),*p'*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
    and *p* = *map project-operation p'*
⟨*proof*⟩

lemma *xfer-relabel-bound*:
  assumes *BOUND*: ∀ *p'*. ((*f*,*l*),*p'*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
    ⟶ *length* (*filter is-RELABEL' p'*) ≤ *B*
  assumes *RUN*: ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts*
  shows *length* (*filter is-RELABEL p*) ≤ *B*
⟨*proof*⟩

lemma *xfer-push-bounds*:
  assumes *BOUND-SAT*: ∀ *p'*. ((*f*,*l*),*p'*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
    ⟶ *length* (*filter is-SAT-PUSH' p'*) ≤ *B1*
  assumes *BOUND-NONSAT*: ∀ *p'*. ((*f*,*l*),*p'*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
    ⟶ *length* (*filter is-NONSAT-PUSH' p'*) ≤ *B2*
  assumes *RUN*: ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts*
  shows *length* (*filter is-PUSH p*) ≤ *B1* + *B2*
⟨*proof*⟩

end — Height Bounded Labeling

2.5.6 Bounding the Relabel Operations

lemma (**in** *Network*) *relabel-action-bound'*:

assumes *A*: (*fxl*,*p*,*fxl'*) ∈ *trcl pr-algo-lts'*
shows *length* (*filter* (*is-RELABEL'*) *p*) ≤ *2* ∗ (*card V*)^2
⟨*proof*⟩

lemma (**in** *Height-Bounded-Labeling*) *relabel-action-bound*:
assumes *A*: ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts*
shows *length* (*filter* (*is-RELABEL*) *p*) ≤ *2* ∗ (*card V*)^2
⟨*proof*⟩

2.5.7 Bounding the Saturating Push Operations

context *Network*
begin

The basic idea is to estimate the saturating push operations per edge: After a saturating push, the edge disappears from the residual graph. It can only re-appear due to a push over the reverse edge, which requires relabeling of the nodes.

The estimation in [Cormen 26.22] uses the same idea. However, it invests some extra work in getting a more precise constant factor by counting the pushes for an edge and its reverse edge together.

lemma *labels-path-increasing*:
assumes ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
shows *l u* ≤ *l' u*
⟨*proof*⟩

lemma *edge-reappears-at-increased-labeling*:
assumes ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
assumes *l u* ≥ *l v* + *1*
assumes (*u*,*v*) ∉ *cfE-of f*
assumes *E'*: (*u*,*v*) ∈ *cfE-of f'*
shows *l v* < *l' v*
⟨*proof*⟩

lemma *sat-push-edge-action-bound'*:
assumes ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
shows *length* (*filter* ((=) (*SAT-PUSH' e*)) *p*) ≤ *2*∗*card V*
⟨*proof*⟩

lemma *sat-push-action-bound'*:
assumes *A*: ((*f*,*l*),*p*,(*f'*,*l'*)) ∈ *trcl pr-algo-lts'*
shows *length* (*filter is-SAT-PUSH' p*) ≤ *4* ∗ *card V* ∗ *card E*
⟨*proof*⟩

end — Network

2.5.8 Bounding the Non-Saturating Push Operations

For estimating the number of non-saturating push operations, we define a potential function that is the sum of the labels of all active nodes, and examine the effect of the operations on this potential:

- A non-saturating push deactivates the source node and may activate the target node. As the source node's label is higher, the potential decreases.
- A saturating push may activate a node, thus increasing the potential by $O(V)$.
- A relabel operation may increase the potential by $O(V)$.

As there are at most $O(V^2)$ relabel and $O(VE)$ saturating push operations, the above bounds suffice to yield an $O(V^2E)$ bound for the non-saturating push operations.
This argumentation corresponds to [Cormen 26.23].

Sum of heights of all active nodes

definition (**in** *Network*) *nonsat-potential f l* ≡ *sum l* {*v*∈*V*. *excess f v* > *0*}

context *Height-Bounded-Labeling*
begin

The potential does not exceed $O(V^2)$.

lemma *nonsat-potential-bound*:
  shows *nonsat-potential f l* ≤ *2* ∗ (*card V*)^*2*
⟨*proof*⟩

A non-saturating push decreases the potential.

lemma *nonsat-push-decr-nonsat-potential*:
  assumes *nonsat-push-precond f l e*
  shows *nonsat-potential* (*push-effect f e*) *l* < *nonsat-potential f l*
⟨*proof*⟩

A saturating push increases the potential by $O(V)$.

lemma *sat-push-nonsat-potential*:
  assumes *PRE*: *sat-push-precond f l e*
  shows *nonsat-potential* (*push-effect f e*) *l*
    ≤ *nonsat-potential f l* + *2* ∗ *card V*
⟨*proof*⟩

A relabeling increases the potential by at most $O(V)$

lemma *relabel-nonsat-potential*:
  assumes *PRE*: *relabel-precond f l u*

shows *nonsat-potential f* (*relabel-effect f l u*)
$\leq$ *nonsat-potential f l* + *2* $*$ *card V*
⟨*proof*⟩

end — Height Bounded Labeling

context *Network*
begin

lemma *nonsat-push-action-bound'*:
assumes *A*: $((f,l),p,(f',l')) \in$ *trcl pr-algo-lts'*
shows *length* (*filter is-NONSAT-PUSH' p*) $\leq$ *18* $*$ (*card V*)^2 $*$ *card E*
⟨*proof*⟩

end — Network

2.5.9 Assembling the Final Theorem

We combine the bounds for saturating and non-saturating push operations.

lemma (**in** *Height-Bounded-Labeling*) *push-action-bound*:
assumes *A*: $((f,l),p,(f',l')) \in$ *trcl pr-algo-lts*
shows *length* (*filter* (*is-PUSH*) *p*) $\leq$ *22* $*$ (*card V*)^2 $*$ *card E*
⟨*proof*⟩

We estimate the cost of a push by $O(1)$, and of a relabel operation by $O(V)$

fun (**in** *Network*) *cost-estimate* :: *pr-operation* $\Rightarrow$ *nat* **where**
cost-estimate RELABEL = *card V*
| *cost-estimate PUSH* = *1*

We show the complexity bound of $O(V^2E)$ when starting from any valid labeling [Cormen 26.24].

theorem (**in** *Height-Bounded-Labeling*) *pr-algo-cost-bound*:
assumes *A*: $((f,l),p,(f',l')) \in$ *trcl pr-algo-lts*
shows ($\sum$ *a*←*p*. *cost-estimate a*) $\leq$ *26* $*$ (*card V*)^*2* $*$ *card E*
⟨*proof*⟩

2.6 Main Theorem: Correctness and Complexity

Finally, we state the main theorem of this section: If the algorithm executes some steps from the beginning, then

1. If no further steps are possible from the reached state, we have computed a maximum flow [Cormen 26.18].
2. The cost of these steps is bounded by $O(V^2E)$ [Cormen 26.24]. Note that this also implies termination.

theorem (**in** *Network*) *generic-preflow-push-OV2E-and-correct*:
  assumes *A*: $((pp\text{-}init\text{-}f,\ pp\text{-}init\text{-}l),\ p,\ (f,\ l)) \in trcl\ pr\text{-}algo\text{-}lts$
  shows $(\sum x{\leftarrow}p.\ cost\text{-}estimate\ x) \leq 26 * (card\ V)\hat{}2 * card\ E$ (**is** *?G1*)
    and $(f,l) \notin Domain\ pr\text{-}algo\text{-}lts \longrightarrow isMaxFlow\ f$ (**is** *?G2*)
⟨*proof*⟩

2.7 Convenience Tools for Implementation

context *Network*
begin

In order to show termination of the algorithm, we only need a well-founded relation over push and relabel steps

inductive-set *pr-algo-rel* **where**
  push: ⟦*Height-Bounded-Labeling c s t f l*; *push-precond f l e*⟧
    $\Longrightarrow$ ((*push-effect f e*,*l*),(*f*,*l*))∈*pr-algo-rel*
| *relabel*: ⟦*Height-Bounded-Labeling c s t f l*; *relabel-precond f l u*⟧
    $\Longrightarrow$ ((*f*,*relabel-effect f l u*),(*f*,*l*))∈*pr-algo-rel*

lemma *pr-algo-rel-alt*: *pr-algo-rel* =
    { ((*push-effect f e*,*l*),(*f*,*l*)) | *f e l*.
        Height-Bounded-Labeling c s t f l ∧ *push-precond f l e* }
  ∪ { ((*f*, *relabel-effect f l u*), (*f*,*l*)) | *f u l*.
        Height-Bounded-Labeling c s t f l ∧ *relabel-precond f l u* }
  ⟨*proof*⟩

definition *pr-algo-len-bound* $\equiv 2 * (card\ V)^2 + 22 * (card\ V)^2 * card\ E$

lemma (**in** *Height-Bounded-Labeling*) *pr-algo-lts-length-bound*:
  assumes *A*: $((f,\ l),\ p,\ (f',\ l')) \in trcl\ pr\text{-}algo\text{-}lts$
  shows *length p* ≤ *pr-algo-len-bound*
⟨*proof*⟩

lemma (**in** *Height-Bounded-Labeling*) *path-set-finite*:
  finite $\{p.\ \exists f'\ l'.\ ((f,l),p,(f',l')) \in trcl\ pr\text{-}algo\text{-}lts\}$
⟨*proof*⟩

definition *pr-algo-measure*
  $\equiv \lambda(f,l).\ Max\ \{length\ p\ |p.\ \exists\ aa\ ba.\ ((f,\ l),\ p,\ aa,\ ba) \in trcl\ pr\text{-}algo\text{-}lts\}$

lemma *pr-algo-measure*:
  assumes $(fl',fl) \in pr\text{-}algo\text{-}rel$
  shows *pr-algo-measure fl′* < *pr-algo-measure fl*
  ⟨*proof*⟩

lemma *wf-pr-algo-rel*[*simp*, *intro*!]: *wf pr-algo-rel*
  ⟨*proof*⟩

end — Network

2.8 Gap Heuristics

context *Network*
begin

If we find a label value k that is assigned to no node, we may relabel all nodes v with $k < l\ v < card\ V$ to $card\ V + 1$.

definition *gap-precond* $l\ k \equiv \forall\, v \in V.\ l\ v \neq k$
definition *gap-effect* $l\ k$
  $\equiv \lambda v.$ *if* $k < l\ v \wedge l\ v < card\ V$ *then* $card\ V + 1$ *else* $l\ v$

The gap heuristics preserves a valid labeling.

lemma (**in** *Labeling*) *gap-pres-Labeling*:
  assumes *PRE*: *gap-precond* $l\ k$
  defines $l' \equiv$ *gap-effect* $l\ k$
  shows *Labeling* $c\ s\ t\ f\ l'$
⟨*proof*⟩

The gap heuristics also preserves the height bounds.

lemma (**in** *Height-Bounded-Labeling*) *gap-pres-hb-labeling*:
  assumes *PRE*: *gap-precond* $l\ k$
  defines $l' \equiv$ *gap-effect* $l\ k$
  shows *Height-Bounded-Labeling* $c\ s\ t\ f\ l'$
⟨*proof*⟩

We combine the regular relabel operation with the gap heuristics: If relabeling results in a gap, the gap heuristics is applied immediately.

definition *gap-relabel-effect* $f\ l\ u \equiv$ *let* $l' =$ *relabel-effect* $f\ l\ u$ *in*
  if (*gap-precond* l' $(l\ u)$) *then* *gap-effect* l' $(l\ u)$ *else* l'

The combined gap-relabel operation preserves a valid labeling.

lemma (**in** *Labeling*) *gap-relabel-pres-Labeling*:
  assumes *PRE*: *relabel-precond* $f\ l\ u$
  defines $l' \equiv$ *gap-relabel-effect* $f\ l\ u$
  shows *Labeling* $c\ s\ t\ f\ l'$
  ⟨*proof*⟩

The combined gap-relabel operation preserves the height-bound.

lemma (**in** *Height-Bounded-Labeling*) *gap-relabel-pres-hb-labeling*:
  assumes *PRE*: *relabel-precond* $f\ l\ u$
  defines $l' \equiv$ *gap-relabel-effect* $f\ l\ u$
  shows *Height-Bounded-Labeling* $c\ s\ t\ f\ l'$
  ⟨*proof*⟩

2.8.1 Termination with Gap Heuristics

Intuitively, the algorithm with the gap heuristics terminates because relabeling according to the gap heuristics preserves the invariant and increases some labels towards their upper bound.

Formally, the simplest way is to combine a heights measure function with the already established measure for the standard algorithm:

lemma (**in** *Height-Bounded-Labeling*) *gap-measure*:
  assumes *gap-precond l k*
  shows *sum-heights-measure* (*gap-effect l k*) ≤ *sum-heights-measure l*
  ⟨*proof*⟩

lemma (**in** *Height-Bounded-Labeling*) *gap-relabel-measure*:
  assumes *PRE*: *relabel-precond f l u*
  shows *sum-heights-measure* (*gap-relabel-effect f l u*) < *sum-heights-measure l*
  ⟨*proof*⟩

Analogously to *pr-algo-rel*, we provide a well-founded relation that over-approximates the steps of a push-relabel algorithm with gap heuristics.

inductive-set *gap-algo-rel* **where**
  push: ⟦*Height-Bounded-Labeling c s t f l*; *push-precond f l e*⟧
    ⟹ ((*push-effect f e*,*l*),(*f*,*l*))∈*gap-algo-rel*
| *relabel*: ⟦*Height-Bounded-Labeling c s t f l*; *relabel-precond f l u*⟧
    ⟹ ((*f*,*gap-relabel-effect f l u*),(*f*,*l*))∈*gap-algo-rel*

lemma *wf-gap-algo-rel*[*simp*, *intro*!]: *wf gap-algo-rel*
⟨*proof*⟩

end — Network

end
theory *Prpu-Common-Inst*
imports
  Flow-Networks.Refine-Add-Fofu
  Generic-Push-Relabel
begin

context *Network*
begin
  definition *relabel f l u* ≡ *do* {
    assert (*Height-Bounded-Labeling c s t f l*);
    assert (*relabel-precond f l u*);
    assert (*u*∈*V*−{*s*,*t*});
    return (*relabel-effect f l u*)
  }

  definition *gap-relabel f l u* ≡ *do* {

```
    assert (u∈V−{s,t});
    assert (Height-Bounded-Labeling c s t f l);
    assert (relabel-precond f l u);
    assert (l u < 2∗card V ∧ relabel-effect f l u u < 2∗card V);
    return (gap-relabel-effect f l u)
  }

  definition push f l ≡ λ(u,v). do {
    assert (push-precond f l (u,v));
    assert (Labeling c s t f l);
    return (push-effect f (u,v))
  }

end

end
```

3 FIFO Push Relabel Algorithm

```
theory Fifo-Push-Relabel
imports
  Flow-Networks.Refine-Add-Fofu
  Generic-Push-Relabel
begin
```

The FIFO push-relabel algorithm maintains a first-in-first-out queue of active nodes. As long as the queue is not empty, it discharges the first node of the queue.

Discharging repeatedly applied push operations from the node. If no more push operations are possible, and the node is still active, it is relabeled and enqueued.

Moreover, we implement the gap heuristics, which may accelerate relabeling if there is a gap in the label values, i.e., a label value that is assigned to no node.

3.1 Implementing the Discharge Operation

```
context Network
begin
```

First, we implement push and relabel operations that maintain a queue of all active nodes.

```
definition fifo-push f l Q ≡ λ(u,v). do {
  assert (push-precond f l (u,v));
  assert (Labeling c s t f l);
  let Q = (if v≠s ∧ v≠t ∧ excess f v = 0 then Q@[v] else Q);
  return (push-effect f (u,v),Q)
```

}

For the relabel operation, we assume that only active nodes are relabeled, and enqueue the relabeled node.

```
definition fifo-gap-relabel f l Q u ≡ do {
  assert (u∈V−{s,t});
  assert (Height-Bounded-Labeling c s t f l);
  let Q = Q@[u];
  assert (relabel-precond f l u);
  assert (l u < 2∗card V ∧ relabel-effect f l u u < 2∗card V);
  let l = gap-relabel-effect f l u;
  return (l,Q)
}
```

The discharge operation iterates over the edges, and pushes flow, as long as then node is active. If the node is still active after all edges have been saturated, the node is relabeled.

```
definition fifo-discharge f0 l Q ≡ do {
  assert (Q≠[]);
  let u=hd Q; let Q=tl Q;
  assert (u∈V ∧ u≠s ∧ u≠t);

   (f,l,Q) ← FOREACHc {v . (u,v)∈cfE-of f0} (λ(f,l,Q). excess f u ≠ 0) (λv
(f,l,Q). do {
    if (l u = l v + 1) then do {
      (f′,Q) ← fifo-push f l Q (u,v);
      assert (∀ v′. v′≠v ⟶ cf-of f′ (u,v′) = cf-of f (u,v′));
      return (f′,l,Q)
    } else return (f,l,Q)
  }) (f0,l,Q);

  if excess f u ≠ 0 then do {
    (l,Q) ← fifo-gap-relabel f l Q u;
    return (f,l,Q)
  } else do {
    return (f,l,Q)
  }
}
```

We will show that the discharge operation maintains the invariant that the queue is disjoint and contains exactly the active nodes:

definition *Q-invar f Q* ≡ *distinct Q* ∧ *set Q* = { *v*∈*V*−{*s*,*t*}. *excess f v* ≠ *0* }

Inside the loop of the discharge operation, we will use the following version of the invariant:

definition *QD-invar u f Q* ≡ *u*∈*V*−{*s*,*t*} ∧ *distinct Q* ∧ *set Q* = { *v*∈*V*−{*s*,*t*,*u*}. *excess f v* ≠ *0* }

lemma *Q-invar-when-discharged1*: ⟦*QD-invar u f Q*; *excess f u* = *0*⟧ ⟹ *Q-invar f Q*
  ⟨*proof*⟩

lemma *Q-invar-when-discharged2*: ⟦*QD-invar u f Q*; *excess f u* ≠ *0*⟧ ⟹ *Q-invar f* (*Q*@[*u*])
  ⟨*proof*⟩

lemma (**in** *Labeling*) *push-no-activate-pres-QD-invar*:
  fixes *v*
  assumes *INV*: *QD-invar u f Q*
  assumes *PRE*: *push-precond f l* (*u*,*v*)
  assumes *VC*: *s*=*v* ∨ *t*=*v* ∨ *excess f v* ≠ *0*
  shows *QD-invar u* (*push-effect f* (*u*,*v*)) *Q*
⟨*proof*⟩

lemma (**in** *Labeling*) *push-activate-pres-QD-invar*:
  fixes *v*
  assumes *INV*: *QD-invar u f Q*
  assumes *PRE*: *push-precond f l* (*u*,*v*)
  assumes *VC*: *s*≠*v*   *t*≠*v* **and** [*simp*]: *excess f v* = *0*
  shows *QD-invar u* (*push-effect f* (*u*,*v*)) (*Q*@[*v*])
⟨*proof*⟩

Main theorem for the discharge operation: It maintains a height bounded labeling, the invariant for the FIFO queue, and only performs valid steps due to the generic push-relabel algorithm with gap-heuristics.

theorem *fifo-discharge-correct*[*THEN order-trans*, *refine-vcg*]:
  assumes *DINV*: *Height-Bounded-Labeling c s t f l*
  assumes *QINV*: *Q-invar f Q* **and** *QNE*: *Q*≠[]
  shows *fifo-discharge f l Q* ≤ *SPEC* (λ(*f'*,*l'*,*Q'*).
      Height-Bounded-Labeling c s t f' l'
    ∧ *Q-invar f' Q'*
    ∧ ((*f'*,*l'*),(*f*,*l*))∈*gap-algo-rel*⁺
)
⟨*proof*⟩

end — Network

3.2 Main Algorithm

context *Network*
begin

The main algorithm initializes the flow, labeling, and the queue, and then applies the discharge operation until the queue is empty:

definition *fifo-push-relabel* ≡ *do* {
  let f = *pp-init-f*;

let l = *pp-init-l*;

$Q \leftarrow$ *spec l. distinct l* $\wedge$ *set* $l = \{v{\in}V - \{s,t\}.\ \textit{excess}\ f\ v \neq 0\}$; — TODO: This is exactly $E``\{s\} - \{t\}$!

$(f,l,\text{-}) \leftarrow \textit{while}_T\ (\lambda(f,l,Q).\ Q \neq [])\ (\lambda(f,l,Q).\ \textit{do}\ \{$
fifo-discharge f l Q
$\})\ (f,l,Q)$;

assert (*Height-Bounded-Labeling c s t f l*);
return f
}

Having proved correctness of the discharge operation, the correctness theorem of the main algorithm is straightforward: As the discharge operation implements the generic algorithm, the loop will terminate after finitely many steps. Upon termination, the queue that contains exactly the active nodes is empty. Thus, all nodes are inactive, and the resulting preflow is actually a maximal flow.

theorem *fifo-push-relabel-correct*:
fifo-push-relabel $\leq$ *SPEC isMaxFlow*
⟨*proof*⟩

end — Network

end

4 Topological Ordering of Graphs

theory *Graph-Topological-Ordering*
imports
Refine-Imperative-HOL.Sepref-Misc
List−Index.List-Index
begin

4.1 List-Before Relation

Two elements of a list are in relation if the first element comes (strictly) before the second element.

definition *list-before-rel l* $\equiv$ { (a,b). $\exists$ *l1 l2 l3*. *l*=*l1*@*a*#*l2*@*b*#*l3* }

list-before only relates elements of the list

lemma *list-before-rel-on-elems*: *list-before-rel l* $\subseteq$ *set l* $\times$ *set l*
⟨*proof*⟩

Irreflexivity of list-before is equivalent to the elements of the list being disjoint.

lemma *list-before-irrefl-eq-distinct*: *irrefl* (*list-before-rel l*) $\longleftrightarrow$ *distinct l*
⟨*proof*⟩

Alternative characterization via indexes

lemma *list-before-rel-alt*: *list-before-rel l* = { (*l*!*i*, *l*!*j*) | *i j*. *i*<*j* ∧ *j*<*length l* }
⟨*proof*⟩

list-before is a strict ordering, i.e., it is transitive and asymmetric.

lemma *list-before-trans*[*trans*]: *distinct l* $\Longrightarrow$ *trans* (*list-before-rel l*)
⟨*proof*⟩

lemma *list-before-asym*: *distinct l* $\Longrightarrow$ *asym* (*list-before-rel l*)
⟨*proof*⟩

Structural properties on the list

lemma *list-before-rel-empty*[*simp*]: *list-before-rel* [] = {}
⟨*proof*⟩

lemma *list-before-rel-cons*: *list-before-rel* (*x*#*l*) = ({*x*}×*set l*) ∪ *list-before-rel l*
⟨*proof*⟩

4.2 Topological Ordering

A topological ordering of a graph (binary relation) is an enumeration of its nodes, such that for any two nodes x,y with x being enumerated earlier than y, there is no path from y to x in the graph.

We define the predicate *is-top-sorted* to capture the sortedness criterion, but not the completeness criterion, i.e., the list needs not contain all nodes of the graph.

definition *is-top-sorted R l* ≡ *list-before-rel l* ∩ $(R^*)^{-1}$ = {}
lemma *is-top-sorted-alt*: *is-top-sorted R l* $\longleftrightarrow$ (∀ *x y*. (*x*,*y*)∈*list-before-rel l* $\longrightarrow$ (*y*,*x*)$\notin R^*$)
⟨*proof*⟩

lemma *is-top-sorted-empty-rel*[*simp*]: *is-top-sorted* {} *l* $\longleftrightarrow$ *distinct l*
⟨*proof*⟩

lemma *is-top-sorted-empty-list*[*simp*]: *is-top-sorted R* []
⟨*proof*⟩

A topological sorted list must be distinct

lemma *is-top-sorted-distinct*:
assumes *is-top-sorted R l*
shows *distinct l*
⟨*proof*⟩

lemma *is-top-sorted-cons*: *is-top-sorted R* $(x\#l) \longleftrightarrow (\{x\}\times set\ l \cap (R^*)^{-1} = \{\})$ $\wedge$ *is-top-sorted R l*
⟨*proof*⟩

lemma *is-top-sorted-append*: *is-top-sorted R* $(l1@l2)$
$\longleftrightarrow (set\ l1\times set\ l2 \cap (R^*)^{-1} = \{\}) \wedge$ *is-top-sorted R l1* $\wedge$ *is-top-sorted R l2*
⟨*proof*⟩

lemma *is-top-sorted-remove-elem*: *is-top-sorted R* $(l1@x\#l2) \Longrightarrow$ *is-top-sorted R* $(l1@l2)$
⟨*proof*⟩

Removing edges from the graph preserves topological sorting

lemma *is-top-sorted-antimono*:
assumes $R \subseteq R'$
assumes *is-top-sorted* R' *l*
shows *is-top-sorted R l*
⟨*proof*⟩

Adding a node to the graph, which has no incoming edges preserves topological ordering.

lemma *is-top-sorted-isolated-constraint*:
assumes $R' \subseteq R \cup \{x\}\times X \quad R' \cap UNIV\times\{x\} = \{\}$
assumes $x \notin set\ l$
assumes *is-top-sorted R l*
shows *is-top-sorted* R' *l*
⟨*proof*⟩

end

5 Relabel-to-Front Algorithm

theory *Relabel-To-Front*
imports
Prpu-Common-Inst
Graph-Topological-Ordering
begin

As an example for an implementation, Cormen et al. discuss the relabel-to-front algorithm. It iterates over a queue of nodes, discharging each node, and putting a node to the front of the queue if it has been relabeled.

5.1 Admissible Network

The admissible network consists of those edges over which we can push flow.

context *Network*
begin
definition *adm-edges* :: *'capacity flow* ⇒ (*nat*⇒*nat*) ⇒ -
where *adm-edges f l* ≡ {(*u*,*v*)∈*cfE-of f*. *l u* = *l v* + *1*}

lemma *adm-edges-inv-disj*: *adm-edges f l* ∩ (*adm-edges f l*)$^{-1}$ = {}
⟨*proof*⟩

lemma *finite-adm-edges*[*simp*, *intro*!]: *finite* (*adm-edges f l*)
⟨*proof*⟩

end — Network

The edge of a push operation is admissible.

lemma (**in** *push-effect-locale*) *uv-adm*: (*u*,*v*)∈*adm-edges f l*
⟨*proof*⟩

A push operation will not create new admissible edges, but the edge that we pushed over may become inadmissible [Cormen 26.27].

lemma (**in** *Labeling*) *push-adm-edges*:
assumes *push-precond f l e*
shows *adm-edges f l* − {*e*} ⊆ *adm-edges* (*push-effect f e*) *l* (**is** *?G1*)
and *adm-edges* (*push-effect f e*) *l* ⊆ *adm-edges f l* (**is** *?G2*)
⟨*proof*⟩

After a relabel operation, there is at least one admissible edge leaving the relabeled node, but no admissible edges do enter the relabeled node [Cormen 26.28]. Moreover, the part of the admissible network not adjacent to the relabeled node does not change.

lemma (**in** *Labeling*) *relabel-adm-edges*:
assumes *PRE*: *relabel-precond f l u*
defines *l'* ≡ *relabel-effect f l u*
shows *adm-edges f l'* ∩ *cf.outgoing u* ≠ {} (**is** *?G1*)
and *adm-edges f l'* ∩ *cf.incoming u* = {} (**is** *?G2*)
and *adm-edges f l'* − *cf.adjacent u* = *adm-edges f l* − *cf.adjacent u* (**is** *?G3*)
⟨*proof*⟩

5.2 Neighbor Lists

For each node, the algorithm will cycle through the adjacent edges when discharging. This cycling takes place across the boundaries of discharge operations, i.e. when a node is discharged, discharging will start at the edge where the last discharge operation stopped.

The crucial invariant for the neighbor lists is that already visited edges are not admissible.

Formally, we maintain a function $n :: node \Rightarrow node\ set$ from each node to the set of target nodes of not yet visited edges.

locale *neighbor-invar* = *Height-Bounded-Labeling* +
  fixes $n :: node \Rightarrow node\ set$
  assumes *neighbors-adm*: $[\![v \in adjacent\text{-}nodes\ u - n\ u]\!] \Longrightarrow (u,v) \notin adm\text{-}edges\ f\ l$
  assumes *neighbors-adj*: $n\ u \subseteq adjacent\text{-}nodes\ u$
  assumes *neighbors-finite*[*simp*, *intro*!]: *finite* (*n u*)
begin

lemma *nbr-is-hbl*: *Height-Bounded-Labeling c s t f l* ⟨*proof*⟩

lemma *push-pres-nbr-invar*:
  assumes *PRE*: *push-precond f l e*
  shows *neighbor-invar c s t* (*push-effect f e*) *l n*
⟨*proof*⟩

lemma *relabel-pres-nbr-invar*:
  assumes *PRE*: *relabel-precond f l u*
  shows *neighbor-invar c s t f* (*relabel-effect f l u*) (*n*(*u*:=*adjacent-nodes u*))
⟨*proof*⟩

lemma *excess-nz-iff-gz*: $[\![\ u{\in}V;\ u{\neq}s\ ]\!] \Longrightarrow excess\ f\ u \neq 0 \longleftrightarrow excess\ f\ u > 0$
  ⟨*proof*⟩

lemma *no-neighbors-relabel-precond*:
  assumes $n\ u = \{\}$ $u{\neq}t$ $u{\neq}s$ $u{\in}V$ $excess\ f\ u \neq 0$
  shows *relabel-precond f l u*
  ⟨*proof*⟩

lemma *remove-neighbor-pres-nbr-invar*: $(u,v){\notin}adm\text{-}edges\ f\ l$
  $\Longrightarrow$ *neighbor-invar c s t f l* $(n\ (u := n\ u - \{v\}))$
  ⟨*proof*⟩

end

5.3 Discharge Operation

context *Network*
begin

The discharge operation performs push and relabel operations on a node until it becomes inactive. The lemmas in this section are based on the ideas described in the proof of [Cormen 26.29].

definition *discharge f l n u* ≡ *do* {
  assert $(u \in V - \{s,t\})$;
  $while_T\ (\lambda(f,l,n).\ excess\ f\ u \neq 0)\ (\lambda(f,l,n).\ do\ \{$
    $v \leftarrow select\ v.\ v{\in}n\ u$;
    case v of

```
      None ⇒ do {
        l ← relabel f l u;
        return (f,l,n(u := adjacent-nodes u))
      }
    | Some v ⇒ do {
        assert (v∈V ∧ (u,v)∈E∪E⁻¹);
        if ((u,v) ∈ cfE-of f ∧ l u = l v + 1) then do {
          f ← push f l (u,v);
          return (f,l,n)
        } else do {
          assert ( (u,v) ∉ adm-edges f l );
          return (f,l,n( u := n u − {v} ))
        }
      }
  }) (f,l,n)
}
```

end — Network

Invariant for the discharge loop

locale *discharge-invar* =
    neighbor-invar c s t f l n
  + *lo*: *neighbor-invar c s t fo lo no*
  for *c s t* **and** *u* :: *node* **and** *fo lo no f l n* +
  assumes *lu-incr*: *lo u ≤ l u*
  assumes *u-node*: *u∈V−{s,t}*
  assumes *no-relabel-adm-edges*: *lo u = l u ⟹ adm-edges f l ⊆ adm-edges fo lo*
  assumes *no-relabel-excess*:
   ⟦lo u = l u; u≠v; excess fo v ≠ excess f v⟧ ⟹ (u,v)∈adm-edges fo lo
  assumes *adm-edges-leaving-u*: *(u′,v)∈adm-edges f l − adm-edges fo lo ⟹ u′=u*
  assumes *relabel-u-no-incoming-adm*: *lo u ≠ l u ⟹ (v,u)∉adm-edges f l*
  assumes *algo-rel*: *((f,l),(fo,lo)) ∈ pr-algo-rel**
begin

lemma *u-node-simp1* [*simp*]: *u≠s   u≠t   s≠u   t≠u* ⟨*proof*⟩
lemma *u-node-simp2* [*simp, intro*!]: *u∈V* ⟨*proof*⟩

lemma *dis-is-lbl*: *Labeling c s t f l* ⟨*proof*⟩
lemma *dis-is-hbl*: *Height-Bounded-Labeling c s t f l* ⟨*proof*⟩
lemma *dis-is-nbr*: *neighbor-invar c s t f l n* ⟨*proof*⟩

lemma *new-adm-imp-relabel*:
  (u′,v)∈adm-edges f l − adm-edges fo lo ⟹ lo u ≠ l u
  ⟨*proof*⟩

lemma *push-pres-dis-invar*:
  assumes *PRE*: *push-precond f l (u,v)*
  shows *discharge-invar c s t u fo lo no (push-effect f (u,v)) l n*
⟨*proof*⟩

lemma *relabel-pres-dis-invar*:
  assumes *PRE*: *relabel-precond f l u*
  shows *discharge-invar c s t u fo lo no f*
    (*relabel-effect f l u*) (*n*(*u* := *adjacent-nodes u*))
⟨*proof*⟩

lemma *push-precondI-nz*:
  ⟦*excess f u* ≠ *0*; (*u*,*v*)∈*cfE-of f*; *l u* = *l v* + *1*⟧ ⟹ *push-precond f l* (*u*,*v*)
  ⟨*proof*⟩

lemma *remove-neighbor-pres-dis-invar*:
  assumes *PRE*: (*u*,*v*)∉*adm-edges f l*
  defines *n′* ≡ *n* (*u* := *n u* − {*v*})
  shows *discharge-invar c s t u fo lo no f l n′*
⟨*proof*⟩

lemma *neighbors-in-V*: *v*∈*n u* ⟹ *v*∈*V*
  ⟨*proof*⟩

lemma *neighbors-in-E*: *v*∈*n u* ⟹ (*u*,*v*)∈*E*∪E^{-1}
  ⟨*proof*⟩

lemma *relabeled-node-has-outgoing*:
  assumes *relabel-precond f l u*
  shows ∃ *v*. (*u*,*v*)∈*cfE-of f*
  ⟨*proof*⟩

end

lemma (**in** *neighbor-invar*) *discharge-invar-init*:
  assumes *u*∈*V*−{*s*,*t*}
  shows *discharge-invar c s t u f l n f l n*
  ⟨*proof*⟩

context *Network* **begin**

The discharge operation preserves the invariant, and discharges the node.

lemma *discharge-correct*[*THEN order-trans*, *refine-vcg*]:
  assumes *DINV*: *neighbor-invar c s t f l n*
  assumes *NOT-ST*: *u*≠*t*   *u*≠*s* **and** *UIV*: *u*∈*V*
  shows *discharge f l n u*
   ≤ *SPEC* (λ(*f′*,*l′*,*n′*). *discharge-invar c s t u f l n f′ l′ n′*
        ∧ *excess f′ u* = *0*)
  ⟨*proof*⟩

end — Network

5.4 Main Algorithm

We state the main algorithm and prove its termination and correctness

context *Network*
begin

Initially, all edges are unprocessed.

definition *rtf-init-n u* ≡ *if u*∈*V*−{*s*,*t*} *then adjacent-nodes u else* {}

lemma *rtf-init-n-finite*[*simp*, *intro*!]: *finite* (*rtf-init-n u*)
⟨*proof*⟩

lemma *init-no-adm-edges*[*simp*]: *adm-edges pp-init-f pp-init-l* = {}
⟨*proof*⟩

lemma *rtf-init-neighbor-invar*:
neighbor-invar c s t pp-init-f pp-init-l rtf-init-n
⟨*proof*⟩

definition *relabel-to-front* ≡ *do* {
let f = *pp-init-f*;
let l = *pp-init-l*;
let n = *rtf-init-n*;

let L-left=[];
L-right ← *spec l*. *distinct l* ∧ *set l* = *V* − {*s*,*t*};

(*f*,*l*,*n*,*L-left*,*L-right*) ← $while_T$
(λ(*f*,*l*,*n*,*L-left*,*L-right*). *L-right* ≠ [])
(λ(*f*,*l*,*n*,*L-left*,*L-right*). *do* {
let u = *hd L-right*;
assert (*u* ∈ *V*);
let old-lu = *l u*;

(*f*,*l*,*n*) ← *discharge f l n u*;

if (*l u* ≠ *old-lu*) *then do* {
— Move *u* to front of *l*, and restart scanning *L*
let (*L-left*,*L-right*) = ([*u*],*L-left* @ *tl L-right*);
return (*f*,*l*,*n*,*L-left*,*L-right*)
} *else do* {
— Goto next node in *l*
let (*L-left*,*L-right*) = (*L-left*@[*u*], *tl L-right*);
return (*f*,*l*,*n*,*L-left*,*L-right*)
}

}) (*f*,*l*,*n*,*L-left*,*L-right*);

assert (*neighbor-invar c s t f l n*);

return f
}

end — Network

Invariant for the main algorithm:

1. Nodes in the queue left of the current node are not active
2. The queue is a topological sort of the admissible network
3. All nodes except source and sink are on the queue

locale *rtf-invar* = *neighbor-invar* +
fixes *L-left L-right* :: *node list*
assumes *left-no-excess*: ∀ *u*∈*set* (*L-left*). *excess f u* = *0*
assumes *L-sorted*: *is-top-sorted* (*adm-edges f l*) (*L-left* @ *L-right*)
assumes *L-set*: *set L-left* ∪ *set L-right* = *V*−{*s*,*t*}
begin
lemma *rtf-is-nbr*: *neighbor-invar c s t f l n* ⟨*proof*⟩

lemma *L-distinct*: *distinct* (*L-left* @ *L-right*)
⟨*proof*⟩

lemma *terminated-imp-maxflow*:
assumes [*simp*]: *L-right* = []
shows *isMaxFlow f*
⟨*proof*⟩

end

context *Network* **begin**
lemma *rtf-init-invar*:
assumes *DIS*: *distinct L-left* **and** *L-set*: *set L-left* = *V*−{*s*,*t*}
shows *rtf-invar c s t pp-init-f pp-init-l rtf-init-n* [] *L-left*
⟨*proof*⟩

theorem *relabel-to-front-correct*:
relabel-to-front ≤ *SPEC isMaxFlow*
⟨*proof*⟩

end — Network

end

6 Tools for Implementing Push-Relabel Algorithms

theory *Prpu-Common-Impl*
imports
  Prpu-Common-Inst
  Flow-Networks.Network-Impl
  Flow-Networks.NetCheck
begin

6.1 Basic Operations

type-synonym *excess-impl* = *node* ⇒ *capacity-impl*

context *Network-Impl*
begin

6.1.1 Excess Map

Obtain an excess map with all nodes mapped to zero.

definition *x-init* :: *excess-impl nres* **where** *x-init* ≡ *return* (λ-. *0*)

Get the excess of a node.

definition *x-get* :: *excess-impl* ⇒ *node* ⇒ *capacity-impl nres*
  where *x-get x u* ≡ *do* {
    assert (*u*∈*V*);
    return (*x u*)
  }

Add a capacity to the excess of a node.

definition *x-add* :: *excess-impl* ⇒ *node* ⇒ *capacity-impl* ⇒ *excess-impl nres*
  where *x-add x u* Δ ≡ *do* {
    assert (*u*∈*V*);
    return (*x*(*u* := *x u* + Δ))
  }

6.1.2 Labeling

Obtain the initial labeling: All nodes are zero, except the source which is labeled by $|V|$. The exact cardinality of V is passed as a parameter.

definition *l-init* :: *nat* ⇒ (*node* ⇒ *nat*) *nres*
  where *l-init C* ≡ *return* ((λ-. *0*)(*s* := *C*))

Get the label of a node.

definition *l-get* :: (*node* ⇒ *nat*) ⇒ *node* ⇒ *nat nres*

```
  where l-get l u ≡ do {
    assert (u ∈ V);
    return (l u)
  }
```

Set the label of a node.

```
definition l-set :: (node ⇒ nat) ⇒ node ⇒ nat ⇒ (node ⇒ nat) nres
  where l-set l u a ≡ do {
    assert (u∈V);
    assert (a < 2∗card V);
    return (l(u := a))
  }
```

6.1.3 Label Frequency Counts for Gap Heuristics

Obtain the frequency counts for the initial labeling. Again, the cardinality of $|V|$, which is required to determine the label of the source node, is passed as an explicit parameter.

```
definition cnt-init :: nat ⇒ (nat ⇒ nat) nres
  where cnt-init C ≡ do {
    assert (C<2∗N);
    return ((λ-. 0)(0 := C − 1, C := 1))
  }
```

Get the count for a label value.

```
definition cnt-get :: (nat ⇒ nat) ⇒ nat ⇒ nat nres
  where cnt-get cnt lv ≡ do {
    assert (lv < 2∗N);
    return (cnt lv)
  }
```

Increment the count for a label value by one.

```
definition cnt-incr :: (nat ⇒ nat) ⇒ nat ⇒ (nat ⇒ nat) nres
  where cnt-incr cnt lv ≡ do {
    assert (lv < 2∗N);
    return (cnt ( lv := cnt lv + 1 ))
  }
```

Decrement the count for a label value by one.

```
definition cnt-decr :: (nat ⇒ nat) ⇒ nat ⇒ (nat ⇒ nat) nres
  where cnt-decr cnt lv ≡ do {
    assert (lv < 2∗N ∧ cnt lv > 0);
    return (cnt ( lv := cnt lv − 1 ))
  }
```

end — Network Implementation Locale

6.2 Refinements to Basic Operations

context *Network-Impl*
begin

In this section, we refine the algorithm to actually use the basic operations.

6.2.1 Explicit Computation of the Excess

definition *xf-rel* ≡ { ((*excess f*,*cf-of f*),*f*) | *f*. *True* }
lemma *xf-rel-RELATES*[*refine-dref-RELATES*]: *RELATES xf-rel*
⟨*proof*⟩

definition *pp-init-x*
≡ λ*u*. (*if u*=*s then* (∑(*u*,*v*)∈*outgoing s*. − *c*(*u*,*v*)) *else c* (*s*,*u*))

lemma *excess-pp-init-f*[*simp*]: *excess pp-init-f* = *pp-init-x*
⟨*proof*⟩

definition *pp-init-cf*
≡ λ(*u*,*v*). *if* (*v*=*s*) *then c* (*v*,*u*) *else if u*=*s then 0 else c* (*u*,*v*)
lemma *cf-of-pp-init-f*[*simp*]: *cf-of pp-init-f* = *pp-init-cf*
⟨*proof*⟩

lemma *pp-init-x-rel*: ((*pp-init-x*, *pp-init-cf*), *pp-init-f*) ∈ *xf-rel*
⟨*proof*⟩

6.2.2 Algorithm to Compute Initial Excess and Flow

```
definition pp-init-xcf2-aux ≡ do {
  let x=(λ-. 0);
  let cf=c;

  foreach (adjacent-nodes s) (λv (x,cf). do {
    assert ((s,v)∈E);
    assert (s ≠ v);
    let a = cf (s,v);
    assert (x v = 0);
    let x = x( s := x s − a, v := a );
    let cf = cf( (s,v) := 0, (v,s) := a);
    return (x,cf)
  }) (x,cf)
}
```

lemma *pp-init-xcf2-aux-spec*:
shows *pp-init-xcf2-aux* ≤ *SPEC* (λ(*x*,*cf*). *x*=*pp-init-x* ∧ *cf* = *pp-init-cf*)
⟨*proof*⟩
applyS (*auto intro*!: *sum.reindex-cong*[**where** *l*=*snd*] *intro*: *inj-onI*)

applyS (*metis* (*mono-tags*, *lifting*) *Compl-iff Graph.zero-cap-simp insertE mem-Collect-eq*)
⟨*proof*⟩

definition *pp-init-xcf2 am* ≡ *do* {
  x ← *x-init*;
  cf ← *cf-init*;

  assert (*s*∈*V*);
  adj ← *am-get am s*;
  nfoldli adj (λ-. *True*) (λ*v* (*x*,*cf*). *do* {
    assert ((*s*,*v*)∈*E*);
    assert (*s* ≠ *v*);
    a ← *cf-get cf* (*s*,*v*);
    x ← *x-add x s* (−*a*);
    x ← *x-add x v a*;
    cf ← *cf-set cf* (*s*,*v*) *0*;
    cf ← *cf-set cf* (*v*,*s*) *a*;
    return (*x*,*cf*)
  }) (*x*,*cf*)
}

lemma *pp-init-xcf2-refine-aux*:
  assumes *AM*: *is-adj-map am*
  shows *pp-init-xcf2 am* ≤ ⇓*Id* (*pp-init-xcf2-aux*)
  ⟨*proof*⟩

lemma *pp-init-xcf2-refine*[*refine2*]:
  assumes *AM*: *is-adj-map am*
  shows *pp-init-xcf2 am* ≤⇓*xf-rel* (*RETURN pp-init-f*)
  ⟨*proof*⟩

6.2.3 Computing the Minimal Adjacent Label

definition (**in** *Network*) *min-adj-label-aux cf l u* ≡ *do* {
  assert (*u*∈*V*);
  x ← *foreach* (*adjacent-nodes u*) (λ*v x*. *do* {
    assert ((*u*,*v*)∈*E*∪E^{-1});
    assert (*v*∈*V*);
    if (*cf* (*u*,*v*) ≠ *0*) *then*
      case x of
        None ⇒ *return* (*Some* (*l v*))
      | *Some xx* ⇒ *return* (*Some* (*min* (*l v*) (*xx*)))
    else
      return x
  }) *None*;

```
  assert (x≠None);
  return (the x)
}
```

lemma (**in** −) *set-filter-xform-aux*:
```
  { f x | x. ( x = a ∨ x∈S ∧ x∉it ) ∧ P x }
  = (if P a then {f a} else {}) ∪ {f x | x. x∈S−it ∧ P x}
  ⟨proof⟩
```

lemma (**in** *Labeling*) *min-adj-label-aux-spec*:
```
  assumes PRE: relabel-precond f l u
  shows min-adj-label-aux cf l u ≤ SPEC (λx. x = Min { l v | v. (u,v)∈cf.E })
⟨proof⟩
```

definition *min-adj-label am cf l u* ≡ *do* {
```
  assert (u∈V);
  adj ← am-get am u;
  x ← nfoldli adj (λ-. True) (λv x. do {
    assert ((u,v)∈ E ∪ E^-1);
    assert (v∈V);
    cfuv ← cf-get cf (u,v);
    if (cfuv ≠ 0) then do {
      lv ← l-get l v;
      case x of
        None ⇒ return (Some lv)
      | Some xx ⇒ return (Some (min lv xx))
    } else
      return x
  }) None;

  assert (x≠None);
  return (the x)
}
```

lemma *min-adj-label-refine*[*THEN order-trans*, *refine-vcg*]:
```
  assumes Height-Bounded-Labeling c s t f l
  assumes AM: (am,adjacent-nodes)∈nat-rel→⟨nat-rel⟩list-set-rel
  assumes PRE: relabel-precond f l u
  assumes [simp]: cf = cf-of f
  shows min-adj-label am cf l u ≤ SPEC (λx. x
        = Min { l v | v. (u,v)∈cfE-of f })
⟨proof⟩
```

6.2.4 Refinement of Relabel

Utilities to Implement Relabel Operations

definition *relabel2 am cf l u* ≡ *do* {
```
  assert (u∈V − {s,t});
```

```
  nl ← min-adj-label am cf l u;
  l ← l-set l u (nl+1);
  return l
}
```

lemma *relabel2-refine*[*refine*]:
  assumes ((*x*,*cf*),*f*)∈*xf-rel*
  assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*li*,*l*)∈*Id* (*ui*,*u*)∈*Id*
  shows *relabel2 am cf li ui* ≤ ⇓*Id* (*relabel f l u*)
⟨*proof*⟩

6.2.5 Refinement of Push

definition *push2-aux x cf* ≡ λ(*u*,*v*). *do* {
  assert ((*u*,*v*) ∈ *E* ∪ E^{-1});
  assert (*u* ≠ *v*);
  let Δ = *min* (*x u*) (*cf* (*u*,*v*));
  return ((*x*(*u* := *x u* − Δ, *v* := *x v* + Δ),*augment-edge-cf cf* (*u*,*v*) Δ))
}

lemma *push2-aux-refine*:
  ⟦((*x*,*cf*),*f*)∈*xf-rel*; (*ei*,*e*)∈$Id \times_r Id$⟧
    ⟹ *push2-aux x cf ei* ≤ ⇓*xf-rel* (*push f l e*)
  ⟨*proof*⟩

definition *push2 x cf* ≡ λ(*u*,*v*). *do* {
  assert ((*u*,*v*) ∈ *E* ∪ E^{-1});
  xu ← *x-get x u*;
  cfuv ← *cf-get cf* (*u*,*v*);
  cfvu ← *cf-get cf* (*v*,*u*);
  let Δ = *min xu cfuv*;
  x ← *x-add x u* (−Δ);
  x ← *x-add x v* Δ;

  cf ← *cf-set cf* (*u*,*v*) (*cfuv* − Δ);
  cf ← *cf-set cf* (*v*,*u*) (*cfvu* + Δ);

  return (*x*,*cf*)
}

lemma *push2-refine*[*refine*]:
  assumes ((*x*,*cf*),*f*)∈*xf-rel* (*ei*,*e*)∈$Id \times_r Id$
  shows *push2 x cf ei* ≤ ⇓*xf-rel* (*push f l e*)
⟨*proof*⟩

6.2.6 Adding frequency counters to labeling

definition *l-invar l* ≡ ∀ *v*. *l v* ≠ *0* ⟶ *v*∈*V*

definition *clc-invar* ≡ λ(*cnt*,*l*).
  (∀ *lv*. *cnt lv* = *card* { *u*∈*V* . *l u* = *lv* })
∧ (∀ *u*. *l u* < *2*∗*N*) ∧ *l-invar l*
definition *clc-rel* ≡ *br snd clc-invar*

definition *clc-init C* ≡ *do* {
  l ← *l-init C*;
  cnt ← *cnt-init C*;
  return (*cnt*,*l*)
}

definition *clc-get* ≡ λ(*cnt*,*l*) *u*. *l-get l u*
definition *clc-set* ≡ λ(*cnt*,*l*) *u a*. *do* {
  assert (*a*<*2*∗*N*);
  lu ← *l-get l u*;
  cnt ← *cnt-decr cnt lu*;
  l ← *l-set l u a*;
  lu ← *l-get l u*;
  cnt ← *cnt-incr cnt lu*;
  return (*cnt*,*l*)
}

definition *clc-has-gap* ≡ λ(*cnt*,*l*) *lu*. *do* {
  nlu ← *cnt-get cnt lu*;
  return (*nlu* = *0*)
}

lemma *cardV-le-N*: *card V* ≤ *N* ⟨*proof*⟩
lemma *N-not-Z*: *N* ≠ *0* ⟨*proof*⟩
lemma *N-ge-2*: *2*≤*N* ⟨*proof*⟩

lemma *clc-init-refine*[*refine*]:
  assumes [*simplified*,*simp*]: (*Ci*,*C*)∈*nat-rel*
  assumes [*simp*]: *C* = *card V*
  shows *clc-init Ci* ≤⇓*clc-rel* (*l-init C*)
⟨*proof*⟩

lemma *clc-get-refine*[*refine*]:
  ⟦ (*clc*,*l*)∈*clc-rel*; (*ui*,*u*)∈*nat-rel* ⟧ ⟹ *clc-get clc ui* ≤⇓*Id* (*l-get l u*)
  ⟨*proof*⟩

definition *l-get-rlx* :: (*node* ⇒ *nat*) ⇒ *node* ⇒ *nat nres*
  where *l-get-rlx l u* ≡ *do* {
    assert (*u* < *N*);
    return (*l u*)
  }
definition *clc-get-rlx* ≡ λ(*cnt*,*l*) *u*. *l-get-rlx l u*

lemma *clc-get-rlx-refine*[*refine*]:
⟦ (*clc*,*l*)∈*clc-rel*; (*ui*,*u*)∈*nat-rel* ⟧
⟹ *clc-get-rlx clc ui* ≤⇓*Id* (*l-get-rlx l u*)
⟨*proof*⟩

lemma *card-insert-disjointI*:
⟦ *finite Y*; *X* = *insert x Y*; *x*∉*Y* ⟧ ⟹ *card X* = *Suc* (*card Y*)
⟨*proof*⟩

lemma *clc-set-refine*[*refine*]:
⟦ (*clc*,*l*) ∈ *clc-rel*; (*ui*,*u*)∈*nat-rel*; (*ai*,*a*)∈*nat-rel* ⟧ ⟹
clc-set clc ui ai ≤⇓*clc-rel* (*l-set l u a*)
⟨*proof*⟩
applyS *auto*
applyS (*auto simp*: *simp*: *card-gt-0-iff*)

⟨*proof*⟩

lemma *clc-has-gap-correct*[*THEN order-trans*, *refine-vcg*]:
⟦(*clc*,*l*)∈*clc-rel*; *k*<*2*∗*N*⟧
⟹ *clc-has-gap clc k* ≤ (*spec r*. *r* ⟷ *gap-precond l k*)
⟨*proof*⟩

6.2.7 Refinement of Gap-Heuristics

Utilities to Implement Gap-Heuristics

definition *gap-aux C l k* ≡ *do* {
nfoldli [*0*..<*N*] (λ-. *True*) (λ*v l*. *do* {
lv ← *l-get-rlx l v*;
if (*k* < *lv* ∧ *lv* < *C*) *then do* {
assert (*C*+*1* < *2*∗*N*);
l ← *l-set l v* (*C*+*1*);
return l
} *else return l*
}) *l*
}

lemma *gap-effect-invar*[*simp*]: *l-invar l* ⟹ *l-invar* (*gap-effect l k*)
⟨*proof*⟩

lemma *relabel-effect-invar*[*simp*]: ⟦*l-invar l*; *u*∈*V*⟧ ⟹ *l-invar* (*relabel-effect f l u*)

⟨*proof*⟩

lemma *gap-aux-correct*[*THEN order-trans*, *refine-vcg*]:
⟦*l-invar l*; *C*=*card V*⟧ ⟹ *gap-aux C l k* ≤ *SPEC* (λ*r*. *r*=*gap-effect l k*)
⟨*proof*⟩

definition *gap2 C clc k* ≡ *do* {

nfoldli [*0..<N*] (λ-. *True*) (λ*v clc. do* {
  lv ← *clc-get-rlx clc v*;
  if (*k* < *lv* ∧ *lv* < *C*) *then do* {
    clc ← *clc-set clc v* (*C*+*1*);
    return clc
  } *else return clc*
}) *clc*
}

lemma *gap2-refine*[*refine*]:
  assumes [*simplified*,*simp*]: (*Ci*,*C*)∈*nat-rel*   (*ki*,*k*)∈*nat-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  shows *gap2 Ci clc ki* ≤⇓*clc-rel* (*gap-aux C l k*)
  ⟨*proof*⟩

definition *gap-relabel-aux C f l u* ≡ *do* {
  lu ← *l-get l u*;
  l ← *relabel f l u*;
  if gap-precond l lu then
    gap-aux C l lu
  else return l
}

lemma *gap-relabel-aux-refine*:
  assumes [*simp*]: *C* = *card V*   *l-invar l*
  shows *gap-relabel-aux C f l u* ≤ *gap-relabel f l u*
  ⟨*proof*⟩

definition *min-adj-label-clc am cf clc u* ≡ *case clc of* (-,*l*) ⇒ *min-adj-label am cf l u*

definition *clc-relabel2 am cf clc u* ≡ *do* {
  assert (*u*∈*V* − {*s*,*t*});
  nl ← *min-adj-label-clc am cf clc u*;
  clc ← *clc-set clc u* (*nl*+*1*);
  return clc
}

lemma *clc-relabel2-refine*[*refine*]:
  assumes *XF*: ((*x*,*cf*),*f*)∈*xf-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*ui*,*u*)∈*Id*
  shows *clc-relabel2 am cf clc ui* ≤ ⇓*clc-rel* (*relabel f l u*)
⟨*proof*⟩

definition *gap-relabel2 C am cf clc u* ≡ *do* {
  lu ← *clc-get clc u*;
  clc ← *clc-relabel2 am cf clc u*;
  has-gap ← *clc-has-gap clc lu*;
  if has-gap then gap2 C clc lu
  else
    RETURN clc
}

lemma *gap-relabel2-refine-aux*:
  assumes *XCF*: ((*x*, *cf*), *f*) ∈ *xf-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*Ci*,*C*)∈*Id*   (*ui*,*u*)∈*Id*
  shows *gap-relabel2 Ci am cf clc ui* ≤ ⇓*clc-rel* (*gap-relabel-aux C f l u*)
  ⟨*proof*⟩

lemma *gap-relabel2-refine*[*refine*]:
  assumes *XCF*: ((*x*, *cf*), *f*) ∈ *xf-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*ui*,*u*)∈*Id*
  assumes *CC*: *C* = *card V*
  shows *gap-relabel2 C am cf clc ui* ≤⇓*clc-rel* (*gap-relabel f l u*)
⟨*proof*⟩

6.3 Refinement to Efficient Data Structures

6.3.1 Registration of Abstract Operations

We register all abstract operations at once, auto-rewriting the capacity matrix type

context includes *Network-Impl-Sepref-Register*
begin
sepref-register *x-get x-add*

sepref-register *l-init l-get l-get-rlx l-set*

sepref-register *clc-init clc-get clc-set clc-has-gap clc-get-rlx*

sepref-register *cnt-init cnt-get cnt-incr cnt-decr*
sepref-register *gap2 min-adj-label min-adj-label-clc*

sepref-register *push2 relabel2 clc-relabel2 gap-relabel2*

sepref-register *pp-init-xcf2*

end — Anonymous Context

6.3.2 Excess by Array

definition *x-assn* ≡ *is-nf N* (*0*::*capacity-impl*)

lemma *x-init-hnr*[*sepref-fr-rules*]:
(*uncurry0* (*Array.new N 0*), *uncurry0 x-init*) ∈ *unit-assn*^{k} $\rightarrow_a$ *x-assn*
⟨*proof*⟩

lemma *x-get-hnr*[*sepref-fr-rules*]:
(*uncurry Array.nth*, *uncurry* (*PR-CONST x-get*))
∈ *x-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *cap-assn*
⟨*proof*⟩

definition (**in** −) *x-add-impl x u* Δ ≡ *do* {
xu ← *Array.nth x u*;
x ← *Array.upd u* (*xu*+Δ) *x*;
return x
}
lemma *x-add-hnr*[*sepref-fr-rules*]:
(*uncurry2 x-add-impl*, *uncurry2* (*PR-CONST x-add*))
∈ *x-assn*^{d} $*_a$ *node-assn*^{k} $*_a$ *cap-assn*^{k} $\rightarrow_a$ *x-assn*
⟨*proof*⟩

6.3.3 Labeling by Array

definition *l-assn* ≡ *is-nf N* (*0*::*nat*)
definition (**in** −) *l-init-impl N s cardV* ≡ *do* {
l ← *Array.new N* (*0*::*nat*);
l ← *Array.upd s cardV l*;
return l
}
lemma *l-init-hnr*[*sepref-fr-rules*]:
(*l-init-impl N s*, (*PR-CONST l-init*)) ∈ *nat-assn*^{k} $\rightarrow_a$ *l-assn*
⟨*proof*⟩

lemma *l-get-hnr*[*sepref-fr-rules*]:
(*uncurry Array.nth*, *uncurry* (*PR-CONST l-get*))
∈ *l-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
⟨*proof*⟩

lemma *l-get-rlx-hnr*[*sepref-fr-rules*]:
(*uncurry Array.nth*, *uncurry* (*PR-CONST l-get-rlx*))
∈ *l-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
⟨*proof*⟩

lemma *l-set-hnr*[*sepref-fr-rules*]:
(*uncurry2* (λ*a i x*. *Array.upd i x a*), *uncurry2* (*PR-CONST l-set*))
∈ *l-assn*^{d} $*_a$ *node-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *l-assn*
⟨*proof*⟩

6.3.4 Label Frequency by Array

definition *cnt-assn* (f::*node*⇒*nat*) *a*
  $\equiv \exists_A l.\ a \mapsto_a l * \uparrow(length\ l = 2*N \wedge (\forall i<2*N.\ l!i = f\ i) \wedge (\forall i \geq 2*N.\ f\ i = 0))$

definition (**in** −) *cnt-init-impl N C* ≡ *do* {
  a ← *Array.new* (*2*∗*N*) (*0*::*nat*);
  a ← *Array.upd 0* (*C*−*1*) *a*;
  a ← *Array.upd C 1 a*;
  return a
}

definition (**in** −) *cnt-incr-impl a k* ≡ *do* {
  freq ← *Array.nth a k*;
  a ← *Array.upd k* (*freq*+*1*) *a*;
  return a
}

definition (**in** −) *cnt-decr-impl a k* ≡ *do* {
  freq ← *Array.nth a k*;
  a ← *Array.upd k* (*freq*−*1*) *a*;
  return a
}

lemma *cnt-init-hnr*[*sepref-fr-rules*]: (*cnt-init-impl N*, *PR-CONST cnt-init*) ∈ $nat\text{-}assn^k \rightarrow_a cnt\text{-}assn$
  ⟨*proof*⟩

lemma *cnt-get-hnr*[*sepref-fr-rules*]: (*uncurry Array.nth*, *uncurry* (*PR-CONST cnt-get*)) $\in cnt\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
  ⟨*proof*⟩

lemma *cnt-incr-hnr*[*sepref-fr-rules*]: (*uncurry cnt-incr-impl*, *uncurry* (*PR-CONST cnt-incr*)) $\in cnt\text{-}assn^d *_a nat\text{-}assn^k \rightarrow_a cnt\text{-}assn$
  ⟨*proof*⟩

lemma *cnt-decr-hnr*[*sepref-fr-rules*]: (*uncurry cnt-decr-impl*, *uncurry* (*PR-CONST cnt-decr*)) $\in cnt\text{-}assn^d *_a nat\text{-}assn^k \rightarrow_a cnt\text{-}assn$
  ⟨*proof*⟩

6.3.5 Combined Frequency Count and Labeling

definition $clc\text{-}assn \equiv cnt\text{-}assn \times_a l\text{-}assn$

sepref-thm *clc-init-impl* **is** *PR-CONST clc-init* :: $nat\text{-}assn^k \rightarrow_a clc\text{-}assn$
  ⟨*proof*⟩
concrete-definition (**in** −) *clc-init-impl*
  uses *Network-Impl.clc-init-impl.refine-raw*

lemmas [*sepref-fr-rules*] = *clc-init-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *clc-get-impl* **is** *uncurry* (*PR-CONST clc-get*)
:: $clc\text{-}assn^k *_a node\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *clc-get-impl*
uses *Network-Impl.clc-get-impl.refine-raw* **is** (*uncurry ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *clc-get-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *clc-get-rlx-impl* **is** *uncurry* (*PR-CONST clc-get-rlx*)
:: $clc\text{-}assn^k *_a node\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *clc-get-rlx-impl*
uses *Network-Impl.clc-get-rlx-impl.refine-raw* **is** (*uncurry ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *clc-get-rlx-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *clc-set-impl* **is** *uncurry2* (*PR-CONST clc-set*)
:: $clc\text{-}assn^d *_a node\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a clc\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *clc-set-impl*
uses *Network-Impl.clc-set-impl.refine-raw* **is** (*uncurry2 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *clc-set-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *clc-has-gap-impl* **is** *uncurry* (*PR-CONST clc-has-gap*)
:: $clc\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a bool\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *clc-has-gap-impl*
uses *Network-Impl.clc-has-gap-impl.refine-raw* **is** (*uncurry ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *clc-has-gap-impl.refine*[*OF Network-Impl-axioms*]

6.3.6 Push

sepref-thm *push-impl* **is** *uncurry2* (*PR-CONST push2*)
:: $x\text{-}assn^d *_a cf\text{-}assn^d *_a edge\text{-}assn^k \rightarrow_a (x\text{-}assn \times_a cf\text{-}assn)$
⟨*proof*⟩
concrete-definition (**in** −) *push-impl*
uses *Network-Impl.push-impl.refine-raw* **is** (*uncurry2 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *push-impl.refine*[*OF Network-Impl-axioms*]

6.3.7 Relabel

sepref-thm *min-adj-label-impl* **is** *uncurry3* (*PR-CONST min-adj-label*)
:: $am\text{-}assn^k *_a cf\text{-}assn^k *_a l\text{-}assn^k *_a node\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *min-adj-label-impl*
uses *Network-Impl.min-adj-label-impl.refine-raw* **is** (*uncurry3 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *min-adj-label-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *relabel-impl* **is** *uncurry3* (*PR-CONST relabel2*)
:: $am\text{-}assn^k *_a cf\text{-}assn^k *_a l\text{-}assn^d *_a node\text{-}assn^k \rightarrow_a l\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *relabel-impl*
uses *Network-Impl.relabel-impl.refine-raw* **is** (*uncurry3 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *relabel-impl.refine*[*OF Network-Impl-axioms*]

6.3.8 Gap-Relabel

sepref-thm *gap-impl* **is** *uncurry2* (*PR-CONST gap2*)
:: $nat\text{-}assn^k *_a clc\text{-}assn^d *_a nat\text{-}assn^k \rightarrow_a clc\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *gap-impl*
uses *Network-Impl.gap-impl.refine-raw* **is** (*uncurry2 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *gap-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *min-adj-label-clc-impl* **is** *uncurry3* (*PR-CONST min-adj-label-clc*)
:: $am\text{-}assn^k *_a cf\text{-}assn^k *_a clc\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*proof*⟩

concrete-definition (**in** −) *min-adj-label-clc-impl*
uses *Network-Impl.min-adj-label-clc-impl.refine-raw* **is** (*uncurry3 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *min-adj-label-clc-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *clc-relabel-impl* **is** *uncurry3* (*PR-CONST clc-relabel2*)
:: $am\text{-}assn^k *_a cf\text{-}assn^k *_a clc\text{-}assn^d *_a node\text{-}assn^k \rightarrow_a clc\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *clc-relabel-impl*
uses *Network-Impl.clc-relabel-impl.refine-raw* **is** (*uncurry3 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *clc-relabel-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *gap-relabel-impl* **is** *uncurry4* (*PR-CONST gap-relabel2*)
:: $nat\text{-}assn^k *_a am\text{-}assn^k *_a cf\text{-}assn^k *_a clc\text{-}assn^d *_a node\text{-}assn^k$
$\rightarrow_a clc\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *gap-relabel-impl*
uses *Network-Impl.gap-relabel-impl.refine-raw* **is** (*uncurry4 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *gap-relabel-impl.refine*[*OF Network-Impl-axioms*]

6.3.9 Initialization

sepref-thm *pp-init-xcf2-impl* **is** (*PR-CONST pp-init-xcf2*)
:: $am\text{-}assn^k \rightarrow_a x\text{-}assn \times_a cf\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *pp-init-xcf2-impl*
uses *Network-Impl.pp-init-xcf2-impl.refine-raw* **is** (*?f*,-)∈-
lemmas [*sepref-fr-rules*] = *pp-init-xcf2-impl.refine*[*OF Network-Impl-axioms*]

end — Network Implementation Locale

end

7 Implementation of the FIFO Push/Relabel Algorithm

theory *Fifo-Push-Relabel-Impl*
imports
  Fifo-Push-Relabel
  Prpu-Common-Impl
begin

7.1 Basic Operations

context *Network-Impl*
begin

7.1.1 Queue

Obtain the empty queue.

definition *q-empty* :: *node list nres* **where**
  q-empty ≡ *return* []

Check whether a queue is empty.

definition *q-is-empty* :: *node list* ⇒ *bool nres* **where**
  q-is-empty Q ≡ *return* ($Q = []$)

Enqueue a node.

definition *q-enqueue* :: *node* ⇒ *node list* ⇒ *node list nres* **where**
  q-enqueue v Q ≡ *do* {
    assert ($v \in V$);
    return (Q@[v])
  }

Dequeue a node.

definition *q-dequeue* :: *node list* ⇒ (*node* × *node list*) *nres* **where**
  q-dequeue Q ≡ *do* {
    assert ($Q \neq []$);
    return (*hd Q*, *tl Q*)
  }

end — Network Implementation Locale

7.2 Refinements to Basic Operations

context *Network-Impl*
begin

In this section, we refine the algorithm to actually use the basic operations.

7.2.1 Refinement of Push

definition *fifo-push2-aux x cf Q* ≡ λ(*u*,*v*). *do* {
  assert ((*u*,*v*) ∈ *E* ∪ E^{-1});
  assert (*u* ≠ *v*);
  let Δ = *min* (*x u*) (*cf* (*u*,*v*));
  let Q = (*if v*≠*s* ∧ *v*≠*t* ∧ *x v* = *0 then Q*@[*v*] *else Q*);
  return ((*x*(*u* := *x u* − Δ, *v* := *x v* + Δ),*augment-edge-cf cf* (*u*,*v*) Δ),*Q*)
}

lemma *fifo-push2-aux-refine*:
  ⟦((*x*,*cf*),*f*)∈*xf-rel*; (*ei*,*e*)∈*Id*$\times_r$*Id*; (*Qi*,*Q*)∈*Id*⟧
    ⟹ *fifo-push2-aux x cf Qi ei* ≤ ⇓(*xf-rel* $\times_r$ *Id*) (*fifo-push f l Q e*)
  ⟨*proof*⟩

definition *fifo-push2 x cf Q* ≡ λ(*u*,*v*). *do* {
  assert ((*u*,*v*) ∈ *E* ∪ E^{-1});
  xu ← *x-get x u*;
  xv ← *x-get x v*;
  cfuv ← *cf-get cf* (*u*,*v*);
  cfvu ← *cf-get cf* (*v*,*u*);
  let Δ = *min xu cfuv*;
  x ← *x-add x u* (−Δ);
  x ← *x-add x v* Δ;

  cf ← *cf-set cf* (*u*,*v*) (*cfuv* − Δ);
  cf ← *cf-set cf* (*v*,*u*) (*cfvu* + Δ);

  if v≠*s* ∧ *v*≠*t* ∧ *xv* = *0 then do* {
    Q ← *q-enqueue v Q*;
    return ((*x*,*cf*),*Q*)
  } *else*
    return ((*x*,*cf*),*Q*)
}

lemma *fifo-push2-refine*[*refine*]:
  assumes ((*x*,*cf*),*f*)∈*xf-rel* (*ei*,*e*)∈*Id*$\times_r$*Id* (*Qi*,*Q*)∈*Id*
  shows *fifo-push2 x cf Qi ei* ≤ ⇓(*xf-rel* $\times_r$ *Id*) (*fifo-push f l Q e*)
⟨*proof*⟩

7.2.2 Refinement of Gap-Relabel

definition *fifo-gap-relabel-aux C f l Q u* ≡ *do* {
  Q ← *q-enqueue u Q*;
  lu ← *l-get l u*;
  l ← *relabel f l u*;
  if gap-precond l lu then do {
    l ← *gap-aux C l lu*;
    return (*l*,*Q*)
  } *else return* (*l*,*Q*)
}

lemma *fifo-gap-relabel-aux-refine*:
  assumes [*simp*]: *C* = *card V*   *l-invar l*
  shows *fifo-gap-relabel-aux C f l Q u* ≤ *fifo-gap-relabel f l Q u*
  ⟨*proof*⟩

definition *fifo-gap-relabel2 C am cf clc Q u* ≡ *do* {
  Q ← *q-enqueue u Q*;
  lu ← *clc-get clc u*;
  clc ← *clc-relabel2 am cf clc u*;
  has-gap ← *clc-has-gap clc lu*;
  if has-gap then do {
    clc ← *gap2 C clc lu*;
    RETURN (*clc*,*Q*)
  } *else*
    RETURN (*clc*,*Q*)
}

lemma *fifo-gap-relabel2-refine-aux*:
  assumes *XCF*: ((*x*, *cf*), *f*) ∈ *xf-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*Ci*,*C*)∈*Id*   (*Qi*,*Q*)∈*Id*   (*ui*,*u*)∈*Id*
  shows *fifo-gap-relabel2 Ci am cf clc Qi ui*
    ≤ ⇓(*clc-rel* $\times_r$ *Id*) (*fifo-gap-relabel-aux C f l Q u*)
  ⟨*proof*⟩

lemma *fifo-gap-relabel2-refine*[*refine*]:
  assumes *XCF*: ((*x*, *cf*), *f*) ∈ *xf-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*Qi*,*Q*)∈*Id*   (*ui*,*u*)∈*Id*
  assumes *CC*: *C* = *card V*
  shows *fifo-gap-relabel2 C am cf clc Qi ui*
    ≤⇓(*clc-rel* $\times_r$ *Id*) (*fifo-gap-relabel f l Q u*)
⟨*proof*⟩

7.2.3 Refinement of Discharge

context begin

Some lengthy, multi-step refinement of discharge, changing the iteration to iteration over adjacent nodes with filter, and showing that we can do the filter wrt. the current state, rather than the original state before the loop.

```
lemma am-nodes-as-filter:
  assumes is-adj-map am
  shows {v . (u,v)∈cfE-of f} = set (filter (λv. cf-of f (u,v) ≠ 0) (am u))
  ⟨proof⟩ lemma adjacent-nodes-iterate-refine1:
  fixes ff u f
  assumes AMR: (am,adjacent-nodes)∈Id → ⟨Id⟩list-set-rel
  assumes CR: ⋀s si. (si,s)∈Id ⟹ cci si ⟷ cc s
  assumes FR: ⋀v vi s si. ⟦(vi,v)∈Id; v∈V; (u,v)∈E∪E⁻¹; (si,s)∈Id⟧ ⟹
    ffi vi si ≤ ⇓Id (do {
                    if (cf-of f (u,v) ≠ 0) then ff v s else RETURN s
                  }) (is ⋀v vi s si. ⟦-;-;-;-⟧ ⟹ - ≤ ⇓- (?ff' v s))
  assumes S0R: (s0i,s0)∈Id
  assumes UR: (ui,u)∈Id
  shows nfoldli (am ui) cci ffi s0i
        ≤⇓Id (FOREACHc {v . (u,v)∈cfE-of f} cc ff s0)
⟨proof⟩ definition dis-loop-aux am f_0 l Q u ≡ do {
  assert (u∈V − {s,t});
  assert (distinct (am u));
  nfoldli (am u) (λ(f,l,Q). excess f u ≠ 0) (λv (f,l,Q). do {
    assert ((u,v)∈E∪E⁻¹ ∧ v∈V);
    if (cf-of f_0 (u,v) ≠ 0) then do {
      if (l u = l v + 1) then do {
        (f',Q) ← fifo-push f l Q (u,v);
        assert (∀ v'. v'≠v ⟶ cf-of f' (u,v') = cf-of f (u,v'));
        return (f',l,Q)
      } else return (f,l,Q)
    } else return (f,l,Q)
  }) (f_0,l,Q)
}

private definition fifo-discharge-aux am f_0 l Q ≡ do {
  (u,Q) ← q-dequeue Q;
  assert (u∈V ∧ u≠s ∧ u≠t);

  (f,l,Q) ← dis-loop-aux am f_0 l Q u;

  if excess f u ≠ 0 then do {
    (l,Q) ← fifo-gap-relabel f l Q u;
    return (f,l,Q)
  } else do {
    return (f,l,Q)
  }
```

}

private lemma *fifo-discharge-aux-refine*:
  assumes *AM*: (*am*,*adjacent-nodes*)∈*Id* → ⟨*Id*⟩*list-set-rel*
  assumes [*simplified*,*simp*]: (*fi*,*f*)∈*Id* (*li*,*l*)∈*Id* (*Qi*,*Q*)∈*Id*
  shows *fifo-discharge-aux am fi li Qi* ≤ ⇓*Id* (*fifo-discharge f l Q*)
  ⟨*proof*⟩ **definition** *dis-loop-aux2 am* f_0 *l Q u* ≡ *do* {
  assert (*u*∈*V* − {*s*,*t*});
  assert (*distinct* (*am u*));
  nfoldli (*am u*) (λ(*f*,*l*,*Q*). *excess f u* ≠ *0*) (λ*v* (*f*,*l*,*Q*). *do* {
    assert ((*u*,*v*)∈*E*∪E^{-1} ∧ *v*∈*V*);
    if (*cf-of f* (*u*,*v*) ≠ *0*) *then do* {
      if (*l u* = *l v* + *1*) *then do* {
        (*f'*,*Q*) ← *fifo-push f l Q* (*u*,*v*);
        assert (∀ *v'*. *v'*≠*v* ⟶ *cf-of f'* (*u*,*v'*) = *cf-of f* (*u*,*v'*));
        return (*f'*,*l*,*Q*)
      } *else return* (*f*,*l*,*Q*)
    } *else return* (*f*,*l*,*Q*)
  }) (f_0,*l*,*Q*)
}

private lemma *dis-loop-aux2-refine*:
  shows *dis-loop-aux2 am* f_0 *l Q u* ≤⇓*Id* (*dis-loop-aux am* f_0 *l Q u*)
  ⟨*proof*⟩ **definition** *dis-loop-aux3 am x cf l Q u* ≡ *do* {
  assert (*u*∈*V* ∧ *distinct* (*am u*));
  monadic-nfoldli (*am u*)
    (λ((*x*,*cf*),*l*,*Q*). *do* { *xu* ← *x-get x u*; *return* (*xu* ≠ *0*) })
    (λ*v* ((*x*,*cf*),*l*,*Q*). *do* {
      cfuv ← *cf-get cf* (*u*,*v*);
      if (*cfuv* ≠ *0*) *then do* {
        lu ← *l-get l u*;
        lv ← *l-get l v*;
        if (*lu* = *lv* + *1*) *then do* {
          ((*x*,*cf*),*Q*) ← *fifo-push2 x cf Q* (*u*,*v*);
          return ((*x*,*cf*),*l*,*Q*)
        } *else return* ((*x*,*cf*),*l*,*Q*)
      } *else return* ((*x*,*cf*),*l*,*Q*)
  }) ((*x*,*cf*),*l*,*Q*)
}

private lemma *dis-loop-aux3-refine*:
  assumes [*simplified*,*simp*]: (*ami*,*am*)∈*Id* (*li*,*l*)∈*Id* (*Qi*,*Q*)∈*Id* (*ui*,*u*)∈*Id*
  assumes *XF*: ((*x*,*cf*),*f*)∈*xf-rel*
  shows *dis-loop-aux3 ami x cf li Qi ui*
    ≤⇓(*xf-rel* $\times_r$ *Id* $\times_r$ *Id*) (*dis-loop-aux2 am f l Q u*)
  ⟨*proof*⟩

definition *dis-loop2 am x cf clc Q u* ≡ *do* {
  assert (*distinct* (*am u*));

```
  amu ← am-get am u;
  monadic-nfoldli amu
    (λ((x,cf),clc,Q). do { xu ← x-get x u; return (xu ≠ 0) })
    (λv ((x,cf),clc,Q). do {
      cfuv ← cf-get cf (u,v);
      if (cfuv ≠ 0) then do {
        lu ← clc-get clc u;
        lv ← clc-get clc v;
        if (lu = lv + 1) then do {
          ((x,cf),Q) ← fifo-push2 x cf Q (u,v);
          return ((x,cf),clc,Q)
        } else return ((x,cf),clc,Q)
      } else return ((x,cf),clc,Q)
  }) ((x,cf),clc,Q)
}
```

private lemma *dis-loop2-refine-aux*:
  assumes [*simplified,simp*]: (*xi*,*x*)∈*Id*   (*cfi*,*cf*)∈*Id*   (*ami*,*am*)∈*Id*
  assumes [*simplified,simp*]: (*li*,*l*)∈*Id*   (*Qi*,*Q*)∈*Id*   (*ui*,*u*)∈*Id*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  shows *dis-loop2 ami xi cfi clc Qi ui*
    ≤⇓(*Id* $\times_r$ *clc-rel* $\times_r$ *Id*) (*dis-loop-aux3 am x cf l Q u*)
  ⟨*proof*⟩

lemma *dis-loop2-refine*[*refine*]:
  assumes *XF*: ((*x*,*cf*),*f*)∈*xf-rel*
  assumes *CLC*: (*clc*,*l*)∈*clc-rel*
  assumes [*simplified,simp*]: (*ami*,*am*)∈*Id*   (*Qi*,*Q*)∈*Id*   (*ui*,*u*)∈*Id*
  shows *dis-loop2 ami x cf clc Qi ui*
    ≤⇓(*xf-rel* $\times_r$ *clc-rel* $\times_r$ *Id*) (*dis-loop-aux am f l Q u*)
⟨*proof*⟩

definition *fifo-discharge2 C am x cf clc Q* ≡ *do* {

```
  (u,Q) ← q-dequeue Q;
  assert (u∈V ∧ u≠s ∧ u≠t);

  ((x,cf),clc,Q) ← dis-loop2 am x cf clc Q u;

  xu ← x-get x u;
  if xu ≠ 0 then do {
    (clc,Q) ← fifo-gap-relabel2 C am cf clc Q u;
    return ((x,cf),clc,Q)
  } else do {
    return ((x,cf),clc,Q)
  }
}
```

lemma *fifo-discharge2-refine*[*refine*]:

assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
assumes *XCF*: ((*x*, *cf*), *f*) ∈ *xf-rel*
assumes *CLC*: (*clc*,*l*)∈*clc-rel*
assumes [*simplified*,*simp*]: (*Qi*,*Q*)∈*Id*
assumes *CC*: *C* = *card V*
shows *fifo-discharge2 C am x cf clc Qi*
≤⇓(*xf-rel* $\times_r$ *clc-rel* $\times_r$ *Id*) (*fifo-discharge f l Q*)
⟨*proof*⟩
applyS *assumption*
⟨*proof*⟩

end — Anonymous Context

7.2.4 Computing the Initial Queue

definition *q-init am* ≡ *do* {
Q ← *q-empty*;
ams ← *am-get am s*;
nfoldli ams (λ-. *True*) (λ*v Q*. *do* {
if v≠*t then q-enqueue v Q else return Q*
}) *Q*
}

lemma *q-init-correct*[*THEN order-trans*, *refine-vcg*]:
assumes *AM*: *is-adj-map am*
shows *q-init am*
≤ (*spec l*. *distinct l* ∧ *set l* = {*v* ∈ *V* − {*s*, *t*}. *excess pp-init-f v* ≠ *0*})
⟨*proof*⟩

7.2.5 Refining the Main Algorithm

definition *fifo-push-relabel-aux am* ≡ *do* {
cardV ← *init-C am*;
assert (*cardV* = *card V*);
let f = *pp-init-f*;
l ← *l-init cardV*;

Q ← *q-init am*;

(*f*,*l*,-) ← *monadic-WHILEIT* (λ-. *True*)
(λ(*f*,*l*,*Q*). *do* {*qe* ← *q-is-empty Q*; *return* (¬*qe*)})
(λ(*f*,*l*,*Q*). *do* {
fifo-discharge f l Q
})
(*f*,*l*,*Q*);

assert (*Height-Bounded-Labeling c s t f l*);
return f
}

lemma *fifo-push-relabel-aux-refine*:
  assumes *AM*: *is-adj-map am*
  shows *fifo-push-relabel-aux am* ≤ ⇓*Id* (*fifo-push-relabel*)
  ⟨*proof*⟩

```
definition fifo-push-relabel2 am ≡ do {
  cardV ← init-C am;
  (x,cf) ← pp-init-xcf2 am;
  clc ← clc-init cardV;
  Q ← q-init am;

  ((x,cf),clc,Q) ← monadic-WHILEIT (λ-. True)
    (λ((x,cf),clc,Q). do {qe ← q-is-empty Q; return (¬qe)})
    (λ((x,cf),clc,Q). do {
      fifo-discharge2 cardV am x cf clc Q
    })
    ((x,cf),clc,Q);

  return cf
}
```

lemma *fifo-push-relabel2-refine*:
  assumes *AM*: *is-adj-map am*
  shows *fifo-push-relabel2 am*
    ≤ ⇓(*br* (*flow-of-cf*) (*RPreGraph c s t*)) *fifo-push-relabel*
⟨*proof*⟩

end — Network Impl. Locale

7.3 Separating out the Initialization of the Adjacency Matrix

context *Network-Impl*
begin

We split the algorithm into an initialization of the adjacency matrix, and the actual algorithm. This way, the algorithm can handle pre-initialized adjacency matrices.

definition *fifo-push-relabel-init2* ≡ *cf-init*

```
definition pp-init-xcf2' am cf ≡ do {
  x ← x-init;

  assert (s∈V);
  adj ← am-get am s;
  nfoldli adj (λ-. True) (λv (x,cf). do {
    assert ((s,v)∈E);
    assert (s ≠ v);
    a ← cf-get cf (s,v);
    x ← x-add x s (−a);
```

```
    x ← x-add x v a;
    cf ← cf-set cf (s,v) 0;
    cf ← cf-set cf (v,s) a;
    return (x,cf)
  }) (x,cf)
}
```

definition *fifo-push-relabel-run2 am cf* ≡ *do* {

```
  cardV ← init-C am;
  (x,cf) ← pp-init-xcf2' am cf;
  clc ← clc-init cardV;
  Q ← q-init am;

  ((x,cf),clc,Q) ← monadic-WHILEIT (λ-. True)
    (λ((x,cf),clc,Q). do {qe ← q-is-empty Q; return (¬qe)})
    (λ((x,cf),clc,Q). do {
      fifo-discharge2 cardV am x cf clc Q
    })
    ((x,cf),clc,Q);

  return cf
}
```

lemma *fifo-push-relabel2-alt*:

```
  fifo-push-relabel2 am = do {
    cf ← fifo-push-relabel-init2;
    fifo-push-relabel-run2 am cf
  }
  ⟨proof⟩
```

end — Network Impl. Locale

7.4 Refinement To Efficient Data Structures

context *Network-Impl*
begin

7.4.1 Registration of Abstract Operations

We register all abstract operations at once, auto-rewriting the capacity matrix type

context includes *Network-Impl-Sepref-Register*
begin

sepref-register *q-empty q-is-empty q-enqueue q-dequeue*

sepref-register *fifo-push2*

sepref-register *fifo-gap-relabel2*

sepref-register *dis-loop2 fifo-discharge2*

sepref-register *q-init pp-init-xcf2'*

sepref-register *fifo-push-relabel-run2 fifo-push-relabel-init2*
sepref-register *fifo-push-relabel2*

end — Anonymous Context

7.4.2 Queue by Two Stacks

definition (**in** −) $q\text{-}\alpha \equiv \lambda(L,R).\ L@rev\ R$
definition (**in** −) $q\text{-}empty\text{-}impl \equiv ([],[])$
definition (**in** −) $q\text{-}is\text{-}empty\text{-}impl \equiv \lambda(L,R).\ is\text{-}Nil\ L \wedge is\text{-}Nil\ R$
definition (**in** −) $q\text{-}enqueue\text{-}impl \equiv \lambda x\ (L,R).\ (L,x\#R)$
definition (**in** −) *q-dequeue-impl*
$\equiv \lambda(x\#L,R) \Rightarrow (x,(L,R))\ |\ ([],R) \Rightarrow case\ rev\ R\ of\ (x\#L) \Rightarrow (x,(L,[]))$

lemma *q-empty-impl-correct*[*simp*]: $q\text{-}\alpha\ q\text{-}empty\text{-}impl = []$
⟨*proof*⟩

lemma *q-enqueue-impl-correct*[*simp*]: $q\text{-}\alpha\ (q\text{-}enqueue\text{-}impl\ x\ Q) = q\text{-}\alpha\ Q\ @\ [x]$
⟨*proof*⟩

lemma *q-is-empty-impl-correct*[*simp*]: $q\text{-}is\text{-}empty\text{-}impl\ Q \longleftrightarrow q\text{-}\alpha\ Q = []$
⟨*proof*⟩

lemma *q-dequeue-impl-correct-aux*:
$[\![q\text{-}\alpha\ Q = x\#xs]\!] \Longrightarrow apsnd\ q\text{-}\alpha\ (q\text{-}dequeue\text{-}impl\ Q) = (x,xs)$
⟨*proof*⟩

lemma *q-dequeue-impl-correct*[*simp*]:
assumes $q\text{-}dequeue\text{-}impl\ Q = (x,Q')$
assumes $q\text{-}\alpha\ Q \neq []$
shows $x = hd\ (q\text{-}\alpha\ Q)$ **and** $q\text{-}\alpha\ Q' = tl\ (q\text{-}\alpha\ Q)$
⟨*proof*⟩

definition $q\text{-}assn \equiv pure\ (br\ q\text{-}\alpha\ (\lambda\text{-}.\ True))$

lemma *q-empty-impl-hnr*[*sepref-fr-rules*]:
$(uncurry0\ (return\ q\text{-}empty\text{-}impl),\ uncurry0\ q\text{-}empty) \in unit\text{-}assn^k \rightarrow_a q\text{-}assn$
⟨*proof*⟩

lemma *q-is-empty-impl-hnr*[*sepref-fr-rules*]:
(*return o q-is-empty-impl*, *q-is-empty*) ∈ *q-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

lemma *q-enqueue-impl-hnr*[*sepref-fr-rules*]:
(*uncurry* (*return oo q-enqueue-impl*), *uncurry* (*PR-CONST q-enqueue*))
∈ *nat-assn*^{k} $*_a$ *q-assn*^{d} $\rightarrow_a$ *q-assn*
⟨*proof*⟩

lemma *q-dequeue-impl-hnr*[*sepref-fr-rules*]:
(*return o q-dequeue-impl*, *q-dequeue*) ∈ *q-assn*^{d} $\rightarrow_a$ *nat-assn* $\times_a$ *q-assn*
⟨*proof*⟩

7.4.3 Push

sepref-thm *fifo-push-impl* **is** *uncurry3* (*PR-CONST fifo-push2*)
:: *x-assn*^{d} $*_a$ *cf-assn*^{d} $*_a$ *q-assn*^{d} $*_a$ *edge-assn*^{k}
$\rightarrow_a$ ((*x-assn*$\times_a$*cf-assn*)$\times_a$*q-assn*)
⟨*proof*⟩
concrete-definition (**in** −) *fifo-push-impl*
uses *Network-Impl.fifo-push-impl.refine-raw* **is** (*uncurry3 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *fifo-push-impl.refine*[*OF Network-Impl-axioms*]

7.4.4 Gap-Relabel

sepref-thm *fifo-gap-relabel-impl* **is** *uncurry5* (*PR-CONST fifo-gap-relabel2*)
:: *nat-assn*^{k} $*_a$ *am-assn*^{k} $*_a$ *cf-assn*^{k} $*_a$ *clc-assn*^{d} $*_a$ *q-assn*^{d} $*_a$ *node-assn*^{k}
$\rightarrow_a$ *clc-assn* $\times_a$ *q-assn*
⟨*proof*⟩
concrete-definition (**in** −) *fifo-gap-relabel-impl*
uses *Network-Impl.fifo-gap-relabel-impl.refine-raw* **is** (*uncurry5 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *fifo-gap-relabel-impl.refine*[*OF Network-Impl-axioms*]

7.4.5 Discharge

sepref-thm *fifo-dis-loop-impl* **is** *uncurry5* (*PR-CONST dis-loop2*)
:: *am-assn*^{k} $*_a$ *x-assn*^{d} $*_a$ *cf-assn*^{d} $*_a$ *clc-assn*^{d} $*_a$ *q-assn*^{d} $*_a$ *node-assn*^{k}
$\rightarrow_a$ (*x-assn*$\times_a$*cf-assn*)$\times_a$*clc-assn* $\times_a$ *q-assn*
⟨*proof*⟩
concrete-definition (**in** −) *fifo-dis-loop-impl*
uses *Network-Impl.fifo-dis-loop-impl.refine-raw* **is** (*uncurry5 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *fifo-dis-loop-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *fifo-fifo-discharge-impl* **is** *uncurry5* (*PR-CONST fifo-discharge2*)
:: *nat-assn*^{k} $*_a$ *am-assn*^{k} $*_a$ *x-assn*^{d} $*_a$ *cf-assn*^{d} $*_a$ *clc-assn*^{d} $*_a$ *q-assn*^{d}
$\rightarrow_a$ (*x-assn*$\times_a$*cf-assn*)$\times_a$*clc-assn* $\times_a$ *q-assn*
⟨*proof*⟩
concrete-definition (**in** −) *fifo-fifo-discharge-impl*
uses *Network-Impl.fifo-fifo-discharge-impl.refine-raw* **is** (*uncurry5 ?f*,-)∈-
lemmas [*sepref-fr-rules*] =

fifo-fifo-discharge-impl.refine[*OF Network-Impl-axioms*]

7.4.6 Computing the Initial State

sepref-thm *fifo-init-C-impl* **is** (*PR-CONST init-C*)
:: $am\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *fifo-init-C-impl*
uses *Network-Impl.fifo-init-C-impl.refine-raw* **is** (*?f*,-)∈-
lemmas [*sepref-fr-rules*] = *fifo-init-C-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *fifo-q-init-impl* **is** (*PR-CONST q-init*)
:: $am\text{-}assn^k \rightarrow_a q\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *fifo-q-init-impl*
uses *Network-Impl.fifo-q-init-impl.refine-raw* **is** (*?f*,-)∈-
lemmas [*sepref-fr-rules*] = *fifo-q-init-impl.refine*[*OF Network-Impl-axioms*]

sepref-thm *pp-init-xcf2'-impl* **is** *uncurry* (*PR-CONST pp-init-xcf2'*)
:: $am\text{-}assn^k *_a cf\text{-}assn^d \rightarrow_a x\text{-}assn \times_a cf\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *pp-init-xcf2'-impl*
uses *Network-Impl.pp-init-xcf2'-impl.refine-raw* **is** (*uncurry ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *pp-init-xcf2'-impl.refine*[*OF Network-Impl-axioms*]

7.4.7 Main Algorithm

sepref-thm *fifo-push-relabel-run-impl*
is *uncurry* (*PR-CONST fifo-push-relabel-run2*)
:: $am\text{-}assn^k *_a cf\text{-}assn^d \rightarrow_a cf\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *fifo-push-relabel-run-impl*
uses *Network-Impl.fifo-push-relabel-run-impl.refine-raw* **is** (*uncurry ?f*,-)∈-
lemmas [*sepref-fr-rules*] =
fifo-push-relabel-run-impl.refine[*OF Network-Impl-axioms*]

sepref-thm *fifo-push-relabel-init-impl*
is *uncurry0* (*PR-CONST fifo-push-relabel-init2*)
:: $unit\text{-}assn^k \rightarrow_a cf\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *fifo-push-relabel-init-impl*
uses *Network-Impl.fifo-push-relabel-init-impl.refine-raw*
is (*uncurry0 ?f*,-)∈-
lemmas [*sepref-fr-rules*] =
fifo-push-relabel-init-impl.refine[*OF Network-Impl-axioms*]

sepref-thm *fifo-push-relabel-impl* **is** (*PR-CONST fifo-push-relabel2*)
:: $am\text{-}assn^k \rightarrow_a cf\text{-}assn$
⟨*proof*⟩

concrete-definition (**in** −) *fifo-push-relabel-impl*
  uses *Network-Impl.fifo-push-relabel-impl.refine-raw* **is** (*?f*,-)∈-
lemmas [*sepref-fr-rules*] = *fifo-push-relabel-impl.refine*[*OF Network-Impl-axioms*]

end — Network Impl. Locale

export-code *fifo-push-relabel-impl* **checking** *SML-imp*

7.5 Combining the Refinement Steps

theorem (**in** *Network-Impl*) *fifo-push-relabel-impl-correct*[*sep-heap-rules*]:
  assumes *AM*: *is-adj-map am*
  shows
    <*am-assn am ami*>
      fifo-push-relabel-impl c s t N ami
    <λ*cfi*. $\exists_A$*cf*.
        am-assn am ami ∗ *cf-assn cf cfi*
      ∗ ↑(*isMaxFlow* (*flow-of-cf cf*) ∧ *RGraph-Impl c s t N cf*)>$_t$
⟨*proof*⟩

7.6 Combination with Network Checker and Main Correctness Theorem

definition *fifo-push-relabel-impl-tab-am c s t N am* ≡ *do* {
  ami ← *Array.make N am*; — TODO/DUP: Called *init-ps* in Edmonds-Karp impl
  cfi ← *fifo-push-relabel-impl c s t N ami*;
  return (*ami*,*cfi*)
}

theorem *fifo-push-relabel-impl-tab-am-correct*[*sep-heap-rules*]:
  assumes *NW*: *Network c s t*
  assumes *VN*: *Graph.V c* ⊆ {*0*..<*N*}
  assumes *ABS-PS*: *Graph.is-adj-map c am*
  shows
    <*emp*>
      fifo-push-relabel-impl-tab-am c s t N am
    <λ(*ami*,*cfi*). $\exists_A$*cf*.
        am-assn N am ami ∗ *cf-assn N cf cfi*
      ∗ ↑(*Network.isMaxFlow c s t* (*Network.flow-of-cf c cf*)
        ∧ *RGraph-Impl c s t N cf*
)>$_t$
⟨*proof*⟩

definition *fifo-push-relabel el s t* ≡ *do* {
  case prepareNet el s t of

```
    None ⇒ return None
  | Some (c,am,N) ⇒ do {
      (ami,cf) ← fifo-push-relabel-impl-tab-am c s t N am;
      return (Some (c,ami,N,cf))
  }
}
```

export-code *fifo-push-relabel* **checking** *SML-imp*

Main correctness statement:

- If *fifo-push-relabel* returns *None*, the edge list was invalid or described an invalid network.
- If it returns *Some* (*c*,*am*,*N*,*cfi*), then the edge list is valid and describes a valid network. Moreover, *cfi* is an integer square matrix of dimension *N*, which describes a valid residual graph in the network, whose corresponding flow is maximal. Finally, *am* is a valid adjacency map of the graph, and the nodes of the graph are integers less than *N*.

theorem *fifo-push-relabel-correct*[*sep-heap-rules*]:

```
  <emp>
  fifo-push-relabel el s t
  <λ
    None ⇒ ↑(¬ln-invar el ∨ ¬Network (ln-α el) s t)
  | Some (c,ami,N,cfi) ⇒
      ↑(c = ln-α el ∧ ln-invar el ∧ Network c s t)
    ∗ (∃A am cf. am-assn N am ami ∗ cf-assn N cf cfi
         ∗ ↑(RGraph-Impl c s t N cf ∧ Graph.is-adj-map c am
           ∧ Network.isMaxFlow c s t (Network.flow-of-cf c cf))
      )
  >t

  ⟨proof⟩
```

7.6.1 Justification of Splitting into Prepare and Run Phase

definition *fifo-push-relabel-prepare-impl el s t* ≡ *do* {

```
  case prepareNet el s t of
    None ⇒ return None
  | Some (c,am,N) ⇒ do {
      ami ← Array.make N am;
      cfi ← fifo-push-relabel-init-impl c N;
      return (Some (N,ami,c,cfi))
    }
}
```

theorem *justify-fifo-push-relabel-prep-run-split*:

```
  fifo-push-relabel el s t =
  do {
```

```
  pr ← fifo-push-relabel-prepare-impl el s t;
  case pr of
    None ⇒ return None
  | Some (N,ami,c,cf) ⇒ do {
      cf ← fifo-push-relabel-run-impl s t N ami cf;
      return (Some (c,ami,N,cf))
    }
}
⟨proof⟩
```

7.7 Usage Example: Computing Maxflow Value

We implement a function to compute the value of the maximum flow.

```
definition fifo-push-relabel-compute-flow-val el s t ≡ do {
  r ← fifo-push-relabel el s t;
  case r of
    None ⇒ return None
  | Some (c,am,N,cf) ⇒ do {
      v ← compute-flow-val-impl s N am cf;
      return (Some v)
    }
}
```

The computed flow value is correct

```
theorem fifo-push-relabel-compute-flow-val-correct:
  <emp>
    fifo-push-relabel-compute-flow-val el s t
  <λ
    None ⇒ ↑(¬ln-invar el ∨ ¬Network (ln-α el) s t)
  | Some v ⇒ ↑( ln-invar el
      ∧ (let c = ln-α el in
          Network c s t ∧ Network.is-max-flow-val c s t v
        ))
  >t
⟨proof⟩

export-code fifo-push-relabel-compute-flow-val checking SML-imp


end
```

8 Implementation of Relabel-to-Front

```
theory Relabel-To-Front-Impl
imports
  Relabel-To-Front
  Prpu-Common-Impl
begin
```

8.1 Basic Operations

context *Network-Impl*
begin

8.1.1 Neighbor Lists

definition *n-init* :: (*node* ⇒ *node list*) ⇒ (*node* ⇒ *node list*) *nres*
  where *n-init am* ≡ *return* (*am*(*s* := [], *t* := []))

definition *n-at-end* :: (*node* ⇒ *node list*) ⇒ *node* ⇒ *bool nres*
  where *n-at-end n u* ≡ *do* {
    assert (*u*∈*V*−{*s*,*t*});
    return (*n u* = [])
  }

definition *n-get-hd* :: (*node* ⇒ *node list*) ⇒ *node* ⇒ *node nres*
  where *n-get-hd n u* ≡ *do* {
    assert (*u*∈*V*−{*s*,*t*} ∧ *n u* ≠ []);
    return (*hd* (*n u*))
  }

definition *n-move-next*
  :: (*node* ⇒ *node list*) ⇒ *node* ⇒ (*node* ⇒ *node list*) *nres*
  where *n-move-next n u* ≡ *do* {
    assert (*u*∈*V*−{*s*,*t*} ∧ *n u* ≠ []);
    return (*n* (*u* := *tl* (*n u*)))
  }

definition *n-reset*
  :: (*node* ⇒ *node list*) ⇒ (*node* ⇒ *node list*) ⇒ *node*
    ⇒ (*node* ⇒ *node list*) *nres*
  where *n-reset am n u* ≡ *do* {
    assert (*u*∈*V*−{*s*,*t*});
    return (*n* (*u* := *am u*))
  }

lemma *n-init-refine*[*refine2*]:
  assumes *AM*: *is-adj-map am*
  shows *n-init am*
    ≤ (*spec c*. (*c*, *rtf-init-n*) ∈ (*nat-rel* → ⟨*nat-rel*⟩*list-set-rel*))
⟨*proof*⟩

8.2 Refinement to Basic Operations

8.2.1 Discharge

definition *discharge2 am x cf l n u* ≡ *do* {
  assert (*u* ∈ *V*);
  monadic-WHILEIT (λ-. *True*)

```
    (λ((x,cf),l,n). do { xu ← x-get x u; return (xu ≠ 0) } )
    (λ((x,cf),l,n). do {
      at-end ← n-at-end n u;
      if at-end then do {
        l ← relabel2 am cf l u;
        n ← n-reset am n u;
        return ((x,cf),l,n)
      } else do {
        v ← n-get-hd n u;
        cfuv ← cf-get cf (u,v);
        lu ← l-get l u;
        lv ← l-get l v;
        if (cfuv ≠ 0 ∧ lu = lv + 1) then do {
          (x,cf) ← push2 x cf (u,v);
          return ((x,cf),l,n)
        } else do {
          n ← n-move-next n u;
          return ((x,cf),l,n)
        }
      }
    }) ((x,cf),l,n)
}
```

```
lemma discharge-structure-refine-aux:
  assumes SR: (ni,n)∈nat-rel → ⟨nat-rel⟩list-set-rel
  assumes SU: (ui,u)∈Id
  assumes fNR: fNi ≤ ⇓R fN
  assumes UIV: u∈V−{s,t}
  assumes fSR: ⋀v vi vs. ⟦
      (vi,v)∈Id; v∈n u; ni u = v#vs; (v#vs,n u)∈⟨nat-rel⟩list-set-rel
    ⟧ ⟹ fSi vi ≤ ⇓R (fS v)
  shows
  ( do {
    at-end ← n-at-end ni ui;
    if at-end then fNi
    else do {
      v ← n-get-hd ni ui;
      fSi v
    }
  } ) ≤ ⇓R (

  do {
    v ← select v. v∈n u;
    case v of
      None ⇒ fN
    | Some v ⇒ fS v
  }) (is ?lhs ≤⇓R ?rhs)
  ⟨proof⟩
```

lemma *xf-rel-RELATES*[*refine-dref-RELATES*]: *RELATES xf-rel*
⟨*proof*⟩

lemma *discharge2-refine*[*refine*]:
assumes *A*: ((*x*,*cf*),*f*) ∈ *xf-rel*
assumes *AM*: (*am*,*adjacent-nodes*)∈*nat-rel*→⟨*nat-rel*⟩*list-set-rel*
assumes [*simplified*,*simp*]: (*li*,*l*)∈*Id* (*ui*,*u*)∈*Id*
assumes *NR*: (*ni*,*n*)∈*nat-rel* → ⟨*nat-rel*⟩*list-set-rel*
shows *discharge2 am x cf li ni ui*
≤ ⇓(*xf-rel* $\times_r$ *Id* $\times_r$ (*nat-rel* → ⟨*nat-rel*⟩*list-set-rel*)) (*discharge f l n u*)
⟨*proof*⟩

8.2.2 Initialization of Queue

lemma *V-is-adj-nodes*: *V* = { *v* . *adjacent-nodes v* ≠ {} }
⟨*proof*⟩

definition *init-CQ am* ≡ *do* {
let cardV=*0*;
let Q=[];
nfoldli [*0*..<*N*] (λ-. *True*) (λ*v* (*cardV*,*Q*). *do* {
assert (*v*<*N*);
inV ← *am-is-in-V am v*;
if inV then do {
let cardV = *cardV* + *1*;
if v≠*s* ∧ *v*≠*t then*
return (*cardV*,*v*#*Q*)
else
return (*cardV*,*Q*)
} *else*
return (*cardV*,*Q*)
}) (*cardV*,*Q*)
}

lemma *init-CQ-correct*[*THEN order-trans*, *refine-vcg*]:
assumes *AM*: *is-adj-map am*
shows *init-CQ am* ≤ *SPEC* (λ(*C*,*Q*). *C* = *card V* ∧ *distinct Q* ∧ *set Q* = *V*−{*s*,*t*})
⟨*proof*⟩

8.2.3 Main Algorithm

definition *relabel-to-front2 am* ≡ *do* {
(*cardV*, *L-right*) ← *init-CQ am*;

xcf ← *pp-init-xcf2 am*;
l ← *l-init cardV*;
n ← *n-init am*;

let L-left=[];

((*x*,*cf*),*l*,*n*,*L-left*,*L-right*) ← $while_T$
  (λ((*x*,*cf*),*l*,*n*,*L-left*,*L-right*). *L-right* ≠ [])
  (λ((*x*,*cf*),*l*,*n*,*L-left*,*L-right*). *do* {
    assert (*L-right* ≠ []);
    let u = *hd L-right*;
    old-lu ← *l-get l u*;

    ((*x*,*cf*),*l*,*n*) ← *discharge2 am x cf l n u*;

    lu ← *l-get l u*;
    if (*lu* ≠ *old-lu*) *then do* {
      — Move *u* to front of *l*, and restart scanning *L*. The cost for
      — *rev-append* is amortized by going to next node in *L*
      let (*L-left*,*L-right*) = ([*u*],*rev-append L-left* (*tl L-right*));
      return ((*x*,*cf*),*l*,*n*,*L-left*,*L-right*)
    } *else do* {
      — Goto next node in *L*
      let (*L-left*,*L-right*) = (*u*#*L-left*, *tl L-right*);
      return ((*x*,*cf*),*l*,*n*,*L-left*,*L-right*)
    }

  }) (*xcf*,*l*,*n*,*L-left*,*L-right*);

  return cf
}

lemma *relabel-to-front2-refine*[*refine*]:
  assumes *AM*: *is-adj-map am*
  shows *relabel-to-front2 am*
    ≤ ⇓(*br* (*flow-of-cf*) (*RPreGraph c s t*)) *relabel-to-front*
⟨*proof*⟩

8.3 Refinement to Efficient Data Structures

context includes *Network-Impl-Sepref-Register*
begin
  sepref-register *n-init*
  sepref-register *n-at-end*
  sepref-register *n-get-hd*
  sepref-register *n-move-next*
  sepref-register *n-reset*
  sepref-register *discharge2*
  sepref-register *init-CQ*
  sepref-register *relabel-to-front2*
end

8.3.1 Neighbor Lists by Array of Lists

definition *n-assn* ≡ *is-nf N* ([]::*nat list*)

definition (**in** −) *n-init-impl s t am* ≡ *do* {
  n ← *array-copy am*;
  n ← *Array.upd s* [] *n*;
  n ← *Array.upd t* [] *n*;
  return n
}

lemma [*sepref-fr-rules*]:
  (*n-init-impl s t*,*PR-CONST n-init*) ∈ $am\text{-}assn^k \rightarrow_a n\text{-}assn$
  ⟨*proof*⟩

definition (**in** −) *n-at-end-impl n u* ≡ *do* {
  nu ← *Array.nth n u*;
  return (*is-Nil nu*)
}

lemma [*sepref-fr-rules*]:
  (*uncurry n-at-end-impl*, *uncurry* (*PR-CONST n-at-end*))
  $\in n\text{-}assn^k *_a node\text{-}assn^k \rightarrow_a bool\text{-}assn$
  ⟨*proof*⟩

definition (**in** −) *n-get-hd-impl n u* ≡ *do* {
  nu ← *Array.nth n u*;
  return (*hd nu*)
}
lemma [*sepref-fr-rules*]:
  (*uncurry n-get-hd-impl*, *uncurry* (*PR-CONST n-get-hd*))
  $\in n\text{-}assn^k *_a node\text{-}assn^k \rightarrow_a node\text{-}assn$
  ⟨*proof*⟩

definition (**in** −) *n-move-next-impl n u* ≡ *do* {
  nu ← *Array.nth n u*;
  n ← *Array.upd u* (*tl nu*) *n*;
  return n
}
lemma [*sepref-fr-rules*]:
  (*uncurry n-move-next-impl*, *uncurry* (*PR-CONST n-move-next*))
  $\in n\text{-}assn^d *_a node\text{-}assn^k \rightarrow_a n\text{-}assn$
  ⟨*proof*⟩

definition (**in** −) *n-reset-impl am n u* ≡ *do* {
  nu ← *Array.nth am u*;
  n ← *Array.upd u nu n*;
  return n
}
lemma [*sepref-fr-rules*]:

$(uncurry2\ n\text{-}reset\text{-}impl,\ uncurry2\ (PR\text{-}CONST\ n\text{-}reset))$
$\in am\text{-}assn^k *_a n\text{-}assn^d *_a node\text{-}assn^k \rightarrow_a n\text{-}assn$
⟨*proof*⟩

8.3.2 Discharge

sepref-thm *discharge-impl* **is** $uncurry5\ (PR\text{-}CONST\ discharge2)$
$:: am\text{-}assn^k *_a x\text{-}assn^d *_a cf\text{-}assn^d *_a l\text{-}assn^d *_a n\text{-}assn^d *_a node\text{-}assn^k$
$\rightarrow_a (x\text{-}assn \times_a cf\text{-}assn) \times_a l\text{-}assn \times_a n\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *discharge-impl*
uses *Network-Impl.discharge-impl.refine-raw* **is** (*uncurry5 ?f*,-)∈-
lemmas [*sepref-fr-rules*] = *discharge-impl.refine*[*OF Network-Impl-axioms*]

8.3.3 Initialization of Queue

sepref-thm *init-CQ-impl* **is** (*PR-CONST init-CQ*)
$:: am\text{-}assn^k \rightarrow_a nat\text{-}assn \times_a list\text{-}assn\ nat\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *init-CQ-impl*
uses *Network-Impl.init-CQ-impl.refine-raw* **is** (*?f*,-)∈-
lemmas [*sepref-fr-rules*] = *init-CQ-impl.refine*[*OF Network-Impl-axioms*]

8.3.4 Main Algorithm

sepref-thm *relabel-to-front-impl* **is**
(*PR-CONST relabel-to-front2*) :: $am\text{-}assn^k \rightarrow_a cf\text{-}assn$
⟨*proof*⟩
concrete-definition (**in** −) *relabel-to-front-impl*
uses *Network-Impl.relabel-to-front-impl.refine-raw* **is** (*?f*,-)∈-
lemmas [*sepref-fr-rules*] = *relabel-to-front-impl.refine*[*OF Network-Impl-axioms*]

end — Network Implementation Locale

export-code *relabel-to-front-impl* **checking** *SML-imp*

8.4 Combination with Network Checker and Correctness

context *Network-Impl* **begin**

theorem *relabel-to-front-impl-correct*[*sep-heap-rules*]:
assumes *AM*: *is-adj-map am*
shows
$<am\text{-}assn\ am\ ami>$
relabel-to-front-impl c s t N ami
$<\lambda cfi.\ \exists_A cf.\ cf\text{-}assn\ cf\ cfi$
$* \uparrow(isMaxFlow\ (flow\text{-}of\text{-}cf\ cf) \wedge RGraph\text{-}Impl\ c\ s\ t\ N\ cf)>_t$
⟨*proof*⟩
end

definition *relabel-to-front-impl-tab-am c s t N am* ≡ *do* {
ami ← *Array.make N am*; — TODO/DUP: Called *init-ps* in Edmonds-Karp impl
relabel-to-front-impl c s t N ami
}

theorem *relabel-to-front-impl-tab-am-correct*[*sep-heap-rules*]:
assumes *NW*: *Network c s t*
assumes *VN*: *Graph.V c* ⊆ {*0..*<*N*}
assumes *ABS-PS*: *Graph.is-adj-map c am*
shows
<*emp*>
relabel-to-front-impl-tab-am c s t N am
<λ*cfi*. $\exists_A$ *cf*.
asmtx-assn N id-assn cf cfi
∗ ↑(*Network.isMaxFlow c s t* (*Network.flow-of-cf c cf*)
∧ *RGraph-Impl c s t N cf*
)>$_t$
⟨*proof*⟩

definition *relabel-to-front el s t* ≡ *do* {
case prepareNet el s t of
None ⇒ *return None*
| *Some* (*c*,*am*,*N*) ⇒ *do* {
cf ← *relabel-to-front-impl-tab-am c s t N am*;
return (*Some* (*c*,*am*,*N*,*cf*))
}
}
export-code *relabel-to-front* **checking** *SML-imp*

Main correctness statement:

- If *relabel-to-front* returns *None*, the edge list was invalid or described an invalid network.
- If it returns *Some* (*c*,*am*,*N*,*cfi*), then the edge list is valid and describes a valid network. Moreover, *cfi* is an integer square matrix of dimension *N*, which describes a valid residual graph in the network, whose corresponding flow is maximal. Finally, *am* is a valid adjacency map of the graph, and the nodes of the graph are integers less than *N*.

theorem *relabel-to-front-correct*:
<*emp*>
relabel-to-front el s t
<λ
None ⇒ ↑(¬*ln-invar el* ∨ ¬*Network* (*ln-α el*) *s t*)
| *Some* (*c*,*am*,*N*,*cfi*) ⇒
↑(*c* = *ln-α el* ∧ *ln-invar el*)
∗ ($\exists_A$ *cf*. *asmtx-assn N int-assn cf cfi*

$* \uparrow(\mathit{RGraph\text{-}Impl}\ c\ s\ t\ N\ cf$
$\wedge\ \mathit{Network.isMaxFlow}\ c\ s\ t\ (\mathit{Network.flow\text{-}of\text{-}cf}\ c\ cf)))$
$* \uparrow(\mathit{Graph.is\text{-}adj\text{-}map}\ c\ am)$
$>_t$

$\langle \mathit{proof} \rangle$

end

9 Conclusion

We have presented a verification of two push-relabel algorithms for solving the maximum flow problem. Starting with a generic push-relabel algorithm, we have used stepwise refinement techniques to derive the relabel-to-front and FIFO push-relabel algorithms. Further refinement yields verified efficient imperative implementations of the algorithms.

References

[1] R.-J. Back. *On the correctness of refinement steps in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[2] R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.

[3] B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4), Oct. 1988.

[6] P. Lammich and S. R. Sefidgar. Formalizing the edmonds-karp algorithm. In *Interactive Theorem Proving*. Springer, 2016. to appear.

[7] P. Lammich and S. R. Sefidgar. Formalizing the edmonds-karp algorithm. *Archive of Formal Proofs*, Aug. 2016. http://isa-afp.org/entries/EdmondsKarp_Maxflow.shtml, Formal proof development.

[8] G. Lee. Correctnesss of ford-fulkersons maximum flow algorithm1. *Formalized Mathematics*, 13(2):305–314, 2005.

[9] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), Apr. 1971.