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 $O(V^2E)$. 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 $l : node \Rightarrow nat$ 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*
  using *that* **by** (*induction p arbitrary*: *u*; *fastforce dest*: *valid*)

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*: ¬*isAugmentingPath p*
proof
  assume *isAugmentingPath p*
  hence *SP*: *cf.isSimplePath s p t* **unfolding** *isAugmentingPath-def* **.**
  hence *cf.isPath s p t* **unfolding** *cf.isSimplePath-def* **by** *auto*
  from *gen-valid*[*OF this*] **have** *length p* ≥ *card V* **by** *auto*
  with *cf.simplePath-length-less-V*[*OF - SP*] **show** *False* **by** *auto*
qed

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 ∀ *u*∈ *V*−{*s*,*t*}. *excess f u* = *0*
  shows *isMaxFlow f*
proof −
  from *assms* **interpret** *NFlow*
    apply *unfold-locales*
    using *no-deficient-nodes* **unfolding** *excess-def* **by** *auto*
  from *noAugPath-iff-maxFlow no-augmenting-path* **show** *isMaxFlow f* **by** *auto*
qed

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* ⇒ -
  where *augment-edge f* ≡ λ(*u*,*v*) Δ.
    if (*u*,*v*)∈*E then f*((*u*,*v*) := *f* (*u*,*v*) + Δ)
    else if (*v*,*u*)∈*E then f*((*v*,*u*) := *f* (*v*,*u*) − Δ)
    else f

lemma *augment-edge-zero*[*simp*]: *augment-edge f e 0* = *f*
  unfolding *augment-edge-def* **by** (*auto split*: *prod.split*)

```
lemma augment-edge-same[simp]: e∈E ⟹ augment-edge f e Δ e = f e + Δ
  unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-other[simp]:⟦e∈E; e'≠e ⟧ ⟹ augment-edge f e Δ e' = f e'

  unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-rev-same[simp]:
  (v,u)∈E ⟹ augment-edge f (u,v) Δ (v,u) = f (v,u) − Δ
  using no-parallel-edge
  unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-rev-other[simp]:
  ⟦(u,v)∉E; e'≠(v,u)⟧ ⟹ augment-edge f (u,v) Δ e' = f e'
  unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-cf[simp]: (u,v)∈E∪E⁻¹ ⟹
    cf-of (augment-edge f (u,v) Δ)
  = (cf-of f)( (u,v) := cf-of f (u,v) − Δ, (v,u) := cf-of f (v,u) + Δ)
  apply (intro ext; cases (u,v)∈E)
  subgoal for e'
    apply (cases e'=(u,v))
    subgoal by (simp split!: if-splits add: no-self-loop residualGraph-def)
    apply (cases e'=(v,u))
    subgoal by (simp split!: if-splits add: no-parallel-edge residualGraph-def)
    subgoal by (simp
                split!: if-splits prod.splits
                add: residualGraph-def augment-edge-def)
    done
  subgoal for e'
    apply (cases e'=(u,v))
    subgoal by (simp split!: if-splits add: no-self-loop residualGraph-def)
    apply (cases e'=(v,u))
    subgoal by (simp split!: if-splits add: no-self-loop residualGraph-def)
    subgoal by (simp
                split!: if-splits prod.splits
                add: residualGraph-def augment-edge-def)
    done
  done

lemma augment-edge-cf': (u,v)∈cfE-of f ⟹
    cf-of (augment-edge f (u,v) Δ)
  = (cf-of f)( (u,v) := cf-of f (u,v) − Δ, (v,u) := cf-of f (v,u) + Δ)
proof −
  assume (u,v)∈cfE-of f
  hence (u,v)∈E∪E⁻¹ using cfE-of-ss-invE ..
  thus ?thesis by simp
qed
```

The effect of augmenting an edge on the residual graph

definition (**in** −) *augment-edge-cf* :: - *flow* ⇒ - **where**
augment-edge-cf cf
≡ λ(*u*,*v*) Δ. (*cf*)((*u*,*v*) := *cf* (*u*,*v*) − Δ, (*v*,*u*) := *cf* (*v*,*u*) + Δ)

lemma *cf-of-augment-edge*:
assumes *A*: (*u*,*v*)∈*cfE-of f*
shows *cf-of* (*augment-edge f* (*u*,*v*) Δ) = *augment-edge-cf* (*cf-of f*) (*u*,*v*) Δ
proof −
show *cf-of* (*augment-edge f* (*u*, *v*) Δ) = *augment-edge-cf* (*cf-of f*) (*u*, *v*) Δ
by (*simp add*: *augment-edge-cf-def A augment-edge-cf′*)
qed

lemma *cfE-augment-ss*:
assumes *EDGE*: (*u*,*v*)∈*cfE-of f*
shows *cfE-of* (*augment-edge f* (*u*,*v*) Δ) ⊆ *insert* (*v*,*u*) (*cfE-of f*)
using *EDGE*
apply (*clarsimp simp*: *augment-edge-cf′*)
unfolding *Graph.E-def*
apply (*auto split*: *if-splits*)
done

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*≤Δ; Δ ≤ *cf* (*u*,*v*); Δ ≤ *excess f u*⟧
⟹ *Preflow c s t* (*augment-edge f* (*u*,*v*) Δ)
apply *unfold-locales*
subgoal
unfolding *residualGraph-def augment-edge-def*
using *capacity-const*
by (*fastforce split*!: *if-splits*)
subgoal
proof (*intro ballI*; *clarsimp*)
assume *0*≤Δ Δ ≤ *cf* (*u*,*v*) Δ ≤ *excess f u*
fix *v′*
assume *V′*: *v′*∈*V* *v′*≠*s* *v′*≠*t*

show *sum* (*augment-edge f* (*u*, *v*) Δ) (*outgoing v′*)
≤ *sum* (*augment-edge f* (*u*, *v*) Δ) (*incoming v′*)
proof (*cases*)
assume Δ = *0*
with *no-deficient-nodes* **show** *?thesis* **using** *V′* **by** *auto*

```
next
  assume Δ ≠ 0 with ‹0≤Δ› have 0<Δ by auto
  with ‹Δ ≤ cf (u,v)› have (u,v)∈cf.E unfolding Graph.E-def by auto

  show ?thesis
  proof (cases)
    assume [simp]: (u,v)∈E
    hence AE: augment-edge f (u,v) Δ = f ( (u,v) := f (u,v) + Δ )
      unfolding augment-edge-def by auto
    have 1: ∀ e∈outgoing v′. augment-edge f (u,v) Δ e = f e if v′≠u
      using that unfolding outgoing-def AE by auto
    have 2: ∀ e∈incoming v′. augment-edge f (u,v) Δ e = f e if v′≠v
      using that unfolding incoming-def AE by auto

    from ‹(u,v)∈E› no-self-loop have u≠v by blast

    {
      assume v′ ≠ u    v′ ≠ v
      with 1 2 V′ no-deficient-nodes have ?thesis by auto
    } moreover {
      assume [simp]: v′=v
      have sum (augment-edge f (u, v) Δ) (outgoing v′)
          = sum f (outgoing v)
        using 1 ‹u≠v› V′ by auto
      also have ... ≤ sum f (incoming v)
        using V′ no-deficient-nodes by auto
      also have ... ≤ sum (augment-edge f (u, v) Δ) (incoming v)
        apply (rule sum-mono)
        using ‹0≤Δ›
        by (auto simp: incoming-def augment-edge-def split!: if-split)
      finally have ?thesis by simp
    } moreover {
      assume [simp]: v′=u
      have A1: sum (augment-edge f (u,v) Δ) (incoming v′)
            = sum f (incoming u)
        using 2 ‹u≠v› by auto
      have (u,v) ∈ outgoing u using ‹(u,v)∈E›
        by (auto simp: outgoing-def)
      note AUX = sum.remove[OF - this, simplified]
      have A2: sum (augment-edge f (u,v) Δ) (outgoing u)
            = sum f (outgoing u) + Δ
        using AUX[of augment-edge f (u,v) Δ] AUX[of  f] by auto
      from A1 A2 ‹Δ ≤ excess f u› no-deficient-nodes V′ have ?thesis
        unfolding excess-def by auto
    } ultimately show ?thesis by blast
  next
    assume [simp]: ‹(u,v)∉E›
    hence [simp]: (v,u)∈E using cfE-ss-invE ‹(u,v)∈cf.E› by auto
    from ‹(u,v)∉E› ‹(v,u)∈E› have u≠v by blast
```

```
        have AE: augment-edge f (u,v) Δ = f ( (v,u) := f (v,u) − Δ )
          unfolding augment-edge-def by simp
        have 1: ∀ e∈outgoing v'. augment-edge f (u,v) Δ e = f e if v'≠v
          using that unfolding outgoing-def AE by auto
        have 2: ∀ e∈incoming v'. augment-edge f (u,v) Δ e = f e if v'≠u
          using that unfolding incoming-def AE by auto

        {
          assume v' ≠ u    v' ≠ v
          with 1 2 V' no-deficient-nodes have ?thesis by auto
        } moreover {
          assume [simp]: v'=u
          have A1: sum (augment-edge f (u, v) Δ) (outgoing v')
                = sum f (outgoing u)
            using 1 ‹u≠v› V' by auto

          have (v,u) ∈ incoming u
            using ‹(v,u)∈E› by (auto simp: incoming-def)
          note AUX = sum.remove[OF - this, simplified]
          have A2: sum (augment-edge f (u,v) Δ) (incoming u)
                = sum f (incoming u) − Δ
            using AUX[of augment-edge f (u,v) Δ] AUX[of f] by auto

          from A1 A2 ‹Δ ≤ excess f u› no-deficient-nodes V' have ?thesis
            unfolding excess-def by auto
        } moreover {
          assume [simp]: v'=v
          have sum (augment-edge f (u,v) Δ) (outgoing v')
              ≤ sum f (outgoing v')
            apply (rule sum-mono)
            using ‹0<Δ›
            by (auto simp: augment-edge-def)
          also have . . . ≤ sum f (incoming v)
            using no-deficient-nodes V' by auto
          also have . . . ≤ sum (augment-edge f (u,v) Δ) (incoming v')
            using 2 ‹u≠v› by auto
          finally have ?thesis by simp
        } ultimately show ?thesis by blast
      qed
    qed
  qed
  done
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*
    ≡ λ(*u*,*v*). *excess f u* > *0* ∧ (*u*,*v*)∈*cfE-of f* ∧ *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*
    ≡ λ(*u*,*v*). *augment-edge f* (*u*,*v*) (*min* (*excess f u*) (*cf-of f* (*u*,*v*)))

lemma *push-precondI*[*intro?*]:
  ⟦*excess f u* > *0*; (*u*,*v*)∈*cfE-of f*; *l u* = *l v* + *1*⟧ ⟹ *push-precond f l* (*u*,*v*)
  unfolding *push-precond-def* **by** *auto*

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*
    ≡ *u*≠*t* ∧ *excess f u* > *0* ∧ (∀ *v*. (*u*,*v*)∈*cfE-of f* ⟶ *l u* ≠ *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*
    ≡ *l*(*u* := *Min* { *l v* | *v*. (*u*,*v*)∈*cfE-of f* } + *1*)

2.2.4 Initialization

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

definition *pp-init-f* ≡ λ(*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* ≡ (λ*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*
  apply (*unfold-locales*;
      ((*auto simp*: *pp-init-f-def pp-init-l-def cap-non-negative*; *fail*)
       | (*intro ballI*) *?*))
proof −
  fix *v*
  assume $v \in V - \{s,t\}$
  hence $\forall e \in outgoing\ v.\ pp\text{-}init\text{-}f\ e = 0$
    by (*auto simp*: *outgoing-def pp-init-f-def*)
  hence [*simp*]: *sum pp-init-f* (*outgoing v*) = *0* **by** *auto*
  have $0 \leq pp\text{-}init\text{-}f\ e$ **for** *e*
    by (*auto simp*: *pp-init-f-def cap-non-negative split*: *prod.split*)
  from *sum-bounded-below*[*of incoming v 0 pp-init-f*, *OF this*]
  have $0 \leq sum\ pp\text{-}init\text{-}f\ (incoming\ v)$ **by** *auto*
  thus $sum\ pp\text{-}init\text{-}f\ (outgoing\ v) \leq sum\ pp\text{-}init\text{-}f\ (incoming\ v)$
    by *auto*

next
  fix *u v*
  assume $(u,\ v) \in Graph.E\ (residualGraph\ c\ pp\text{-}init\text{-}f)$
  thus $pp\text{-}init\text{-}l\ u \leq pp\text{-}init\text{-}l\ v + 1$
    unfolding *pp-init-l-def Graph.E-def pp-init-f-def residualGraph-def*
    by (*auto split*: *if-splits*)

qed

lemma *pp-init-f-preflow*: *NPreflow c s t pp-init-f*
proof −
  from *pp-init-invar* **interpret** *Labeling c s t pp-init-f pp-init-l* **.**
  show *?thesis* **by** *unfold-locales*
qed

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
  unfolding push-effect-def
proof (cases e; clarsimp)
  fix u v
  assume [simp]: e=(u,v)
  let ?f' = (augment-edge f (u, v) (min (excess f u) (cf (u, v))))

  from assms have
    ACTIVE: excess f u > 0
    and EDGE: (u,v)∈cf.E
    and ADM: l u = l v + 1
    unfolding push-precond-def by auto

  interpret cf': Preflow c s t ?f'
   apply (rule augment-edge-preflow-preserve)
   using ACTIVE resE-nonNegative
   by auto
  show Labeling c s t ?f' l
    apply unfold-locales using valid
    using cfE-augment-ss[OF EDGE] ADM
    apply (fastforce)
    by auto
qed

lemma finite-min-cf-outgoing[simp, intro!]: finite {l v |v. (u, v) ∈ cf.E}
proof −
  have {l v |v. (u, v) ∈ cf.E} = l‘snd‘cf.outgoing u
    by (auto simp: cf.outgoing-def)
  moreover have finite (l‘snd‘cf.outgoing u) by auto
  ultimately show ?thesis by auto
qed
```

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 −
  from PRE have
        NOT-SINK: u≠t
    and ACTIVE: excess f u > 0
    and NO-ADM: ⋀v. (u,v)∈cf.E ⟹ l u ≠ l v + 1
  unfolding relabel-precond-def by auto

  from ACTIVE have [simp]: s≠u using excess-s-non-pos by auto
```

from *active-has-cf-outgoing*[*OF ACTIVE*] **have** [*simp*]: ∃ *v*. (*u*, *v*) ∈ *cf.E*
by (*auto simp*: *cf.outgoing-def*)

from *NO-ADM valid* **have** *l u* < *l v* + *1* **if** (*u*,*v*)∈*cf.E* **for** *v*
by (*simp add*: *nat-less-le that*)
hence *LU-INCR*: *l u* ≤ *Min* { *l v* | *v*. (*u*,*v*)∈*cf.E* }
by (*auto simp*: *less-Suc-eq-le*)
with *valid* **have** ∀ *u'*. (*u'*,*u*)∈*cf.E* ⟶ *l u'* ≤ *Min* { *l v* | *v*. (*u*,*v*)∈*cf.E* } + *1*
by (*smt ab-semigroup-add-class.add.commute add-le-cancel-left le-trans*)
moreover have ∀ *v*. (*u*,*v*)∈*cf.E* ⟶ *Min* { *l v* | *v*. (*u*,*v*)∈*cf.E* } + *1* ≤ *l v* + *1*
using *Min-le* **by** *auto*
ultimately show *?G1 ?G2*
unfolding *relabel-effect-def*
apply (*clarsimp-all simp*: *PRE*)
subgoal using *LU-INCR* **by** (*simp add*: *less-Suc-eq-le*)
apply (*unfold-locales*)
subgoal for *u' v'* **using** *valid* **by** *auto*
subgoal by *auto*
subgoal using *NOT-SINK* **by** *auto*
done
qed

lemma *relabel-preserve-other*: *u*≠*v* ⟹ *relabel-effect f l u v* = *l v*
unfolding *relabel-effect-def* **by** *auto*

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*: ∀ (*u*,*v*)∈*cf.E*. ¬*push-precond f l* (*u*,*v*)
assumes *no-relabel*: ∀ *u*. ¬*relabel-precond f l u*
shows *isMaxFlow f*
proof −
from *assms* **have** ∀ *u*∈*V*−{*t*}. *excess f u* ≤ *0*
unfolding *push-precond-def relabel-precond-def*
by *force*
with *excess-non-negative* **have** ∀ *u*∈*V*−{*s*,*t*}. *excess f u* = *0* **by** *force*
with *no-excess-imp-maxflow* **show** *?thesis* **.**
qed

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′ ≡ push-effect f (u,v)
  sublocale l′: Labeling c s t f′ l
    using push-pres-Labeling[OF PRE] .

  lemma uv-cf-edge[simp, intro!]: (u,v)∈cf.E
    using PRE unfolding push-precond-def by auto
  lemma excess-u-pos: excess f u > 0
    using PRE unfolding push-precond-def by auto
  lemma l-u-eq[simp]: l u = l v + 1
    using PRE unfolding push-precond-def by auto

  lemma uv-edge-cases:
    obtains (par) (u,v)∈E   (v,u)∉E
        | (rev) (v,u)∈E   (u,v)∉E
    using uv-cf-edge cfE-ss-invE no-parallel-edge by blast

  lemma uv-nodes[simp, intro!]: u∈V   v∈V
    using E-ss-VxV cfE-ss-invE no-parallel-edge by auto

  lemma uv-not-eq[simp]: u≠v   v≠u
    using E-ss-VxV cfE-ss-invE[THEN subsetD, OF uv-cf-edge] no-parallel-edge
    by auto

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

  lemma Δ-positive: Δ > 0
    unfolding Δ-def
    using excess-u-pos uv-cf-edge[unfolded cf.E-def] resE-positive
    by auto

  lemma f′-alt: f′ = augment-edge f (u,v) Δ
    unfolding push-effect-def Δ-def by auto

  lemma cf′-alt: l′.cf = augment-edge-cf cf (u,v) Δ
    unfolding push-effect-def Δ-def augment-edge-cf-def
    by (auto simp: augment-edge-cf′)

  lemma excess′-u[simp]: excess f′ u = excess f u − Δ
    unfolding excess-def[where f=f′]
  proof −
    show sum f′ (incoming u) − sum f′ (outgoing u) = excess f u − Δ
    proof (cases rule: uv-edge-cases)
      case [simp]: par
      hence UV-ONI:(u,v)∈outgoing u − incoming u
        by (auto simp: incoming-def outgoing-def no-self-loop)
      have 1: sum f′ (incoming u) = sum f (incoming u)
        apply (rule sum.cong[OF refl])
```

```
    using UV-ONI unfolding f'-alt
    apply (subst augment-edge-other)
    by auto

  have sum f' (outgoing u)
    = sum f (outgoing u) + (∑x∈outgoing u. if x = (u, v) then Δ else 0)
    unfolding f'-alt augment-edge-def sum.distrib[symmetric]
      by (rule sum.cong) auto
  also have ... = sum f (outgoing u) + Δ
    using UV-ONI by (auto simp: sum.delta)
  finally show ?thesis using 1 unfolding excess-def by simp
 next
  case [simp]: rev
  have UV-INO:(v,u)∈incoming u − outgoing u
    by (auto simp: incoming-def outgoing-def no-self-loop)
  have 1: sum f' (outgoing u) = sum f (outgoing u)
    apply (rule sum.cong[OF refl])
    using UV-INO unfolding f'-alt
    apply (subst augment-edge-rev-other)
    by (auto)
  have sum f' (incoming u)
    = sum f (incoming u) + (∑x∈incoming u. if x = (v, u) then − Δ else 0)
    unfolding f'-alt augment-edge-def sum.distrib[symmetric]
      by (rule sum.cong) auto
  also have ... = sum f (incoming u) − Δ
    using UV-INO by (auto simp: sum.delta)
  finally show ?thesis using 1 unfolding excess-def by auto
 qed
qed

lemma excess'-v[simp]: excess f' v = excess f v + Δ
 unfolding excess-def[where f=f']
proof −
 show sum f' (incoming v) − sum f' (outgoing v) = excess f v + Δ
 proof (cases rule: uv-edge-cases)
  case [simp]: par
  have UV-INO: (u,v)∈incoming v − outgoing v
    unfolding incoming-def outgoing-def by (auto simp: no-self-loop)
  have 1: sum f' (outgoing v) = sum f (outgoing v)
    using UV-INO unfolding f'-alt
    by (auto simp: augment-edge-def intro: sum.cong)

  have sum f' (incoming v)
    = sum f (incoming v) + (∑x∈incoming v. if x=(u,v) then Δ else 0)
    unfolding f'-alt augment-edge-def sum.distrib[symmetric]
    apply (rule sum.cong)
    using UV-INO unfolding f'-alt by auto
  also have ... = sum f (incoming v) + Δ
    using UV-INO by (auto simp: sum.delta)
```

```
    finally show ?thesis using 1 by (auto simp: excess-def)
  next
    case [simp]: rev
    have UV-INO:(v,u)∈outgoing v − incoming v
      by (auto simp: incoming-def outgoing-def no-self-loop)

    have 1: sum f′ (incoming v) = sum f (incoming v)
      using UV-INO unfolding f′-alt
      by (auto simp: augment-edge-def intro: sum.cong)

    have sum f′ (outgoing v)
      = sum f (outgoing v) + (∑x∈outgoing v. if x=(v,u) then − Δ else 0)
      unfolding f′-alt augment-edge-def sum.distrib[symmetric]
      apply (rule sum.cong)
      using UV-INO unfolding f′-alt by auto
    also have ... = sum f (outgoing v) − Δ
      using UV-INO by (auto simp: sum.delta)
    finally show ?thesis using 1 by (auto simp: excess-def)
  qed
qed

lemma excess′-other[simp]:
  assumes x ≠ u   x ≠ v
  shows excess f′ x = excess f x
proof −
  have NE: (u,v)∉incoming x   (u,v)∉outgoing x
      (v,u)∉incoming x   (v,u)∉outgoing x
    using assms unfolding incoming-def outgoing-def by auto
  have
    sum f′ (outgoing x) = sum f (outgoing x)
    sum f′ (incoming x) = sum f (incoming x)
    by (auto
        simp: augment-edge-def f′-alt NE
        split!: if-split
        intro: sum.cong)
  thus ?thesis
    unfolding excess-def by auto
qed

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)
  by simp

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 $\neq$ *0*   *card E* $\geq$ *Suc 0*   *card E* > *0*
proof −
  show *card E* $\neq$ *0* **by** (*simp add*: *E-not-empty*)
  thus *card E* $\geq$ *Suc 0* **by** *linarith*
  thus *card E* > *0* **by** *auto*
qed

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* $\leq$ *2* ∗ *card E*
proof −
  have *card V* $\leq$ *card* (*fst*‘*E*) + *card* (*snd*‘*E*)
    by (*auto simp*: *card-Un-le V-alt*)
  also note *card-image-le*[*OF finite-E*]
  also note *card-image-le*[*OF finite-E*]
  finally show *card V* $\leq$ *2* ∗ *card E* **by** *auto*
qed

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*: ∀ *u*∈*V*. *l u* $\leq$ *2*∗*card V* − *1*
begin
  lemma *height-bound*′: *u*∈*V* ⟹ *l u* $\leq$ *2*∗*card V* − *1*
    using *height-bound* **by** *auto*
end

lemma (**in** *Network*) *pp-init-height-bound*:

```
  Height-Bounded-Labeling c s t pp-init-f pp-init-l
proof –
  interpret Labeling c s t pp-init-f pp-init-l by (rule pp-init-invar)
  show ?thesis by unfold-locales (auto simp: pp-init-l-def)
qed

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 –
  from push-pres-Labeling[OF assms]
  interpret l′: Labeling c s t push-effect f e l .
  show ?thesis using height-bound by unfold-locales
qed
```

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 –
  obtain U where U-def: U = {v|p v. cf.isSimplePath u p v} by blast
  have fct1: U ⊆ V
  proof
    fix v
    assume v ∈ U
    then have (u, v) ∈ cf.E*
      using U-def cf.isSimplePath-def cf.isPath-rtc by auto
    then obtain u′ where u = v ∨ ((u, u′) ∈ cf.E* ∧ (u′, v) ∈ cf.E)
      by (meson rtranclE)
    thus v ∈ V
    proof
      assume u = v
      thus ?thesis using excess-nodes-only[OF assms] by blast
    next
      assume (u, u′) ∈ cf.E* ∧ (u′, v) ∈ cf.E
      then have v ∈ cf.V unfolding cf.V-def by blast
      thus ?thesis by simp
    qed
  qed

  have s ∈ U
  proof(rule ccontr)
    assume s ∉ U
    obtain U′ where U′-def: U′ = V – U by blast
```

```
  have (∑ u∈U. excess f u)
      = (∑ u∈U. (∑ v∈U'. f (v, u))) − (∑ u∈U. (∑ v∈U'. f (u, v)))
  proof −
    have (∑ u∈U. excess f u)
        = (∑ u∈U. (∑ v∈incoming u. f v)) − (∑ u∈U.(∑ v∈outgoing u. f v))
      (is - = ?R1 − ?R2) unfolding excess-def by (simp add: sum-subtractf)
    also have ?R1 = (∑ u∈U. (∑ v∈V. f (v, u)))
      using sum-incoming-alt-flow fct1 by (meson subsetCE sum.cong)
     also have ... = (∑ u∈U. (∑ v∈U. f (v, u))) + (∑ u∈U. (∑ v∈U'. f (v,
u)))
    proof −
      have (∑ v∈V. f (v, u)) = (∑ v∈U. f (v, u)) + (∑ v∈U'. f (v, u)) for u
        using U'-def fct1 finite-V
        by (metis ab-semigroup-add-class.add.commute sum.subset-diff)
      thus ?thesis by (simp add: sum.distrib)
    qed
    also have ?R2 = (∑ u∈U. (∑ v∈V. f (u, v)))
      using sum-outgoing-alt-flow fct1 by (meson subsetCE sum.cong)
     also have ... = (∑ u∈U. (∑ v∈U. f (u, v))) + (∑ u∈U. (∑ v∈U'. f (u,
v)))
    proof −
      have (∑ v∈V. f (u, v)) = (∑ v∈U. f (u, v)) + (∑ v∈U'. f (u, v)) for u
        using U'-def fct1 finite-V
        by (metis ab-semigroup-add-class.add.commute sum.subset-diff)
      thus ?thesis by (simp add: sum.distrib)
    qed
    also have (∑ u∈U. (∑ v∈U. f (u, v))) = (∑ u∈U. (∑ v∈U. f (v, u)))
    proof −
      {
        fix A :: nat set
        assume finite A
        then have (∑ u∈A. (∑ v∈A. f (u, v))) = (∑ u∈A. (∑ v∈A. f (v, u)))
        proof (induction card A arbitrary: A)
          case 0
          then show ?case by auto
        next
          case (Suc x)
          then obtain A' a
            where o1:A = insert a A' and o2:x = card A' and o3:finite A'
            by (metis card-insert-disjoint card-le-Suc-iff le-refl nat.inject)
          then have lm:(∑ e∈A. g e) = (∑ e∈A'. g e) + g a
            for g :: nat ⇒ 'a
            using Suc.hyps(2)
            by (metis card-insert-if n-not-Suc-n
                  semiring-normalization-rules(24) sum.insert)

          have (∑ u∈A. (∑ v∈A. f (u, v)))
            = (∑ u∈A'. (∑ v∈A. f (u, v))) + (∑ v∈A. f (a, v))
```

```
          (is - = ?R1 + ?R2) using lm by auto
        also have ?R1 = (∑ u∈A'. (∑ v∈A'. f (u, v))) + (∑ u∈A'. f(u, a))
          (is - = ?R1-1 + ?R1-2) using lm sum.distrib by force
        also note add.assoc
        also have ?R1-2 + ?R2 = (∑ u∈A'. f(a, u)) + (∑ v∈A. f(v, a))
          (is - = ?R1-2' + ?R2') using lm by auto
        also have ?R1-1 = (∑ u∈A'. (∑ v∈A'. f (v, u)))
          (is - = ?R1-1') using Suc.hyps(1)[of A'] o2 o3 by auto
        also note add.assoc[symmetric]
        also have ?R1-1' + ?R1-2' = (∑ u∈A'. (∑ v∈A. f (v, u)))
          by (metis (no-types, lifting) lm sum.cong sum.distrib)
        finally show ?case using lm[symmetric] by auto
      qed
    } note this[of U]
    thus ?thesis using fct1 finite-V finite-subset by auto
  qed
  finally show ?thesis by arith
qed
moreover have (∑ u∈U. excess f u) > 0
proof −
  have u ∈ U using U-def by simp
  moreover have u ∈ U ⟹ excess f u ≥ 0 for u
    using fct1 excess-non-negative' ‹s ∉ U› by auto
  ultimately show ?thesis using assms fct1 finite-V
    by (metis Diff-cancel Diff-eq-empty-iff
          Diff-infinite-finite finite-Diff sum-pos2)
qed
ultimately have
  fct2: (∑ u∈U. (∑ v∈U'. f (v, u))) − (∑ u∈U. (∑ v∈U'. f (u, v))) > 0
  by simp

have fct3: (∑ u∈U. (∑ v∈U'. f (v, u))) > 0
proof −
  have (∑ u∈U. (∑ v∈U'. f (v, u))) ≥ 0
    using capacity-const by (simp add: sum-nonneg)
  moreover have (∑ u∈U. (∑ v∈U'. f (u, v))) ≥ 0
    using capacity-const by (simp add: sum-nonneg)
  ultimately show ?thesis using fct2 by simp
qed

have ∃ u' v'. (u' ∈ U ∧ v' ∈ U' ∧ f (v', u') > 0)
proof(rule ccontr)
  assume ¬ (∃ u' v'. u' ∈ U ∧ v' ∈ U' ∧ f (v', u') > 0)
  then have (∀ u' v'. (u' ∈ U ∧ v' ∈ U' ⟶ f (v', u') = 0))
    using capacity-const by (metis le-neq-trans)
  thus False using fct3 by simp
qed
then obtain u' v' where u' ∈ U and v' ∈ U' and f (v', u') > 0
  by blast
```

obtain *p1* **where** *cf.isSimplePath u p1 u′* **using** *U-def* ‹*u′* ∈ *U*› **by** *auto*
moreover have (*u′*, *v′*) ∈ *cf.E*
proof −
have (*v′*, *u′*) ∈ *E*
using *capacity-const* ‹*f* (*v′*, *u′*) > *0*›
by (*metis not-less zero-flow-simp*)
then have *cf* (*u′*, *v′*) > *0* **unfolding** *cf-def*
using *no-parallel-edge* ‹*f* (*v′*, *u′*) > *0*› **by** (*auto split*: *if-split*)
thus *?thesis* **unfolding** *cf.E-def* **by** *simp*
qed
ultimately have *cf.isPath u* (*p1* @ [(*u′*, *v′*)]) *v′*
using *Graph.isPath-append-edge Graph.isSimplePath-def* **by** *blast*
then obtain *p2* **where** *cf.isSimplePath u p2 v′*
using *cf.isSPath-pathLE* **by** *blast*
then have *v′* ∈ *U* **using** *U-def* **by** *auto*
thus *False* **using** ‹*v′* ∈ *U′*› **and** *U′-def* **by** *simp*
qed
then obtain *p′* **where** *cf.isSimplePath u p′ s* **using** *U-def* **by** *auto*
thus *?thesis* **..**
qed

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 −
let *?l′* = *relabel-effect f l u*

from *relabel-pres-Labeling*[*OF assms*]
interpret *l′*: *Labeling c s t f ?l′* **.**

from *assms* **have** *excess f u* > *0* **unfolding** *relabel-precond-def* **by** *auto*
with *l′.excess-imp-source-path* **obtain** *p* **where** *p-obt*: *cf.isSimplePath u p s* **.**

have *u* ∈ *V* **using** *excess-nodes-only* ‹*excess f u* > *0*› **.**
then have *length p* < *card V*
using *cf.simplePath-length-less-V*[*of u p*] *p-obt* **by** *auto*
moreover have *?l′ u* ≤ *?l′ s* + *length p*
using *p-obt l′.gen-valid*[*of u p s*] *p-obt*
unfolding *cf.isSimplePath-def* **by** *auto*
moreover have *?l′ s* = *card V*
using *l′.Labeling-axioms Labeling-def Labeling-axioms-def* **by** *auto*
ultimately have *?l′ u* ≤ *2*∗*card V* − *1* **by** *auto*
thus *Height-Bounded-Labeling c s t f ?l′*
apply *unfold-locales*
using *height-bound relabel-preserve-other*
by *metis*
qed

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 ≡ ∑v∈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 −
  let ?l' = relabel-effect f l u
  from relabel-pres-height-bound[OF assms]
  interpret l': Height-Bounded-Labeling c s t f ?l' .

  from assms have u∈V
    by (simp add: excess-nodes-only relabel-precond-def)

  hence V-split: V = insert u V by auto

  show ?thesis
    using relabel-increase-u[OF assms] relabel-preserve-other[of u]
    using l'.height-bound
    unfolding sum-heights-measure-def
    apply (rewrite at ∑-∈¤. - V-split)+
    apply (subst sum.insert-remove[OF finite-V])+
    using ‹u∈V›
    by auto
qed
end — Height Bounded Labeling

lemma (in Network) sum-height-measure-is-OV2:
  sum-heights-measure l ≤ 2*(card V)²
  unfolding sum-heights-measure-def
proof −
  have 2 * card V − l v ≤ 2 * card V for v by auto
  then have (∑v∈V. 2 * card V − l v) ≤ (∑v∈V. 2 * card V)
    by (meson sum-mono)
  also have (∑v∈V. 2 * card V) = card V * (2 * card V)
    using finite-V by auto
  finally show (∑v∈V. 2 * card V − l v) ≤ 2 * (card V)^2
    by (simp add: power2-eq-square)
qed
```

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'*
  using *assms*
  by *cases* (*simp-all add*: *push-pres-height-bound relabel-pres-height-bound*)

lemma (**in** *Height-Bounded-Labeling*) *pr-algo-term-maxflow*:
  assumes (*f*,*l*)∉*Domain pr-algo-lts*
  shows *isMaxFlow f*
proof −
  from *assms* **have** ∄ *e*. *push-precond f l e* **and** ∄ *u*. *relabel-precond f l u*
    by (*auto simp*: *Domain-iff dest*: *pr-algo-lts.intros*)
  with *push-relabel-term-imp-maxflow* **show** *?thesis* **by** *blast*
qed

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*
unfolding *push-precond-def sat-push-precond-def nonsat-push-precond-def*
by *auto*

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*
unfolding *sat-push-precond-def nonsat-push-precond-def*
by *auto*

lemma *sat-push-alt*: *sat-push-precond f l e*
⟹ *push-effect f e* = *augment-edge f e* (*cf-of f e*)
unfolding *push-effect-def push-precond-eq-sat-or-nonsat sat-push-precond-def*
by (*auto simp*: *min-absorb2*)

lemma *nonsat-push-alt*: *nonsat-push-precond f l* (*u*,*v*)
⟹ *push-effect f* (*u*,*v*) = *augment-edge f* (*u*,*v*) (*excess f u*)
unfolding *push-effect-def push-precond-eq-sat-or-nonsat nonsat-push-precond-def*

by (*auto simp*: *min-absorb1*)

end — Network

context *push-effect-locale*
begin
lemma *nonsat-push-*Δ: *nonsat-push-precond f l* (*u*,*v*) ⟹ Δ = *excess f u*
unfolding Δ*-def nonsat-push-precond-def* **by** *auto*
lemma *sat-push-*Δ: *sat-push-precond f l* (*u*,*v*) ⟹ Δ = *cf* (*u*,*v*)
unfolding Δ*-def sat-push-precond-def* **by** *auto*

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* ⟧
  ⟹ ((*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′*
 apply (*clarsimp intro*!: *ext*) **subgoal for** *x* **by** (*cases x*) *auto* **done**

lemma *is-PUSH-project-conv*[*simp*]:
 is-PUSH ∘ *project-operation* = (λ*x*. *is-SAT-PUSH′ x* ∨ *is-NONSAT-PUSH′ x*)
 apply (*clarsimp intro*!: *ext*) **subgoal for** *x* **by** (*cases x*) *auto* **done**

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 −
 have ∃ *p′*.
    Height-Bounded-Labeling c s t f′ l′
  ∧ ((*f*,*l*),*p′*,(*f′*,*l′*)) ∈ *trcl pr-algo-lts′*
  ∧ *p* = *map project-operation p′*
  using *assms*
 proof (*induction p arbitrary*: *f′ l′ rule*: *rev-induct*)
  case *Nil* **thus** *?case* **using** *Height-Bounded-Labeling-axioms* **by** *simp*
 next
  case (*snoc a p*)
  from *snoc.prems* **obtain** *fh lh*
   where *PP*: ((*f*, *l*), *p*, *fh*, *lh*) ∈ *trcl pr-algo-lts*
    and *LAST*: ((*fh*,*lh*),*a*,(*f′*,*l′*))∈*pr-algo-lts*
   by (*auto dest*!: *trcl-rev-uncons*)

  from *snoc.IH*[*OF PP*] **obtain** *p′*
   where *HBL*: *Height-Bounded-Labeling c s t fh lh*
    and *PP′*: ((*f*, *l*), *p′*, *fh*, *lh*) ∈ *trcl pr-algo-lts′*
    and [*simp*]: *p* = *map project-operation p′*
   by *blast*

from *LAST* **obtain** *a′*
where *LAST′*: ((*fh*,*lh*),*a′*,(*f′*,*l′*))∈*pr-algo-lts′*
and [*simp*]: *a* = *project-operation a′*
apply *cases*
by (*auto*
simp: *push-precond-eq-sat-or-nonsat*
dest: *relabel′*[*OF HBL*] *nonsat-push′*[*OF HBL*] *sat-push′*[*OF HBL*])

note *HBL′* =
Height-Bounded-Labeling.pr-algo-maintains-hb-labeling[*OF HBL LAST*]

from *HBL′ trcl-rev-cons*[*OF PP′ LAST′*] **show** *?case* **by** *auto*
qed
with *assms that* **show** *?thesis* **by** *blast*
qed

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 −
from *xfer-run*[*OF RUN*] **obtain** *p′*
where *RUN′*: ((*f*,*l*),*p′*,(*f′*,*l′*)) ∈ *trcl pr-algo-lts′*
and [*simp*]: *p* = *map project-operation p′* **.**

have *length* (*filter is-RELABEL p*) = *length* (*filter is-RELABEL′ p′*)
by *simp*
also from *BOUND*[*rule-format*,*OF RUN′*]
have *length* (*filter is-RELABEL′ p′*) ≤ *B* **.**
finally show *?thesis* **.**
qed

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 −

from *xfer-run*[*OF RUN*] **obtain** *p′*
where *RUN′*: ((*f*,*l*),*p′*,(*f′*,*l′*)) ∈ *trcl pr-algo-lts′*
and [*simp*]: *p* = *map project-operation p′* **.**

have [*simp*]: *length* [*x*←*p′* . *is-SAT-PUSH′ x* ∨ *is-NONSAT-PUSH′ x*]
= *length* (*filter is-SAT-PUSH′ p′*) + *length* (*filter is-NONSAT-PUSH′ p′*)
by (*induction p′*) *auto*

have *length* (*filter is-PUSH p*)
  = *length* (*filter is-SAT-PUSH′ p′*) + *length* (*filter is-NONSAT-PUSH′ p′*)
  by *simp*
also note *BOUND-SAT*[*rule-format*,*OF RUN′*]
also note *BOUND-NONSAT*[*rule-format*,*OF RUN′*]
finally show *?thesis* **by** *simp*
qed

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*) $\leq$ *2* ∗ (*card V*)^2
proof −
  from *A* **have** *length* (*filter* (*is-RELABEL′*) *p*) $\leq$ *sum-heights-measure* (*snd fxl*)
    apply (*induction rule*: *trcl.induct*)
    apply (*auto elim*!: *pr-algo-lts′.cases*)
    apply (*drule* (*1*) *Height-Bounded-Labeling.relabel-measure*)
    apply *auto*
    done
  also note *sum-height-measure-is-OV2*
  finally show *length* (*filter* (*is-RELABEL′*) *p*) $\leq$ *2* ∗ (*card V*)^2 **.**
qed

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*) $\leq$ *2* ∗ (*card V*)^2
  using *xfer-relabel-bound relabel-action-bound′ A* **by** *meson*

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* $\leq$ *l′ u*

```
  using assms
proof (induction p arbitrary: f l)
  case Nil thus ?case by simp
next
  case (Cons a p)
  then obtain fh lh
    where FIRST: ((f,l),a,(fh,lh)) ∈ pr-algo-lts'
      and PP: ((fh,lh),p,(f',l')): trcl pr-algo-lts'
    by (auto simp: trcl-conv)

  from FIRST interpret Height-Bounded-Labeling c s t f l
    by cases auto

  from FIRST Cons.IH[OF PP] show ?case
    apply (auto elim!: pr-algo-lts'.cases)
    using relabel-increase-u relabel-preserve-other
    by (metis le-trans nat-le-linear not-less)
qed

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
  using assms(1−3)
proof (induction p arbitrary: f l)
  case Nil thus ?case using E' by auto
next
  case (Cons a p)
  then obtain fh lh
    where FIRST: ((f,l),a,(fh,lh)) ∈ pr-algo-lts'
      and PP: ((fh,lh),p,(f',l')): trcl pr-algo-lts'
    by (auto simp: trcl-conv)

  from FIRST interpret Height-Bounded-Labeling c s t f l
    by cases auto

  consider
    (push) u' v'
      where push-precond f l (u',v')    fh = push-effect f (u',v')    lh=l
  | (relabel) u'
      where relabel-precond f l u'    fh=f    lh=relabel-effect f l u'
    using FIRST
    by (auto elim!: pr-algo-lts'.cases simp: push-precond-eq-sat-or-nonsat)
  then show ?case proof cases
    case push
    note [simp] = push(2,3)
```

The push operation cannot go on edge (u,v) or (v,u)

```
    from push(1) have (u′,v′)≠(u,v)    (u′,v′)≠(v,u)    (u′,v′)∈cf.E
      using ‹l u ≥ l v + 1› ‹(u,v)∉cf.E›
      by (auto simp: push-precond-def)
    hence NE′: (u,v)∉cfE-of fh using ‹(u,v)∉cf.E›
      using cfE-augment-ss[of u′ v′ f]
      by (auto simp: push-effect-def)
    from Cons.IH[OF PP - NE′] ‹l u ≥ l v + 1› show ?thesis by simp
  next
    case relabel
    note [simp] = relabel(2)

    show ?thesis
    proof (cases u′=v)
      case False
      from False relabel(3) relabel-preserve-other have [simp]: lh v = l v
        by auto
      from False relabel(3)
           relabel-preserve-other relabel-increase-u[OF relabel(1)]
      have lh u ≥ l u by (cases u′=u) auto
      with ‹l u ≥ l v + 1› have LHG: lh u ≥ lh v + 1 by auto

      from Cons.IH[OF PP LHG] ‹(u,v)∉cf.E› show ?thesis by simp
    next
      case True
      note [simp] = relabel(3)
      from True relabel-increase-u[OF relabel(1)]
      have l v < lh v by simp
      also note labels-path-increasing[OF PP, of v]
      finally show ?thesis by simp
    qed
  qed
qed

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 −
  obtain u v where [simp]: e=(u,v) by (cases e)

  have length (filter ((=) (SAT-PUSH′ (u,v))) p) ≤ 2∗card V − l v
    if ((f,l),p,(f′,l′)) ∈ trcl pr-algo-lts′ for p
    using that
  proof (induction p arbitrary: f l rule: length-induct)
    case (1 p) thus ?case
    proof (cases p)
      case Nil thus ?thesis by auto
    next
      case [simp]: (Cons a p′)
      from 1.prems obtain fh lh
```

where *FIRST*: $((f,l),a,(fh,lh)) \in$ *pr-algo-lts′*
and *PP*: $((fh,lh),p',(f',l')) \in$ *trcl pr-algo-lts′*
by (*auto dest*!: *trcl-uncons*)

from *FIRST* **interpret** *Height-Bounded-Labeling c s t f l*
by *cases auto*

show *?thesis*
proof (*cases a* = *SAT-PUSH′* (*u*,*v*))
case [*simp*]: *False*
from *1.IH PP* **have**
length (*filter* ((=) (*SAT-PUSH′* (*u*, *v*))) *p′*)
$\leq$ *2* $*$ *card V* $-$ *lh v*
by *auto*
with *FIRST* **show** *?thesis*
apply (*cases*; *clarsimp*)
proof −
fix *ua* :: *nat*
assume *a1*: *length* (*filter* ((=) (*SAT-PUSH′* (*u*, *v*))) *p′*)
$\leq$ *2* $*$ *card V* $-$ *relabel-effect f l ua v*
assume *a2*: *relabel-precond f l ua*
have *2* $*$ *card V* $-$ *relabel-effect f l ua v* $\leq$ *2* $*$ *card V* $-$ *l v*
$\longrightarrow$ *length* (*filter* ((=) (*SAT-PUSH′* (*u*, *v*))) *p′*) $\leq$ *2* $*$ *card V* $-$ *l v*
using *a1 order-trans* **by** *blast*
then show *length* (*filter* ((=) (*SAT-PUSH′* (*u*, *v*))) *p′*)
$\leq$ *2* $*$ *card V* $-$ *l v*
using *a2 a1* **by** (*metis* (*no-types*) *Labeling.relabel-increase-u*
Labeling-axioms diff-le-mono2 nat-less-le
relabel-preserve-other)
qed
next
case [*simp*]: *True*

from *FIRST* **have**
[*simp*]: *fh* = *push-effect f* (*u*,*v*)   *lh* = *l*
and *PRE*: *sat-push-precond f l* (*u*,*v*)
by (*auto elim* !: *pr-algo-lts′.cases*)

from *PRE* **have** $(u,v) \in cf.E$ $l\ u = l\ v + 1$
unfolding *sat-push-precond-def* **by** *auto*
hence $u \in V$ $v \in V$ $u \neq v$ **using** *cfE-ss-invE E-ss-VxV* **by** *auto*

have *UVNEH*: $(u,v) \notin$ *cfE-of fh*
using ‹$u \neq v$›
apply (*simp*
add: *sat-push-alt*[*OF PRE*] *augment-edge-cf′*[*OF* ‹$(u,v) \in cf.E$›])
unfolding *Graph.E-def* **by** *simp*

```
      show ?thesis
      proof (cases SAT-PUSH' (u,v) ∈ set p')
        case False
        hence [simp]: filter ((=) (SAT-PUSH' (u,v))) p' = []
          by (induction p') auto
        show ?thesis
          using bspec[OF height-bound ‹u∈V›]
          using bspec[OF height-bound ‹v∈V›]
          using card-V-ge2
          by simp
      next
        case True
        then obtain p1 p2
          where [simp]: p'=p1@SAT-PUSH' (u,v)#p2
            and NP1: SAT-PUSH' (u,v) ∉ set p1
          using in-set-conv-decomp-first[of - p'] by auto

        from NP1 have [simp]: filter ((=) (SAT-PUSH' (u,v))) p1 = []
          by (induction p1) auto

        from PP obtain f2 l2 f3 l3
          where P1: ((fh,lh),p1,(f2,l2)) ∈ trcl pr-algo-lts'
            and S: ((f2,l2),SAT-PUSH' (u,v),(f3,l3)) ∈ pr-algo-lts'
            and P2: ((f3,l3),p2,(f',l'))∈trcl pr-algo-lts'
          by (auto simp: trcl-conv)
        from S have (u,v)∈cfE-of f2 and [simp]: l3=l2
          by (auto elim!: pr-algo-lts'.cases simp: sat-push-precond-def)
        with edge-reappears-at-increased-labeling[OF P1 - UVNEH]
          ‹l u = l v + 1›
        have AUX1: l v < l2 v by auto

        from S interpret l2: Height-Bounded-Labeling c s t f2 l2
          by (auto elim!: pr-algo-lts'.cases)

        from spec[OF 1.IH, of   SAT-PUSH' (u,v)#p2] S P2 have
          Suc (length (filter ((=) (SAT-PUSH' (u, v))) p2))
          ≤ 2 * card V − l2 v
          by (auto simp: trcl-conv)
        also have … + 1 ≤ 2*card V − l v
          using AUX1
          using bspec[OF l2.height-bound ‹u∈V›]
          using bspec[OF l2.height-bound ‹v∈V›]
          by auto
        finally show ?thesis
          by simp
      qed
    qed
  qed
qed
```

```
  thus ?thesis using assms by fastforce
qed

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 –
  from A have IN-E: e∈E∪E⁻¹ if SAT-PUSH' e ∈ set p for e
    using that cfE-of-ss-invE
    apply (induction p arbitrary: f l)
    apply (auto
        simp: trcl-conv sat-push-precond-def
        elim!: pr-algo-lts'.cases
      ; blast)+
    done

  have AUX: length (filter (λa. ∃ e∈S. a = SAT-PUSH' e) p)
    = (∑ e∈S. length (filter ((=) (SAT-PUSH' e)) p)) if finite S for S
    using that
    apply induction
    apply simp
    apply clarsimp
    apply (subst length-filter-disj-or-conv; clarsimp)
    apply (fo-rule arg-cong)
    subgoal premises by (induction p) auto
    done

  have is-SAT-PUSH' a = (∃ e∈E∪E⁻¹. a = SAT-PUSH' e) if a∈set p for a
    using IN-E that by (cases a) auto
  hence length (filter is-SAT-PUSH' p)
    = length (filter (λa. ∃ e∈E∪E⁻¹. a = SAT-PUSH' e) p)
    by (auto cong: filter-cong)
  also have ... = (∑ e∈E∪E⁻¹. length (filter ((=) (SAT-PUSH' e)) p))
    by (auto simp: AUX)
  also have ... ≤ (∑ i∈E ∪ E⁻¹. 2 * card V)
    using sum-mono[OF sat-push-edge-action-bound'[OF A], where K=E∪E⁻¹] .
  also have ... ≤ 4 * card V * card E using card-Un-le[of E E⁻¹] by simp
  finally show length (filter is-SAT-PUSH' p) ≤ 4 * card V * card E .
qed

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 −
  have *nonsat-potential f l* = (∑ *v*∈{*v* ∈ *V*. *0* < *excess f v*}. *l v*)
    unfolding *nonsat-potential-def* **by** *auto*
  also have ... ≤ (∑ *v*∈*V*. *l v*)
  proof −
    have *f1*:{*v* ∈ *V*. *0* < *excess f v*} ⊆ *V* **by** *auto*
    thus *?thesis* **using** *sum.subset-diff*[*OF f1 finite-V*, *of l*] **by** *auto*
  qed
  also have ... ≤ (∑ *v*∈*V*. *2* ∗ *card V* − *1*)
    using *height-bound* **by** (*meson sum-mono*)
  also have ... = *card V* ∗ (*2* ∗ *card V* − *1*) **by** *auto*
  also have *card V* ∗ (*2* ∗ *card V* − *1*) ≤ *2* ∗ *card V* ∗ *card V* **by** *auto*
  finally show *?thesis* **by** (*simp add*: *power2-eq-square*)
qed

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 (*cases e*)
  case [*simp*]: (*Pair u v*)

  show *?thesis*
  proof *simp*
    interpret *push-effect-locale c s t f l u v*
      apply *unfold-locales* **using** *assms*

```
      by (simp add: push-precond-eq-sat-or-nonsat)

    note [simp] = nonsat-push-Δ[OF assms[simplified]]

    define S where S={x∈V. x≠u ∧ x≠v ∧ 0<excess f x}
    have S-alt: S = {x∈V. x≠u ∧ x≠v ∧ 0<excess f' x}
      unfolding S-def by auto

    have NES: s∉S   u∉S   v∉S
      and [simp, intro!]: finite S
      unfolding S-def using excess-s-non-pos
      by auto

    have 1: {v∈V. 0 < excess f' v} = (if s=v then S else insert v S)
      unfolding S-alt
      using excess-u-pos excess-non-negative' l'.excess-s-non-pos
      by (auto intro!: add-nonneg-pos)

    have 2: {v∈V. 0 < excess f v}
      = insert u S ∪ (if excess f v>0 then {v} else {})
      unfolding S-def using excess-u-pos by auto

    show nonsat-potential f' l < nonsat-potential f l
      unfolding nonsat-potential-def 1 2
      by (cases s=v; cases   0<excess f v; auto simp: NES)
  qed
qed
```

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 −
  obtain u v where [simp]: e = (u, v) by (cases e) auto

  interpret push-effect-locale c s t f l u v
    using PRE
    by unfold-locales (simp add: push-precond-eq-sat-or-nonsat)

  have [simp, intro!]: finite {v∈V. excess f v > 0}
    by auto
```

Only target node may get activated

```
  have {v∈V. excess f' v > 0} ⊆ insert v {v∈V. excess f v > 0}
    using Δ-positive
    by (auto simp: excess'-if)
```

Thus, potential increases by at most $l\ v$

with *sum-mono2*[*OF - this*, *of l*]
have *nonsat-potential f′ l* ≤ *nonsat-potential f l* + *l v*
unfolding *nonsat-potential-def*
by (*auto simp*: *sum.insert-if split*: *if-splits*)

Which is bounded by $O(V)$

also note *height-bound′*[*of v*]
finally show *?thesis* **by** *simp*
qed

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*)
≤ *nonsat-potential f l* + *2* ∗ *card V*
proof −
have [*simp*, *intro*!]: *finite* {*v*∈*V*. *excess f v* > *0*}
by *auto*

let *?l′* = *relabel-effect f l u*

interpret *l′*: *Height-Bounded-Labeling c s t f ?l′*
using *relabel-pres-height-bound*[*OF assms*] **.**

from *PRE* **have** *U-ACTIVE*: *u* ∈ {*v*∈*V*. *excess f v* > *0*} **and** [*simp*]: *u*∈*V*
unfolding *relabel-precond-def* **using** *excess-nodes-only*
by *auto*

have *nonsat-potential f ?l′*
= *sum ?l′* ({*v* ∈ *V*. *0* < *excess f v*} − {*u*}) + *?l′ u*
unfolding *nonsat-potential-def*
using *U-ACTIVE* **by** (*auto intro*: *sum-arb*)
also have *sum ?l′* ({*v* ∈ *V*. *0* < *excess f v*} − {*u*})
= *sum l* ({*v* ∈ *V*. *0* < *excess f v*} − {*u*})
using *relabel-preserve-other* **by** *auto*
also have *?l′ u* ≤ *l u* + *2*∗*card V*
using *l′.height-bound′*[*OF* ‹*u*∈*V*›] **by** *auto*
finally have *nonsat-potential f ?l′*
≤ *sum l* ({*v* ∈ *V*. *0* < *excess f v*} − {*u*}) + *l u* + *2* ∗ *card V*
by *auto*
also have *sum l* ({*v* ∈ *V*. *0* < *excess f v*} − {*u*}) + *l u*
= *nonsat-potential f l*
unfolding *nonsat-potential-def*
using *U-ACTIVE* **by** (*auto intro*: *sum-arb*[*symmetric*])
finally show *?thesis* **.**
qed

end — Height Bounded Labeling

```
context Network
begin

lemma nonsat-push-action-bound':
  assumes A: ((f,l),p,(f',l')) ∈ trcl pr-algo-lts'
  shows length (filter is-NONSAT-PUSH' p) ≤ 18 * (card V)^2 * card E
proof -
  have B1: length (filter is-NONSAT-PUSH' p)
    ≤   nonsat-potential f l
      + 2 * card V * (length (filter is-SAT-PUSH' p))
      + 2 * card V * (length (filter is-RELABEL' p))
    using A
  proof (induction p arbitrary: f l)
    case Nil thus ?case by auto
  next
    case [simp]: (Cons a p)
    then obtain fh lh
      where FIRST: ((f,l),a,(fh,lh))∈pr-algo-lts'
        and PP: ((fh,lh),p,(f',l')) ∈ trcl pr-algo-lts'
      by (auto simp: trcl-conv)
    note IH = Cons.IH[OF PP]

    from FIRST interpret Height-Bounded-Labeling c s t f l
      by cases auto

    show ?case using FIRST IH
      apply (cases a)
      apply (auto
          elim!: pr-algo-lts'.cases
          dest!: relabel-nonsat-potential nonsat-push-decr-nonsat-potential
          dest!: sat-push-nonsat-potential
      )
      done
  qed

  show ?thesis proof (cases p)
    case Nil thus ?thesis by simp
  next
    case (Cons a' p')
    then interpret Height-Bounded-Labeling c s t f l using A
      by (auto simp: trcl-conv elim!: pr-algo-lts'.cases)
    note B1
    also note nonsat-potential-bound
    also note sat-push-action-bound'[OF A]
    also note relabel-action-bound'[OF A]
    finally have length (filter is-NONSAT-PUSH' p)
```

$\leq 2 * (card\ V)^2 + 8 * (card\ V)^2 * card\ E + 4 * (card\ V)\hat{}3$
by (*simp add*: *power2-eq-square power3-eq-cube*)
also have $(card\ V)\hat{}3 \leq 2 * (card\ V)^2 * card\ E$
by (*simp add*: *card-V-est-E power2-eq-square power3-eq-cube*)
finally have *length* (*filter is-NONSAT-PUSH′ p*)
$\leq 2 * (card\ V)^2 + 16 * (card\ V)^2 * card\ E$
by *linarith*
also have $2 * (card\ V)^2 \leq 2*(card\ V)^2 * card\ E$ **by** *auto*
finally show *length* (*filter is-NONSAT-PUSH′ p*) $\leq 18 * (card\ V)^2 * card\ E$
by *linarith*
qed
qed

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*) $\leq 22 * (card\ V)^2 * card\ E$
apply (*rule order-trans*[*OF xfer-push-bounds*[*OF - - A*]]; (*intro allI impI*)?)
apply (*erule sat-push-action-bound′*; *fail*)
apply (*erule nonsat-push-action-bound′*; *fail*)
apply (*auto simp*: *power2-eq-square*)
done

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* ⇒ *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 $(\sum a{\leftarrow}p.\ cost\text{-}estimate\ a) \leq 26 * (card\ V)\hat{}2 * card\ E$
proof −
have $(\sum a{\leftarrow}p.\ cost\text{-}estimate\ a)$
= *card V* ∗ *length* (*filter is-RELABEL p*) + *length* (*filter is-PUSH p*)
proof (*induction p*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a p*)
then show *?case* **by** (*cases a*) *auto*
qed
also have *card V* ∗ *length* (*filter is-RELABEL p*) $\leq 2 * (card\ V)\hat{}3$

```
    using relabel-action-bound[OF A]
    by (auto simp: power2-eq-square power3-eq-cube)
  also note push-action-bound[OF A]
  finally have sum-list (map cost-estimate p)
          ≤ 2 * card V ^ 3 + 22 * (card V)^2 * card E
    by simp
  also have (card V)^3 ≤ 2 * (card V)^2 * card E
    by (simp add: card-V-est-E power2-eq-square power3-eq-cube)
  finally show ?thesis by linarith
qed
```

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 $O(V^2 E)$ [Cormen 26.24]. Note that this also implies termination.

```
theorem (in Network) generic-preflow-push-OV2E-and-correct:
  assumes A: ((pp-init-f, pp-init-l), p, (f, l)) ∈ trcl pr-algo-lts
  shows (∑x←p. cost-estimate x) ≤ 26 * (card V)^2 * card E (is ?G1)
    and (f,l)∉Domain pr-algo-lts ⟶ isMaxFlow f (is ?G2)
proof −
  show ?G1
    using pp-init-height-bound Height-Bounded-Labeling.pr-algo-cost-bound A
    by blast

  show ?G2
    proof −
    from A interpret Height-Bounded-Labeling c s t f l
      apply (induction p arbitrary: f l rule: rev-induct)
      apply (auto
        simp: pp-init-height-bound trcl-conv
        intro: Height-Bounded-Labeling.pr-algo-maintains-hb-labeling)
      done
    from pr-algo-term-maxflow show ?G2 by simp
  qed
qed
```

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⟧
    ⟹ ((push-effect f e,l),(f,l))∈pr-algo-rel
| relabel: ⟦Height-Bounded-Labeling c s t f l; relabel-precond f l u⟧
    ⟹ ((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 }
  by (auto elim!: pr-algo-rel.cases intro: pr-algo-rel.intros)


definition pr-algo-len-bound ≡ 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')) ∈ trcl pr-algo-lts
  shows length p ≤ pr-algo-len-bound
proof −
  have length p = length (filter is-PUSH p) + length (filter is-RELABEL p)
  proof (induction p)
    case Nil then show ?case by simp
  next
    case (Cons a p) then show ?case by (cases a) auto
  qed
  also note push-action-bound[OF A]
  also note relabel-action-bound[OF A]
  finally show ?thesis unfolding pr-algo-len-bound-def by simp
qed

lemma (in Height-Bounded-Labeling) path-set-finite:
  finite {p. ∃f' l'. ((f,l),p,(f',l')) ∈ trcl pr-algo-lts}
proof −
  have FIN-OPS: finite (UNIV::pr-operation set)
    apply (rule finite-subset[where B={PUSH,RELABEL}])
    using pr-operation.exhaust by auto

  have {p. ∃f' l'. ((f,l),p,(f',l')) ∈ trcl pr-algo-lts}
    ⊆ {p. length p ≤ pr-algo-len-bound}
    by (auto simp: pr-algo-lts-length-bound)
  also note finite-lists-length-le[OF FIN-OPS, simplified]
  finally (finite-subset) show ?thesis .
qed

definition pr-algo-measure
  ≡ λ(f,l). Max {length p |p. ∃aa ba. ((f, l), p, aa, ba) ∈ trcl pr-algo-lts}
```

```
lemma pr-algo-measure:
  assumes (fl',fl) ∈ pr-algo-rel
  shows pr-algo-measure fl' < pr-algo-measure fl
  using assms
proof (cases fl'; cases fl; simp)
  fix f l f' l'
  assume A: ((f',l'),(f,l)) ∈ pr-algo-rel
  then obtain a where LTS-STEP: ((f,l),a,(f',l'))∈pr-algo-lts
    by cases (auto intro: pr-algo-lts.intros)

  from A interpret Height-Bounded-Labeling c s t f l by cases auto
  from pr-algo-maintains-hb-labeling[OF LTS-STEP]
  interpret f': Height-Bounded-Labeling c s t f' l' .

  let ?S1 = {length p |p. ∃fx lx. ((f, l), p, fx, lx) ∈ trcl pr-algo-lts}
  let ?S2 = {length p |p. ∃fx lx. ((f', l'), p, fx, lx) ∈ trcl pr-algo-lts}

  have finite ?S1 using finite-image-set path-set-finite by blast
  moreover have ?S1 ≠ {} by (auto intro: exI[where x=[]])
  ultimately obtain p fx lx where
    length p = Max ?S1
    ((f, l), p, fx, lx) ∈ trcl pr-algo-lts
    apply −
    apply (drule (1) Max-in)
    by auto

  have finite ?S2 using finite-image-set f'.path-set-finite by blast
  have ?S2 ≠ {} by (auto intro: exI[where x=[]])
  {
    assume MG: Max ?S2 ≥ Max ?S1

    from Max-in[OF ‹finite ?S2› ‹?S2≠{}›] obtain p fx lx where
      length p = Max ?S2
      ((f', l'), p, fx, lx) ∈ trcl pr-algo-lts
      by auto
    with MG LTS-STEP have
      LEN: length (a#p) > Max ?S1
      and P: ((f,l),a#p,(fx,lx)) ∈ trcl pr-algo-lts
      by (auto simp: trcl-conv)
    from P have length (a#p) ∈ ?S1 by blast
    from Max-ge[OF ‹finite ?S1› this] LEN have False by simp
  } thus pr-algo-measure (f', l') < pr-algo-measure (f, l)
    unfolding pr-algo-measure-def by (rule ccontr) auto
qed

lemma wf-pr-algo-rel[simp, intro!]: wf pr-algo-rel
  apply (rule wf-subset)
  apply (rule wf-measure[where f=pr-algo-measure])
```

by (*auto simp*: *pr-algo-measure*)

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

The gap heuristics preserves a valid labeling.

lemma (**in** *Labeling*) *gap-pres-Labeling*:
  assumes *PRE*: *gap-precond l k*
  defines $l' \equiv gap\text{-}effect\ l\ k$
  shows *Labeling c s t f* l'
proof
  from *lab-src* **show** $l'\ s = card\ V$ **unfolding** l'*-def gap-effect-def* **by** *auto*
  from *lab-sink* **show** $l'\ t = 0$ **unfolding** l'*-def gap-effect-def* **by** *auto*

  have l'*-incr*: $l'\ v \geq l\ v$ **for** *v* **unfolding** l'*-def gap-effect-def* **by** *auto*

  fix *u v*
  assume *A*: $(u,v) \in cf.E$
  hence $u{\in}V \quad v{\in}V$ **using** *cfE-ss-invE E-ss-VxV* **by** *auto*
  thus $l'\ u \leq l'\ v + 1$
    unfolding l'*-def gap-effect-def*
    using *valid*[*OF A*] *PRE*
    unfolding *gap-precond-def*
    by *auto*
qed

The gap heuristics also preserves the height bounds.

lemma (**in** *Height-Bounded-Labeling*) *gap-pres-hb-labeling*:
  assumes *PRE*: *gap-precond l k*
  defines $l' \equiv gap\text{-}effect\ l\ k$
  shows *Height-Bounded-Labeling c s t f* l'
proof −
  from *gap-pres-Labeling*[*OF PRE*] **interpret** *Labeling c s t f* l'
    unfolding l'*-def* **.**

  show *?thesis*
    apply *unfold-locales*
    unfolding l'*-def gap-effect-def* **using** *height-bound* **by** *auto*

qed

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* ≡ *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′* ≡ *gap-relabel-effect f l u*
shows *Labeling c s t f l′*
unfolding *l′-def gap-relabel-effect-def*
using *relabel-pres-Labeling*[*OF PRE*] *Labeling.gap-pres-Labeling*
by (*fastforce simp*: *Let-def*)

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′* ≡ *gap-relabel-effect f l u*
shows *Height-Bounded-Labeling c s t f l′*
unfolding *l′-def gap-relabel-effect-def*
using *relabel-pres-height-bound*[*OF PRE*] *Height-Bounded-Labeling.gap-pres-hb-labeling*
by (*fastforce simp*: *Let-def*)

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*
unfolding *gap-effect-def sum-heights-measure-def*
by (*auto intro*!: *sum-mono*)

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*
unfolding *gap-relabel-effect-def*
using *relabel-measure*[*OF PRE*] *relabel-pres-height-bound*[*OF PRE*] *Height-Bounded-Labeling.gap-measure*
by (*fastforce simp*: *Let-def*)

Analogously to *pr-algo-rel*, we provide a well-founded relation that overapproximates 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 –
 have gap-algo-rel ⊆ inv-image (less-than <*lex*> less-than) (λ(f,l). (sum-heights-measure
l, pr-algo-measure (f,l)))
    using pr-algo-measure
    using Height-Bounded-Labeling.gap-relabel-measure
    by (fastforce elim!: gap-algo-rel.cases intro: pr-algo-rel.intros )
  thus ?thesis
    by (rule-tac wf-subset; auto)
qed

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* f_0 *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* f_0} (λ(*f*,*l*,*Q*). *excess f u* ≠ *0*) (λ*v* (*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*)
  }) (f_0,*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*
  unfolding *Q-invar-def QD-invar-def* **by** *auto*

lemma *Q-invar-when-discharged2*: ⟦*QD-invar u f Q*; *excess f u* ≠ *0*⟧ ⟹ *Q-invar f* (*Q*@[*u*])
  unfolding *Q-invar-def QD-invar-def*
  by *auto*

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 −
interpret *push-effect-locale c s t f l u v*
using *PRE* **by** *unfold-locales*

from *excess-non-negative* Δ*-positive* **have** *excess f v* + Δ ≠ *0* **if** *v*∉{*s*,*t*}
using *that* **by** *force*
thus *?thesis*
using *VC INV*
unfolding *QD-invar-def*
by (*auto simp*: *excess'-if split*!: *if-splits*)
qed

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 −
interpret *push-effect-locale c s t f l u v*
using *PRE* **by** *unfold-locales*

show *?thesis*
using *VC INV* Δ*-positive*
unfolding *QD-invar-def*
by (*auto simp*: *excess'-if split*!: *if-splits*)
qed

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 −
from *QNE* **obtain** *u Qr* **where** [*simp*]: *Q*=*u*#*Qr* **by** (*cases Q*) *auto*

from *QINV* **have** *U*: *u*∈*V*−{*s*,*t*} *QD-invar u f Qr* **and** *XU-orig*: *excess f u* ≠ *0*

by (*auto simp*: *Q-invar-def QD-invar-def*)

have [*simp*, *intro*!]: *finite* {*v*. (*u*, *v*) ∈ *cfE-of f*}
apply (*rule finite-subset*[**where** *B*=*V*])
using *cfE-of-ss-VxV*
by *auto*

show *?thesis*
using *U*
unfolding *fifo-discharge-def fifo-push-def fifo-gap-relabel-def*
apply (*simp only*: *split nres-monad-laws*)
apply (*rewrite* **in** *FOREACHc - - ⌑ - vcg-intro-frame*)
apply (*rewrite* **in** *if excess - - ≠ 0 then ⌑ else - vcg-intro-frame*)
apply (*refine-vcg FOREACHc-rule*[**where**
I=*λit* (*f′*,*l′*,*Q′*).
Height-Bounded-Labeling c s t f′ l′
∧ *QD-invar u f′ Q′*
∧ ((*f′*,*l′*),(*f*,*l*))∈*gap-algo-rel**
∧ *it* ⊆ {*v*. (*u*,*v*) ∈ *cfE-of f′* }
∧ (*excess f′ u*≠*0* ⟶ (∀ *v*∈{*v*. (*u*,*v*) ∈ *cfE-of f′* }−*it*. *l′ u* ≠ *l′ v* + *1*)
)

])
apply (*vc-solve simp*: *DINV QINV it-step-insert-iff split del*: *if-split*)
subgoal for *v it f′ l′ Q′* **proof** −
assume *HBL*: *Height-Bounded-Labeling c s t f′ l′*
then interpret *l′*: *Height-Bounded-Labeling c s t f′ l′* **.**

assume *X*: *excess f′ u* ≠ *0* **and** *UI*: *u* ∈ *V* *u* ≠ *s* *u* ≠ *t*
and *QDI*: *QD-invar u f′ Q′*

assume *v* ∈ *it* **and** *ITSS*: *it* ⊆ {*v*. (*u*, *v*) ∈ *l′*.*cf*.*E*}
hence *UVE*: (*u*,*v*) ∈ *l′*.*cf*.*E* **by** *auto*

assume *REL*: ((*f′*, *l′*), *f*, *l*) ∈ *gap-algo-rel**

assume *SAT-EDGES*: ∀ *v*∈{*v*. (*u*, *v*) ∈ *cfE-of f′*} − *it*. *l′ u* ≠ *Suc* (*l′ v*)

from *X UI l′*.*excess-non-negative* **have** *X′*: *excess f′ u* > *0* **by** *force*

have *PP*: *push-precond f′ l′* (*u*, *v*) **if** *l′ u* = *l′ v* + *1*
unfolding *push-precond-def* **using** *that UVE X′* **by** *auto*

show *?thesis*
apply (*rule vcg-rem-frame*)
apply (*rewrite* **in** *if - then* (*assert -* ≫ ⌑) *else - vcg-intro-frame*)
apply *refine-vcg*
apply (*vc-solve simp*: *REL solve*: *PP l′*.*push-pres-height-bound HBL QDI split del*: *if-split*)

```
      subgoal proof –
        assume [simp]: l' u = Suc (l' v)
        assume PRE: push-precond f' l' (u, v)
        then interpret pe: push-effect-locale c s t f' l' u v by unfold-locales

        have UVNE': l'.cf (u, v) ≠ 0
          using l'.resE-positive by fastforce

        show ?thesis
          apply (rule vcg-rem-frame)
          apply refine-vcg
          apply (vc-solve simp: l'.push-pres-height-bound[OF PRE])
          subgoal by unfold-locales
          subgoal by (auto simp: pe.cf'-alt augment-edge-cf-def)
          subgoal
            using l'.push-activate-pres-QD-invar[OF QDI PRE]
            using l'.push-no-activate-pres-QD-invar[OF QDI PRE]
            by auto
          subgoal
              by (meson gap-algo-rel.push REL PRE converse-rtrancl-into-rtrancl
HBL)
          subgoal for x proof –
            assume x∈it    x≠v
            with ITSS have (u,x)∈l'.cf.E by auto
            thus ?thesis
              using ‹x≠v›
              unfolding pe.f'-alt
              apply (simp add: augment-edge-cf')
              unfolding Graph.E-def
              by (auto)
          qed
          subgoal for v' proof –
            assume excess f' u ≠ pe.Δ
            hence PED: pe.Δ = l'.cf (u,v)
              unfolding pe.Δ-def by auto
            hence E'SS: pe.l'.cf.E ⊆ (l'.cf.E ∪ {(v,u)}) – {(u,v)}
              unfolding pe.f'-alt
              apply (simp add: augment-edge-cf')
              unfolding Graph.E-def
              by auto

            assume v' ∈ it ⟶ v' = v and UV'E: (u, v') ∈ pe.l'.cf.E and LUSLV':
l' v = l' v'
            with E'SS have v'∉it by auto
             moreover from UV'E E'SS pe.uv-not-eq(2) have (u,v')∈l'.cf.E by
auto
            ultimately have l' u ≠ Suc (l' v') using SAT-EDGES by auto
            with LUSLV' show False by simp
          qed
```

done
qed
subgoal using *ITSS* **by** *auto*
subgoal using *SAT-EDGES* **by** *auto*
done
qed
subgoal premises *prems* **for** f' l' Q' **proof** −
from *prems* **interpret** l': *Height-Bounded-Labeling c s t* f' l' **by** *simp*
from *prems* **have** *UI*: *u*∈*V* *u*≠*s* *u*≠*t*
and *X*: *excess* f' *u* ≠ *0*
and *QDI*: *QD-invar u* f' Q'
and *REL*: ((f', l'), *f*, *l*) ∈ *gap-algo-rel**
and *NO-ADM*: ∀ *v*. (*u*, *v*) ∈ l'.*cf*.*E* ⟶ l' *u* ≠ *Suc* (l' *v*)
by *simp-all*

from *X* **have** X': *excess* f' *u* > *0* **using** l'.*excess-non-negative UI* **by** *force*

from X' *UI NO-ADM* **have** *PRE*: *relabel-precond* f' l' *u*
unfolding *relabel-precond-def* **by** *auto*

from l'.*height-bound* ‹*u*∈*V*› *card-V-ge2* **have** [*simp*]: l' *u* < *2*∗*card V* **by** *auto*

from l'.*relabel-pres-height-bound*[*OF PRE*]
interpret l'': *Height-Bounded-Labeling c s t* f' *relabel-effect* f' l' *u* **.**

from l''.*height-bound* ‹*u*∈*V*› *card-V-ge2* **have** [*simp*]: *relabel-effect* f' l' *u u* < *2*∗*card V* **by** *auto*

show *?thesis*
apply (*rule vcg-rem-frame*)
apply *refine-vcg*
apply (*vc-solve*
simp: *UI PRE*
simp: l'.*gap-relabel-pres-hb-labeling*[*OF PRE*]
simp: *Q-invar-when-discharged2*[*OF QDI X*])
subgoal by *unfold-locales*
subgoal
by (*meson PRE REL gap-algo-rel.relabel* l'.*Height-Bounded-Labeling-axioms rtrancl-into-trancl2*)
done
qed
subgoal by (*auto simp*: *Q-invar-when-discharged1 Q-invar-when-discharged2*)
subgoal using *XU-orig* **by** (*metis Pair-inject rtranclD*)
subgoal by (*auto simp*: *Q-invar-when-discharged1*)
subgoal using *XU-orig* **by** (*metis Pair-inject rtranclD*)
done
qed

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 ← spec l. distinct l ∧ set l = {v∈V − {s,t}. excess f v ≠ 0}; — TODO: This
     is exactly E``{s} − {t}!

  (f,l,-) ← while_T (λ(f,l,Q). Q ≠ []) (λ(f,l,Q). 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 ≤ SPEC isMaxFlow
  unfolding fifo-push-relabel-def
  apply (refine-vcg
      WHILET-rule[where
          I=λ(f,l,Q). Height-Bounded-Labeling c s t f l ∧ Q-invar f Q
        and R=inv-image (gap-algo-rel⁺) (λ(f,l,Q). ((f,l)))
        ]
      )
  apply (vc-solve solve: pp-init-height-bound)
  subgoal by (blast intro: wf-lex-prod wf-trancl)
  subgoal unfolding Q-invar-def by auto
  subgoal for initQ f l proof −
    assume Height-Bounded-Labeling c s t f l
    then interpret Height-Bounded-Labeling c s t f l .
    assume Q-invar f []
    hence ∀u∈V−{s,t}. excess f u = 0 unfolding Q-invar-def by auto
```

  thus *isMaxFlow f* **by** (*rule no-excess-imp-maxflow*)
 qed
 done

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* ≡ { (*a*,*b*). ∃ *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* ⊆ *set l* × *set l*
 unfolding *list-before-rel-def* **by** *auto*

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*) ⟷ *distinct l*
 using *not-distinct-decomp*[*of l*]
 by (*auto simp*: *irrefl-def list-before-rel-def*)

Alternative characterization via indexes

lemma *list-before-rel-alt*: *list-before-rel l* = { (*l*!*i*, *l*!*j*) | *i j*. *i*<*j* ∧ *j*<*length l* }
 unfolding *list-before-rel-def*
 apply (*rule*; *clarsimp*)
 subgoal for *a b l1 l2 l3*
  apply (*rule exI*[*of - length l1*]; *simp*)
  apply (*rule exI*[*of - length l1* + *Suc* (*length l2*)]; *auto simp*: *nth-append*)
  done
 subgoal for *i j*
  apply (*rule exI*[*of - take i l*])
  apply (*rule exI*[*of - drop* (*Suc i*) (*take j l*)])
  apply (*rule exI*[*of - drop* (*Suc j*) *l*])
  by (*simp add*: *Cons-nth-drop-Suc drop-take-drop-unsplit*)
 done

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

```
lemma list-before-trans[trans]: distinct l ⟹ trans (list-before-rel l)
  by (clarsimp simp: trans-def list-before-rel-alt) (metis index-nth-id less-trans)

lemma list-before-asym: distinct l ⟹ asym (list-before-rel l)
  by (meson asymI irrefl-def list-before-irrefl-eq-distinct list-before-trans transE)
```

Structural properties on the list

```
lemma list-before-rel-empty[simp]: list-before-rel [] = {}
  unfolding list-before-rel-def by auto

lemma list-before-rel-cons: list-before-rel (x#l) = ({x}×set l) ∪ list-before-rel l
  apply (intro equalityI subsetI; simp add: split-paired-all)
  subgoal for a b proof −
    assume (a,b) ∈ list-before-rel (x # l)
     then obtain i j where IDX-BOUND: i<j   j<Suc (length l) and [simp]: a=(x#l)!i   b=(x#l)!j
      unfolding list-before-rel-alt by auto

    {
      assume i=0
      hence x=a   b∈set l using IDX-BOUND
        by (auto simp: nth-Cons split: nat.splits)
    } moreover {
      assume i≠0
       with IDX-BOUND have a=l!(i−1)   b=l!(j−1)   i−1 < j−1   j−1 < length l
        by auto
      hence (a, b) ∈ list-before-rel l unfolding list-before-rel-alt by blast
    } ultimately show ?thesis by blast
  qed
  subgoal premises prems for a b
  proof −
    {
      assume [simp]: a=x and b∈set l
      then obtain j where b = l!j   j<length l by (auto simp: in-set-conv-nth)
      hence a=(x#l)!0   b = (x#l)!Suc j   0 < Suc j   Suc j < length (x#l) by auto
      hence ?thesis unfolding list-before-rel-alt by blast
    } moreover {
      assume (a, b) ∈ list-before-rel l
      hence ?thesis unfolding list-before-rel-alt
        by clarsimp (metis Suc-mono nth-Cons-Suc)
    } ultimately show ?thesis using prems by blast
  qed
  done
```

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 $\textit{is-top-sorted}\ R\ l \equiv \textit{list-before-rel}\ l \cap (R^*)^{-1} = \{\}$
lemma *is-top-sorted-alt*: $\textit{is-top-sorted}\ R\ l \longleftrightarrow (\forall\, x\ y.\ (x,y)\in \textit{list-before-rel}\ l \longrightarrow (y,x)\notin R^*)$
  unfolding *is-top-sorted-def* **by** *auto*

lemma *is-top-sorted-empty-rel*[*simp*]: *is-top-sorted* {} *l* ⟷ *distinct l*
  by (*auto simp*: *is-top-sorted-def list-before-irrefl-eq-distinct*[*symmetric*] *irrefl-def*)

lemma *is-top-sorted-empty-list*[*simp*]: *is-top-sorted R* []
  by (*auto simp*: *is-top-sorted-def*)

A topological sorted list must be distinct

lemma *is-top-sorted-distinct*:
  assumes *is-top-sorted R l*
  shows *distinct l*
proof (*rule ccontr*)
  assume ¬*distinct l*
  with *list-before-irrefl-eq-distinct*[*of l*] **obtain** *x* **where** $(x,x)\in(\textit{list-before-rel}\ l)$
    by (*auto simp*: *irrefl-def*)
  with *assms* **show** *False* **unfolding** *is-top-sorted-def* **by** *auto*
qed

lemma *is-top-sorted-cons*: $\textit{is-top-sorted}\ R\ (x\#l) \longleftrightarrow (\{x\}\times \textit{set}\ l \cap (R^*)^{-1} = \{\}) \wedge \textit{is-top-sorted}\ R\ l$
  unfolding *is-top-sorted-def*
  by (*auto simp*: *list-before-rel-cons*)

lemma *is-top-sorted-append*: *is-top-sorted R* (*l1*@*l2*)
  $\longleftrightarrow (\textit{set}\ l1 \times \textit{set}\ l2 \cap (R^*)^{-1} = \{\}) \wedge \textit{is-top-sorted}\ R\ l1 \wedge \textit{is-top-sorted}\ R\ l2$
  by (*induction l1*) (*auto simp*: *is-top-sorted-cons*)

lemma *is-top-sorted-remove-elem*: *is-top-sorted R* (*l1*@*x*#*l2*) ⟹ *is-top-sorted R* (*l1*@*l2*)
  by (*auto simp*: *is-top-sorted-cons is-top-sorted-append*)

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
  using assms
  unfolding is-top-sorted-alt
  by (auto dest: rtrancl-mono-mp)
```

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

```
lemma is-top-sorted-isolated-constraint:
  assumes R′ ⊆ R ∪ {x}×X    R′∩UNIV×{x} = {}
  assumes x∉set l
  assumes is-top-sorted R l
  shows is-top-sorted R′ l
proof −
  {
    fix a b
    assume (a,b)∈R′*    a≠x    b≠x
    hence (a,b)∈R*
    proof (induction rule: converse-rtrancl-induct)
      case base
      then show ?case by simp
    next
      case (step y z)
      with assms(1,2) have z≠x    (y,z)∈R by auto
      with step show ?case by auto
    qed
  } note AUX=this

  show ?thesis
    using assms(3,4) AUX list-before-rel-on-elems
    unfolding is-top-sorted-def by fastforce
qed



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*)$^{-1}$ = {}
    unfolding *adm-edges-def* **by** *auto*

  lemma *finite-adm-edges*[*simp*, *intro*!]: *finite* (*adm-edges f l*)
    apply (*rule finite-subset*[*of* - *cfE-of f*])
    by (*auto simp*: *adm-edges-def*)

end — Network

The edge of a push operation is admissible.

lemma (**in** *push-effect-locale*) *uv-adm*: (*u*,*v*)∈*adm-edges f l*
  unfolding *adm-edges-def* **by** *auto*

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 −
  from *assms* **consider** (*sat*) *sat-push-precond f l e*
        | (*nonsat*) *nonsat-push-precond f l e*
    by (*auto simp*: *push-precond-eq-sat-or-nonsat*)
  hence *?G1* ∧ *?G2*
  proof *cases*
    case *sat* **have** *adm-edges* (*push-effect f e*) *l* = *adm-edges f l* − {*e*}
      unfolding *sat-push-alt*[*OF sat*]
    proof −

      let *?f'*=(*augment-edge f e* (*cf e*))
      interpret *l'*: *Labeling c s t ?f' l*
        using *push-pres-Labeling*[*OF assms*]
        unfolding *sat-push-alt*[*OF sat*] **.**

      from *sat* **have** *G1*: *e*∈*adm-edges f l*
        unfolding *sat-push-precond-def adm-edges-def* **by** *auto*

      have *l'*.*cf*.*E* ⊆ *insert* (*prod*.*swap e*) *cf*.*E* − {*e*}   *l'*.*cf*.*E* ⊇ *cf*.*E* − {*e*}
        unfolding *l'*.*cf-def cf-def*

```
      unfolding augment-edge-def residualGraph-def Graph.E-def
      by (auto split!: if-splits prod.splits)
    hence l'.cf.E = insert (prod.swap e) cf.E − {e} ∨ l'.cf.E = cf.E − {e}
      by auto
    thus adm-edges ?f' l = adm-edges f l − {e}
    proof (cases rule: disjE[consumes 1])
      case 1
    from sat have e ∈ adm-edges f l unfolding sat-push-precond-def adm-edges-def
by auto
      with adm-edges-inv-disj have prod.swap e ∉ adm-edges f l by (auto simp:
swap-in-iff-inv)
      thus adm-edges ?f' l = adm-edges f l − {e} using G1
        unfolding adm-edges-def 1
        by auto
    next
      case 2
      thus adm-edges ?f' l = adm-edges f l − {e}
        unfolding adm-edges-def 2
        by auto
    qed
  qed
  thus ?thesis by auto
next
  case nonsat
  hence adm-edges (push-effect f e) l = adm-edges f l
  proof (cases e; simp add: nonsat-push-alt)
    fix u v assume [simp]: e=(u,v)

    let ?f'=(augment-edge f (u,v) (excess f u))
    interpret l': Labeling c s t ?f' l
      using push-pres-Labeling[OF assms] nonsat-push-alt nonsat
      by auto

    from nonsat have e ∈ adm-edges f l
      unfolding nonsat-push-precond-def adm-edges-def by auto
    with adm-edges-inv-disj have AUX: prod.swap e ∉ adm-edges f l
      by (auto simp: swap-in-iff-inv)

    from nonsat have
      excess f u < cf (u,v)    0 < excess f u
      and [simp]: l u = l v + 1
      unfolding nonsat-push-precond-def by auto
    hence l'.cf.E ⊆ insert (prod.swap e) cf.E    l'.cf.E ⊇ cf.E
      unfolding l'.cf-def cf-def
      unfolding augment-edge-def residualGraph-def Graph.E-def
      apply (safe)
      apply (simp split: if-splits)
      apply (simp split: if-splits)
      subgoal
```

```
        by (metis (full-types) capacity-const diff-0-right
              diff-strict-left-mono not-less)
      subgoal
        by (metis add-le-same-cancel1 f-non-negative linorder-not-le)
      done
    hence l′.cf.E = insert (prod.swap e) cf.E ∨ l′.cf.E = cf.E
      by auto
    thus adm-edges ?f′ l = adm-edges f l using AUX
      unfolding adm-edges-def
      by auto
  qed
  thus ?thesis by auto
 qed
 thus ?G1 ?G2 by auto
qed
```

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 −
 from PRE have
     NOT-SINK: u≠t
  and ACTIVE: excess f u > 0
  and NO-ADM: ⋀v. (u,v)∈cf.E ⟹ l u ≠ l v + 1
 unfolding relabel-precond-def by auto

 have NE: {l v |v. (u, v) ∈ cf.E} ≠ {}
  using active-has-cf-outgoing[OF ACTIVE] cf.outgoing-def by blast
 obtain v
  where VUE: (u,v)∈cf.E and [simp]: l v = Min {l v |v. (u, v) ∈ cf.E}
  using Min-in[OF finite-min-cf-outgoing[of u] NE] by auto
 hence (u,v) ∈ adm-edges f l′ ∩ cf.outgoing u
  unfolding l′-def relabel-effect-def adm-edges-def cf.outgoing-def
  by (auto simp: cf-no-self-loop)
 thus ?G1 by blast

 {
  fix uh
  assume (uh,u) ∈ adm-edges f l′
  hence 1: l′ uh = l′ u + 1 and UHUE: (uh,u) ∈ cf.E
   unfolding adm-edges-def by auto
  hence uh ≠ u using cf-no-self-loop by auto
```

hence [*simp*]: l' *uh* = *l uh* **unfolding** l'*-def relabel-effect-def* **by** *simp*
from *1 relabel-increase-u*[*OF PRE*, *folded* l'*-def*] **have** *l uh* > *l u* + *1*
by *simp*
with *valid*[*OF UHUE*] **have** *False* **by** *auto*
}
thus *?G2* **by** (*auto simp*: *cf.incoming-def*)

show *?G3*
unfolding *adm-edges-def*
by (*auto*
simp: l'*-def relabel-effect-def cf.adjacent-def*
simp: *cf.incoming-def cf.outgoing-def*
split: *if-splits*)

qed

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* ⇒ *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* ⇒ *node set*
assumes *neighbors-adm*: ⟦*v* ∈ *adjacent-nodes u* − *n u*⟧ ⟹ (*u*,*v*) ∉ *adm-edges f*
l
assumes *neighbors-adj*: *n u* ⊆ *adjacent-nodes u*
assumes *neighbors-finite*[*simp*, *intro*!]: *finite* (*n u*)
begin

lemma *nbr-is-hbl*: *Height-Bounded-Labeling c s t f l* **by** *unfold-locales*

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 (*cases e*)
case [*simp*]: (*Pair u v*)
show *?thesis* **proof** *simp*
from *PRE* **interpret** *push-effect-locale c s t f l u v*
by *unfold-locales simp*
from *push-pres-height-bound*[*OF PRE*]
interpret l': *Height-Bounded-Labeling c s t f' l* **.**

```
    show neighbor-invar c s t f′ l n
      apply unfold-locales
      using push-adm-edges[OF PRE] neighbors-adm neighbors-adj
      by auto
  qed
qed

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 −
  let ?l′ = relabel-effect f l u
  from relabel-pres-height-bound[OF PRE]
  interpret l′: Height-Bounded-Labeling c s t f ?l′ .

  show ?thesis
    using neighbors-adj
  proof (unfold-locales; clarsimp split: if-splits)
    fix a b
    assume A: a≠u   b∈adjacent-nodes a   b ∉ n a   (a,b)∈adm-edges f ?l′
    hence (a,b)∈cf.E unfolding adm-edges-def by auto
    with A relabel-adm-edges(2,3)[OF PRE] neighbors-adm
    show False
      apply (auto)
      by (smt DiffD2 Diff-triv adm-edges-def cf.incoming-def
          mem-Collect-eq prod.simps(2) relabel-preserve-other)
  qed
qed

lemma excess-nz-iff-gz: ⟦ u∈V; u≠s ⟧ ⟹ excess f u ≠ 0 ⟷ excess f u > 0
  using excess-non-negative′ by force

lemma no-neighbors-relabel-precond:
  assumes n u = {}   u≠t   u≠s   u∈V   excess f u ≠ 0
  shows relabel-precond f l u
  using assms neighbors-adm cfE-ss-invE
  unfolding relabel-precond-def adm-edges-def
  by (auto simp: adjacent-nodes-def excess-nz-iff-gz)

lemma remove-neighbor-pres-nbr-invar: (u,v)∉adm-edges f l
  ⟹ neighbor-invar c s t f l (n (u := n u − {v}))
  apply unfold-locales
  using neighbors-adm neighbors-adj
  by (auto split: if-splits)

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 ($u \in V - \{s,t\}$);
  $while_T$ (λ(*f*,*l*,*n*). *excess f u* ≠ *0*) (λ(*f*,*l*,*n*). *do* {
    v ← *select v*. *v*∈*n u*;
    case v of
      None ⇒ *do* {
        l ← *relabel f l u*;
        return (*f*,*l*,*n*(*u* := *adjacent-nodes u*))
      }
    | *Some v* ⇒ *do* {
        assert ($v \in V \wedge (u,v) \in E \cup E^{-1}$);
        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*)) ∈ $pr\text{-}algo\text{-}rel^*$
begin

lemma *u-node-simp1*[*simp*]: *u*≠*s* *u*≠*t* *s*≠*u* *t*≠*u* **using** *u-node* **by** *auto*
lemma *u-node-simp2*[*simp*, *intro*!]: *u*∈*V* **using** *u-node* **by** *auto*

```
lemma dis-is-lbl: Labeling c s t f l by unfold-locales
lemma dis-is-hbl: Height-Bounded-Labeling c s t f l by unfold-locales
lemma dis-is-nbr: neighbor-invar c s t f l n by unfold-locales

lemma new-adm-imp-relabel:
  (u',v)∈adm-edges f l − adm-edges fo lo ⟹ lo u ≠ l u
  using no-relabel-adm-edges adm-edges-leaving-u by auto

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 −
  from PRE interpret push-effect-locale by unfold-locales

  from push-pres-nbr-invar[OF PRE] interpret neighbor-invar c s t f' l n .

  show discharge-invar c s t u fo lo no f' l n
    apply unfold-locales
    subgoal using lu-incr by auto
    subgoal by auto
    subgoal using no-relabel-adm-edges push-adm-edges(2)[OF PRE] by auto
    subgoal for v' proof −
      assume LOU: lo u = l u
      assume EXNE: excess fo v' ≠ excess f' v'
      assume UNV': u≠v'
      {
        assume excess fo v' ≠ excess f v'
        from no-relabel-excess[OF LOU UNV' this] have ?thesis .
      } moreover {
        assume excess fo v' = excess f v'
        with EXNE have excess f v' ≠ excess f' v' by simp
        hence v'=v using UNV' by (auto simp: excess'-if split: if-splits)
        hence ?thesis using no-relabel-adm-edges[OF LOU] uv-adm by auto
      } ultimately show ?thesis by blast
    qed
    subgoal
     by (meson Diff-iff push-adm-edges(2)[OF PRE] adm-edges-leaving-u subsetCE)

    subgoal
      using push-adm-edges(2)[OF PRE] relabel-u-no-incoming-adm by blast
    subgoal
      using converse-rtrancl-into-rtrancl[
              OF pr-algo-rel.push[OF dis-is-hbl PRE] algo-rel]
      .
    done
qed

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 -
  let ?l' = relabel-effect f l u
  let ?n' = n(u := adjacent-nodes u)
  from relabel-pres-nbr-invar[OF PRE]
  interpret l': neighbor-invar c s t f ?l' ?n' .

  note lu-incr
  also note relabel-increase-u[OF PRE]
  finally have INCR: lo u < ?l' u .

  show ?thesis
    apply unfold-locales
    using INCR
    apply simp-all
    subgoal for u' v
    proof clarsimp
      assume IN': (u', v) ∈ adm-edges f ?l'
        and NOT-INO: (u', v) ∉ adm-edges fo lo
      {
        assume IN: (u', v) ∈ adm-edges f l
        with adm-edges-leaving-u NOT-INO have u'=u by auto
      } moreover {
        assume NOT-IN: (u', v) ∉ adm-edges f l
        with IN' relabel-adm-edges[OF PRE] have u'=u
          unfolding cf.incoming-def cf.outgoing-def cf.adjacent-def
          by auto
      } ultimately show ?thesis by blast
    qed
    subgoal
      using relabel-adm-edges(2)[OF PRE]
      unfolding adm-edges-def cf.incoming-def
      by fastforce
    subgoal
      using converse-rtrancl-into-rtrancl[
              OF pr-algo-rel.relabel[OF dis-is-hbl PRE] algo-rel]
      .
    done
qed

lemma push-precondI-nz:
  ⟦excess f u ≠ 0; (u,v)∈cfE-of f; l u = l v + 1⟧ ⟹ push-precond f l (u,v)
  unfolding push-precond-def by (auto simp: excess-nz-iff-gz)


lemma remove-neighbor-pres-dis-invar:
  assumes PRE: (u,v)∉adm-edges f l
```

defines $n' \equiv n\ (u := n\ u - \{v\})$
shows *discharge-invar c s t u fo lo no f l n′*
proof −
from *remove-neighbor-pres-nbr-invar*[*OF PRE*]
interpret *neighbor-invar c s t f l n′* **unfolding** *n′-def* **.**
show *?thesis*
apply *unfold-locales*
using *lu-incr no-relabel-adm-edges no-relabel-excess adm-edges-leaving-u relabel-u-no-incoming-adm algo-rel*
by *auto*
qed

lemma *neighbors-in-V*: $v \in n\ u \Longrightarrow v \in V$
using *neighbors-adj*[*of u*] *E-ss-VxV* **unfolding** *adjacent-nodes-def* **by** *auto*

lemma *neighbors-in-E*: $v \in n\ u \Longrightarrow (u,v) \in E \cup E^{-1}$
using *neighbors-adj*[*of u*] *E-ss-VxV* **unfolding** *adjacent-nodes-def* **by** *auto*

lemma *relabeled-node-has-outgoing*:
assumes *relabel-precond f l u*
shows $\exists\, v.\ (u,v) \in$ *cfE-of f*
using *assms* **unfolding** *relabel-precond-def*
using *active-has-cf-outgoing* **unfolding** *cf.outgoing-def* **by** *auto*

end

lemma (**in** *neighbor-invar*) *discharge-invar-init*:
assumes $u \in V - \{s,t\}$
shows *discharge-invar c s t u f l n f l n*
using *assms*
by *unfold-locales auto*

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*: $u \neq t \quad u \neq s$ **and** *UIV*: $u \in V$
shows *discharge f l n u*
$\leq$ *SPEC* $(\lambda(f',l',n').$ *discharge-invar c s t u f l n f′ l′ n′*
$\wedge$ *excess* $f'\ u = 0)$
unfolding *discharge-def push-def relabel-def*
apply (*refine-vcg WHILET-rule*[**where**
$I = \lambda(f',l',n').$ *discharge-invar c s t u f l n f′ l′ n′*
and *R=inv-image* (*pr-algo-rel* <∗*lex*∗> *finite-psubset*)
$(\lambda(f',l',n').\ ((f',l'),n'\ u))$]

)
apply (*vc-solve*
solve: *wf-lex-prod DINV*
solve: *neighbor-invar.discharge-invar-init*[*OF DINV*]
solve: *neighbor-invar.no-neighbors-relabel-precond*
solve: *discharge-invar.relabel-pres-dis-invar*
solve: *discharge-invar.push-pres-dis-invar*
solve: *discharge-invar.push-precondI-nz pr-algo-rel.relabel*
solve: *pr-algo-rel.push*[*OF discharge-invar.dis-is-hbl*]
solve: *discharge-invar.remove-neighbor-pres-dis-invar*
solve: *discharge-invar.neighbors-in-V*
solve: *discharge-invar.relabeled-node-has-outgoing*
solve: *discharge-invar.dis-is-hbl*
intro: *discharge-invar.dis-is-nbr*
solve: *discharge-invar.dis-is-lbl*
simp: *NOT-ST*
simp: *neighbor-invar.neighbors-finite*[*OF discharge-invar.dis-is-nbr*] *UIV*)
subgoal by (*auto dest*: *discharge-invar.neighbors-in-E*)
subgoal unfolding *adm-edges-def* **by** *auto*
subgoal by (*auto*)
done

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*)
unfolding *rtf-init-n-def*
by *auto*

lemma *init-no-adm-edges*[*simp*]: *adm-edges pp-init-f pp-init-l* = {}
unfolding *adm-edges-def pp-init-l-def*
using *card-V-ge2*
by *auto*

lemma *rtf-init-neighbor-invar*:
neighbor-invar c s t pp-init-f pp-init-l rtf-init-n
proof −
from *pp-init-height-bound*
interpret *Height-Bounded-Labeling c s t pp-init-f pp-init-l* **.**

have [*simp*]: *rtf-init-n u* ⊆ *adjacent-nodes u* **for** *u*

```
    by (auto simp: rtf-init-n-def)

  show ?thesis by unfold-locales auto
qed


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 by unfold-locales

  lemma L-distinct: distinct (L-left @ L-right)
    using is-top-sorted-distinct[OF L-sorted] .

  lemma terminated-imp-maxflow:
    assumes [simp]: L-right = []
    shows isMaxFlow f
  proof −
    from L-set left-no-excess have ∀ u∈V−{s,t}. excess f u = 0 by auto
    with no-excess-imp-maxflow show ?thesis .
  qed


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 −
  from rtf-init-neighbor-invar
  interpret neighbor-invar c s t pp-init-f pp-init-l rtf-init-n .
  show ?thesis using DIS L-set by unfold-locales auto
qed

theorem relabel-to-front-correct:
  relabel-to-front ≤ SPEC isMaxFlow
  unfolding relabel-to-front-def
  apply (rewrite in while_T - ⌑ vcg-intro-frame)
  apply (refine-vcg
      WHILET-rule[where
            I=λ(f,l,n,L-left,L-right). rtf-invar c s t f l n L-left L-right
        and R=inv-image
                (pr-algo-rel⁺ <*lex*> less-than)
                (λ(f,l,n,L-left,L-right). ((f,l),length L-right))
        ]
      )
  apply (vc-solve simp: rtf-init-invar rtf-invar.rtf-is-nbr)
  subgoal by (blast intro: wf-lex-prod wf-trancl)
  subgoal for - f l n L-left L-right proof −
    assume rtf-invar c s t f l n L-left L-right
    then interpret rtf-invar c s t f l n L-left L-right .
```

assume *L-right* ≠ [] **then obtain** *u L-right′*
  where [*simp*]: *L-right* = *u*#*L-right′* **by** (*cases L-right*) *auto*

from *L-set* **have** [*simp*]: *u*∈*V*   *u*≠*s*   *u*≠*t*   *s*≠*u*   *t*≠*u* **by** *auto*

from *L-distinct* **have** [*simp*]: *u*∉*set L-left*   *u*∉*set L-right′* **by** *auto*

show *?thesis*
  apply (*rule vcg-rem-frame*)
  apply (*rewrite* **in** *do* {(-,-,-) ← *discharge* - - - -; ⌑} *vcg-intro-frame*)
  apply *refine-vcg*
  apply (*vc-solve simp*: *rtf-is-nbr split del*: *if-split*)
  subgoal for *f′ l′ n′* **proof** −
    assume *discharge-invar c s t u f l n f′ l′ n′*
    then interpret *l′*: *discharge-invar c s t u f l n f′ l′ n′* **.**

    assume [*simp*]: *excess f′ u* = *0*

    show *?thesis*
      apply (*rule vcg-rem-frame*)
      apply *refine-vcg*
      apply (*vc-solve*)
      subgoal proof −
        assume *RELABEL*: *l′ u* ≠ *l u*

        have *AUX1*: *x*=*u* **if** (*x*, *u*) ∈ (*adm-edges f′ l′*)* **for** *x*
          using *that l′.relabel-u-no-incoming-adm*[*OF RELABEL*[*symmetric*]]
          by (*auto elim*: *rtranclE*)

        have *TS1*: *is-top-sorted* (*adm-edges f l*) (*L-left* @ *L-right′*)
          using *L-sorted* **by** (*auto intro*: *is-top-sorted-remove-elem*)

        — Intuition:
          — new edges come from *u*, but *u* has no incoming edges, nor is it in *L-left*@*L-right′*.
          — thus, these new edges cannot add effective constraints.
        from *l′.adm-edges-leaving-u*
          and *l′.relabel-u-no-incoming-adm*[*OF RELABEL*[*symmetric*]]
        have *adm-edges f′ l′* ⊆ *adm-edges f l* ∪ {*u*}×*UNIV*
          and *adm-edges f′ l′* ∩ *UNIV*×{*u*}={} **by** *auto*
        from *is-top-sorted-isolated-constraint*[*OF this - TS1*]
        have *AUX2*: *is-top-sorted* (*adm-edges f′ l′*) (*L-left* @ *L-right′*)
          by *simp*

        show *rtf-invar c s t f′ l′ n′* [*u*] (*L-left* @ *L-right′*)
          apply *unfold-locales*
          subgoal by *simp*
          subgoal using *AUX2* **by** (*auto simp*: *is-top-sorted-cons dest*!: *AUX1*)

```
        subgoal using L-set by auto
        done
      qed
      subgoal using l'.algo-rel by (auto dest: rtranclD)
      subgoal proof −
        assume NO-RELABEL[simp]: l' u = l u
        — Intuition: non-zero excess would imply an admissible edge contrary to
  top-sorted.
        have AUX: excess f' v = 0 if v∈set L-left for v
        proof (rule ccontr)
          from that ‹u∉set L-left› have u ≠ v by blast
          moreover assume excess f' v ≠ 0
          moreover from that left-no-excess have excess f v = 0 by auto
          ultimately have (u,v)∈adm-edges f l
            using l'.no-relabel-excess[OF NO-RELABEL[symmetric]]
            by auto

          with L-sorted that show False
            by (auto simp: is-top-sorted-append is-top-sorted-cons)
        qed
        show rtf-invar c s t f' l' n' (L-left @ [u]) L-right'
          apply unfold-locales
          subgoal by (auto simp: AUX)
          subgoal
            apply (rule is-top-sorted-antimono[
              OF l'.no-relabel-adm-edges[OF NO-RELABEL[symmetric]]])
            using L-sorted by simp
          subgoal using L-set by auto
          done
      qed
      subgoal using l'.algo-rel by (auto dest: rtranclD)
      done
    qed
    done
  qed
  subgoal by (auto intro: rtf-invar.terminated-imp-maxflow)
  done

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*

```
  by (auto simp: RELATES-def)

definition pp-init-x
  ≡ λu. (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
  apply (intro ext)
  subgoal for u
    unfolding excess-def pp-init-f-def pp-init-x-def
    apply (cases u=s)
    subgoal
      unfolding outgoing-def incoming-def
      by (auto intro: sum.cong simp: sum-negf)
    subgoal proof −
      assume [simp]: u≠s
      have [simp]:
        (case e of (u, v) ⇒ if u = s then c (u, v) else 0) = 0
        if e∈outgoing u for e
        using that by (auto simp: outgoing-def)
      have [simp]: (case e of (u, v) ⇒ if u = s then c (u, v) else 0)
        = (if e = (s,u) then c (s,u) else 0)
        if e∈incoming u for e
        using that by (auto simp: incoming-def split: if-splits)
      show ?thesis by (simp add: sum.delta) (simp add: incoming-def)
    qed
    done
  done

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
  apply (intro ext)
  unfolding pp-init-cf-def pp-init-f-def residualGraph-def
  using no-parallel-edge
  by auto


lemma pp-init-x-rel: ((pp-init-x, pp-init-cf), pp-init-f) ∈ xf-rel
  unfolding xf-rel-def by auto
```

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 −
  have ADJ-S-AUX: adjacent-nodes s = {v . (s,v)∈E}
    unfolding adjacent-nodes-def using no-incoming-s by auto

  have CSU-AUX: c (s,u) = 0 if u∉adjacent-nodes s for u
    using that unfolding adjacent-nodes-def by auto

  show ?thesis
    unfolding pp-init-xcf2-aux-def
    apply (refine-vcg FOREACH-rule[where I=λit (x,cf).
        x s = (∑ v∈adjacent-nodes s − it. − c(s,v))
      ∧ (∀ v∈adjacent-nodes s. x v = (if v∈it then 0 else c (s,v)))
      ∧ (∀ v∈−insert s (adjacent-nodes s). x v = 0)
      ∧ (∀ v∈adjacent-nodes s.
            if v∉it then cf (s,v) = 0 ∧ cf (v,s) = c (s,v)
            else cf (s,v) = c (s,v) ∧ cf (v,s) = c (v,s))
      ∧ (∀ u v. u≠s ∧ v≠s ⟶ cf (u,v) = c (u,v) )
      ∧ (∀ u. u∉adjacent-nodes s ⟶ cf (u,s) = 0 ∧ cf (s,u) = 0)
    ])
    apply (vc-solve simp: it-step-insert-iff simp: CSU-AUX)
    subgoal for v it by (auto simp: ADJ-S-AUX)
    subgoal for u it - v by (auto split: if-splits)
    subgoal by (auto simp: ADJ-S-AUX)
    subgoal by (auto simp: ADJ-S-AUX)
    subgoal by (auto split: if-splits)

    subgoal for x
      unfolding pp-init-x-def
      apply (intro ext)
      subgoal for u
        apply (clarsimp simp: ADJ-S-AUX outgoing-def; intro conjI)
        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)
        done
      done
    subgoal for x cf
      unfolding pp-init-cf-def
      apply (intro ext)
```

```
    apply (clarsimp; auto simp: CSU-AUX)
    done
  done
qed

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


lemma pp-init-xcf2-refine-aux:
  assumes AM: is-adj-map am
  shows pp-init-xcf2 am ≤ ⇓Id (pp-init-xcf2-aux)
  unfolding pp-init-xcf2-def pp-init-xcf2-aux-def
  unfolding x-init-def cf-init-def am-get-def cf-get-def cf-set-def x-add-def
  apply (simp only: nres-monad-laws)
  supply LFO-refine[OF am-to-adj-nodes-refine[OF AM], refine]
  apply refine-rcg
  using E-ss-VxV
  by auto


lemma pp-init-xcf2-refine[refine2]:
  assumes AM: is-adj-map am
  shows pp-init-xcf2 am ≤⇓xf-rel (RETURN pp-init-f)
  using pp-init-xcf2-refine-aux[OF AM] pp-init-xcf2-aux-spec pp-init-x-rel
  by (auto simp: pw-le-iff refine-pw-simps)
```

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) (λv 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}
  by auto

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 −
  have AUX: cf (u,v) ≠ 0 ⟷ (u,v)∈cf.E for v unfolding cf.E-def by auto

  have EQ: { l v | v. (u,v)∈cf.E }
    = { l v | v. v∈adjacent-nodes u ∧ cf (u,v)≠0 }
    unfolding AUX
    using cfE-ss-invE
    by (auto simp: adjacent-nodes-def)

  define Min-option :: nat set ⇀ nat
    where Min-option X ≡ if X={} then None else Some (Min X) for X

  from PRE active-has-cf-outgoing have cf.outgoing u ≠ {}
    unfolding relabel-precond-def by auto
  hence [simp]: u∈V unfolding cf.outgoing-def using cfE-of-ss-VxV by auto
  from ‹cf.outgoing u ≠ {}›
  have AUX2: ∃ v. v ∈ adjacent-nodes u ∧ cf (u, v) ≠ 0
    by (smt AUX Collect-empty-eq Image-singleton-iff UnCI adjacent-nodes-def
          cf.outgoing-def cf-def converse-iff prod.simps(2))

  show ?thesis unfolding min-adj-label-aux-def EQ
    apply (refine-vcg
        FOREACH-rule[where
          I=λit x. x = Min-option
                        { l v | v. v∈adjacent-nodes u − it ∧ cf (u,v)≠0 }]
        )
    apply (vc-solve
        simp: Min-option-def it-step-insert-iff set-filter-xform-aux
```

```
      split: if-splits)
    subgoal unfolding adjacent-nodes-def by auto
    subgoal unfolding adjacent-nodes-def by auto
    subgoal using adjacent-nodes-ss-V by auto
    subgoal using adjacent-nodes-ss-V by auto
    subgoal by (auto simp: Min.insert-remove)
    subgoal using AUX2 by auto
    done
qed

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 −
  interpret Height-Bounded-Labeling c s t f l by fact

  have min-adj-label am (cf-of f) l u ≤ ⇓Id (min-adj-label-aux (cf-of f) l u)
    unfolding min-adj-label-def min-adj-label-aux-def Let-def
    unfolding am-get-def cf-get-def l-get-def
    apply (simp only: nres-monad-laws)
    supply LFO-refine[OF fun-relD[OF AM IdI] - IdI, refine]
    apply (refine-rcg)
    apply refine-dref-type
    by auto
  also note min-adj-label-aux-spec[OF PRE]
```

finally show *?thesis* **by** *simp*
qed

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 −
  have [*simp*]: {*l v* |*v*. *v* ∈ *V* ∧ *cf-of f* (*u*, *v*) ≠ *0*} = {*l v* |*v*. *cf-of f* (*u*, *v*) ≠ *0*}
    using *cfE-of-ss-VxV*[*of f*]
    by (*auto simp*: *Graph.E-def*)

  show *?thesis*
    using *assms*
    unfolding *relabel2-def relabel-def*
    unfolding *l-set-def*
    apply (*refine-vcg AM*)
    apply (*vc-solve* (*nopre*) *simp*: *xf-rel-def relabel-effect-def solve*: *asm-rl*)
    subgoal premises *prems* **for** *a* **proof** −
      from *prems* **interpret** *Height-Bounded-Labeling c s t f l* **by** *simp*
      interpret *l′*: *Height-Bounded-Labeling c s t f relabel-effect f l u*
        by (*rule relabel-pres-height-bound*) (*rule prems*)
      from *prems* **have** *u*∈*V* **by** *simp*
      from *prems* **have** *a* + *1* = *relabel-effect f l u u*
        by (*auto simp*: *relabel-effect-def*)
      also note *l′.height-bound*[*THEN bspec*, *OF* ‹*u*∈*V*›]
      finally show *a* + *1* < *2* ∗ *card V* **using** *card-V-ge2* **by** *auto*
    qed
    done
qed

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×r Id⟧
    ⟹ push2-aux x cf ei ≤ ⇓xf-rel (push f l e)
  unfolding push-def push2-aux-def
  apply refine-vcg
  apply (vc-solve simp: xf-rel-def no-self-loop)
  subgoal for u v
    unfolding push-precond-def using cfE-of-ss-invE by auto
  subgoal for u v
  proof −
    assume [simp]: Labeling c s t f l
    then interpret Labeling c s t f l .
    assume push-precond f l (u, v)
    then interpret l′: push-effect-locale c s t f l u v by unfold-locales
    show ?thesis
      apply (safe intro!: ext)
      using l′.excess′-if l′.Δ-def l′.cf′-alt l′.uv-not-eq(1)
      by (auto)
  qed
  done

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×r Id
  shows push2 x cf ei ≤ ⇓xf-rel (push f l e)
proof −
  have push2 x cf ei ≤ (push2-aux x cf ei)
    unfolding push2-def push2-aux-def
    unfolding x-get-def x-add-def cf-get-def cf-set-def
    unfolding augment-edge-cf-def
    apply (simp only: nres-monad-laws)
    apply refine-vcg
```

using *E-ss-VxV*
by *auto*
also note *push2-aux-refine*[*OF assms*]
finally show *?thesis* **.**
qed

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* **using** *card-mono*[*OF - V-ss*] **by** *auto*
lemma *N-not-Z*: *N* ≠ *0* **using** *card-V-ge2 cardV-le-N* **by** *auto*
lemma *N-ge-2*: *2*≤*N* **using** *card-V-ge2 cardV-le-N* **by** *auto*

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 −
have *AUX*: {*u*. *u* ≠ *s* ∧ *u* ∈ *V*} = *V*−{*s*} **by** *auto*

show *?thesis*

unfolding *clc-init-def l-init-def cnt-init-def*
apply *refine-vcg*
unfolding *clc-rel-def clc-invar-def*
using *cardV-le-N N-not-Z*
by (*auto simp*: *in-br-conv V-not-empty AUX l-invar-def*)
qed

lemma *clc-get-refine*[*refine*]:
⟦ (*clc*,*l*)∈*clc-rel*; (*ui*,*u*)∈*nat-rel* ⟧ ⟹ *clc-get clc ui* ≤⇓*Id* (*l-get l u*)
unfolding *clc-get-def clc-rel-def*
by (*auto simp*: *in-br-conv split*: *prod.split*)

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*)
unfolding *clc-get-rlx-def clc-rel-def*
by (*auto simp*: *in-br-conv split*: *prod.split*)

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

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*)
unfolding *clc-set-def l-set-def l-get-def cnt-decr-def cnt-incr-def*
apply *refine-vcg*
apply *vc-solve*
unfolding *clc-rel-def in-br-conv clc-invar-def l-invar-def*
subgoal using *cardV-le-N* **by** *auto*
applyS *auto*
applyS (*auto simp*: *simp*: *card-gt-0-iff*)

subgoal for *cnt ll*
apply *clarsimp*
apply (*intro impI conjI*; *clarsimp?*)
subgoal
apply (*subst le-imp-diff-is-add*; *simp*)
apply (*rule card-insert-disjointI*[**where** *x*=*u*])
by *auto*
subgoal
apply (*rule card-insert-disjointI*[**where** *x*=*u*, *symmetric*])

```
      by auto
    subgoal
      by (auto intro!: arg-cong[where f=card])
    done
  done

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)
  unfolding clc-has-gap-def cnt-get-def gap-precond-def
  apply refine-vcg
  unfolding clc-rel-def clc-invar-def in-br-conv
  by auto
```

6.2.7 Refinement of Gap-Heuristics

Utilities to Implement Gap-Heuristics

```
definition gap-aux C l k ≡ do {
  nfoldli [0..<N] (λ-. True) (λv 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)
  unfolding gap-effect-def l-invar-def
  by auto

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

  unfolding relabel-effect-def l-invar-def by auto

lemma gap-aux-correct[THEN order-trans, refine-vcg]:
  ⟦l-invar l; C=card V⟧ ⟹ gap-aux C l k ≤ SPEC (λr. r=gap-effect l k)
  unfolding gap-aux-def l-get-rlx-def l-set-def
  apply (simp only: nres-monad-laws)
  apply (refine-vcg nfoldli-rule[where I = λit1 it2 l'. ∀u. if u∈set it2 then l' u =
l u else l' u = gap-effect l k u])
  apply (vc-solve simp: upt-eq-lel-conv)
  subgoal
    apply (frule gap-effect-invar[where k=k])
    unfolding l-invar-def using V-ss by force
  subgoal using N-not-Z cardV-le-N by auto
  subgoal unfolding l-invar-def by auto
  subgoal unfolding gap-effect-def by auto
```

```
  subgoal for v l' u
    apply (drule spec[where x=u])
    by (auto split: if-splits simp: gap-effect-def)
  subgoal by auto
  done

definition gap2 C clc k ≡ do {
  nfoldli [0..<N] (λ-. True) (λv 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)
  unfolding gap2-def gap-aux-def
  apply (refine-rcg CLC)
  apply refine-dref-type
  by auto


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
  unfolding gap-relabel-aux-def gap-relabel-def relabel-def
    gap-relabel-effect-def l-get-def
  apply (simp only: Let-def nres-monad-laws)
  apply refine-vcg
  by auto



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 −
  have clc-relabel2 am cf clc ui ≤⇓clc-rel (relabel2 am cf l ui)
    unfolding clc-relabel2-def relabel2-def
    apply (refine-rcg)
    apply (refine-dref-type)
    apply (vc-solve simp: CLC)
    subgoal
      using CLC
      unfolding clc-rel-def in-br-conv min-adj-label-clc-def
      by (auto split: prod.split)
    done
  also note relabel2-refine[OF XF AM, of l l ui u]
  finally show ?thesis by simp
qed


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)
  unfolding gap-relabel2-def gap-relabel-aux-def
  apply (refine-vcg XCF AM CLC if-bind-cond-refine bind-refine')
  apply (vc-solve solve: refl )
  subgoal for - lu
    using CLC
    unfolding clc-get-def l-get-def clc-rel-def in-br-conv clc-invar-def
```

by (*auto simp*: *refine-pw-simps split*: *prod.splits*)
done

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 −
from *CLC* **have** *LINV*: *l-invar l* **unfolding** *clc-rel-def in-br-conv clc-invar-def*
by *auto*

note *gap-relabel2-refine-aux*[*OF XCF CLC AM IdI IdI*]
also note *gap-relabel-aux-refine*[*OF CC LINV*]
finally show *?thesis* **by** *simp*
qed

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*) ∈ $unit\text{-}assn^k \rightarrow_a x\text{-}assn$
apply *sepref-to-hoare* **unfolding** *x-assn-def x-init-def*

by (*sep-auto heap*: *nf-init-rule*)

lemma *x-get-hnr*[*sepref-fr-rules*]:
(*uncurry Array.nth*, *uncurry* (*PR-CONST x-get*))
$\in$ *x-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *cap-assn*
apply *sepref-to-hoare*
unfolding *x-assn-def x-get-def* **by** (*sep-auto simp*: *refine-pw-simps*)

definition (**in** −) *x-add-impl x u* Δ $\equiv$ *do* {
xu $\leftarrow$ *Array.nth x u*;
x $\leftarrow$ *Array.upd u* (*xu*+Δ) *x*;
return x
}
lemma *x-add-hnr*[*sepref-fr-rules*]:
(*uncurry2 x-add-impl*, *uncurry2* (*PR-CONST x-add*))
$\in$ *x-assn*^{d} $*_a$ *node-assn*^{k} $*_a$ *cap-assn*^{k} $\rightarrow_a$ *x-assn*
apply *sepref-to-hoare*
unfolding *x-assn-def x-add-impl-def x-add-def*
by (*sep-auto simp*: *refine-pw-simps*)

6.3.3 Labeling by Array

definition *l-assn* $\equiv$ *is-nf N* (*0*::*nat*)
definition (**in** −) *l-init-impl N s cardV* $\equiv$ *do* {
l $\leftarrow$ *Array.new N* (*0*::*nat*);
l $\leftarrow$ *Array.upd s cardV l*;
return l
}
lemma *l-init-hnr*[*sepref-fr-rules*]:
(*l-init-impl N s*, (*PR-CONST l-init*)) $\in$ *nat-assn*^{k} $\rightarrow_a$ *l-assn*
apply *sepref-to-hoare*
unfolding *l-assn-def l-init-def l-init-impl-def*
by (*sep-auto heap*: *nf-init-rule*)

lemma *l-get-hnr*[*sepref-fr-rules*]:
(*uncurry Array.nth*, *uncurry* (*PR-CONST l-get*))
$\in$ *l-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
apply *sepref-to-hoare*
unfolding *l-assn-def l-get-def* **by** (*sep-auto simp*: *refine-pw-simps*)

lemma *l-get-rlx-hnr*[*sepref-fr-rules*]:
(*uncurry Array.nth*, *uncurry* (*PR-CONST l-get-rlx*))
$\in$ *l-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
apply *sepref-to-hoare*
unfolding *l-assn-def l-get-rlx-def* **by** (*sep-auto simp*: *refine-pw-simps*)

lemma *l-set-hnr*[*sepref-fr-rules*]:
(*uncurry2* (λ*a i x. Array.upd i x a*), *uncurry2* (*PR-CONST l-set*))

$\in$ *l-assn*^{d} $*_a$ *node-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *l-assn*
apply *sepref-to-hoare*
unfolding *l-assn-def l-set-def*
by (*sep-auto simp*: *refine-pw-simps split*: *prod.split*)

6.3.4 Label Frequency by Array

definition *cnt-assn* (f::*node*$\Rightarrow$*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 $\equiv$ *do* {
$a \leftarrow$ *Array.new* $(2*N)$ $(0::nat)$;
$a \leftarrow$ *Array.upd* 0 $(C-1)$ a;
$a \leftarrow$ *Array.upd* C 1 a;
return a
}

definition (**in** $-$) *cnt-incr-impl* a k $\equiv$ *do* {
freq $\leftarrow$ *Array.nth* a k;
$a \leftarrow$ *Array.upd* k $(freq+1)$ a;
return a
}

definition (**in** $-$) *cnt-decr-impl* a k $\equiv$ *do* {
freq $\leftarrow$ *Array.nth* a k;
$a \leftarrow$ *Array.upd* k $(freq-1)$ a;
return a
}

lemma *cnt-init-hnr*[*sepref-fr-rules*]: (*cnt-init-impl* N, *PR-CONST cnt-init*) $\in$ *nat-assn*^{k} $\rightarrow_a$ *cnt-assn*
apply *sepref-to-hoare*
unfolding *cnt-init-def cnt-init-impl-def cnt-assn-def*
by (*sep-auto simp*: *refine-pw-simps*)

lemma *cnt-get-hnr*[*sepref-fr-rules*]: (*uncurry Array.nth*, *uncurry* (*PR-CONST cnt-get*)) $\in$ *cnt-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *nat-assn*
apply *sepref-to-hoare*
unfolding *cnt-get-def cnt-assn-def*
by (*sep-auto simp*: *refine-pw-simps*)

lemma *cnt-incr-hnr*[*sepref-fr-rules*]: (*uncurry cnt-incr-impl*, *uncurry* (*PR-CONST cnt-incr*)) $\in$ *cnt-assn*^{d} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *cnt-assn*
apply *sepref-to-hoare*
unfolding *cnt-incr-def cnt-incr-impl-def cnt-assn-def*
by (*sep-auto simp*: *refine-pw-simps*)

lemma *cnt-decr-hnr*[*sepref-fr-rules*]: (*uncurry cnt-decr-impl*, *uncurry* (*PR-CONST*

cnt-decr)) ∈ *cnt-assn*^{d} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *cnt-assn*
  apply *sepref-to-hoare*
  unfolding *cnt-decr-def cnt-decr-impl-def cnt-assn-def*
  by (*sep-auto simp*: *refine-pw-simps*)

6.3.5 Combined Frequency Count and Labeling

definition *clc-assn* ≡ *cnt-assn* $\times_a$ *l-assn*

sepref-thm *clc-init-impl* **is** *PR-CONST clc-init* :: *nat-assn*^{k} $\rightarrow_a$ *clc-assn*
  unfolding *clc-init-def PR-CONST-def clc-assn-def*
  by *sepref*
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-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
  unfolding *clc-get-def PR-CONST-def clc-assn-def*
  by *sepref*
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-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
  unfolding *clc-get-rlx-def PR-CONST-def clc-assn-def*
  by *sepref*
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-assn*^{d} $*_a$ *node-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *clc-assn*
  unfolding *clc-set-def PR-CONST-def clc-assn-def*
  by *sepref*
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-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *bool-assn*
  unfolding *clc-has-gap-def PR-CONST-def clc-assn-def*
  by *sepref*
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-assn*^{d} $*_a$ *cf-assn*^{d} $*_a$ *edge-assn*^{k} $\rightarrow_a$ (*x-assn*$\times_a$*cf-assn*)
unfolding *push2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $*_a$ *cf-assn*^{k} $*_a$ *l-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *nat-assn*
unfolding *min-adj-label-def PR-CONST-def*
by *sepref*
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-assn*^{k} $*_a$ *cf-assn*^{k} $*_a$ *l-assn*^{d} $*_a$ *node-assn*^{k} $\rightarrow_a$ *l-assn*
unfolding *relabel2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $*_a$ *clc-assn*^{d} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *clc-assn*
unfolding *gap2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $*_a$ *cf-assn*^{k} $*_a$ *clc-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *nat-assn*
unfolding *min-adj-label-clc-def PR-CONST-def clc-assn-def*
by *sepref*

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-assn*^{k} $*_a$ *cf-assn*^{k} $*_a$ *clc-assn*^{d} $*_a$ *node-assn*^{k} $\rightarrow_a$ *clc-assn*

unfolding *clc-relabel2-def PR-CONST-def*
by *sepref*
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*)
:: $nat\text{-}assn^k *_a am\text{-}assn^k *_a cf\text{-}assn^k *_a clc\text{-}assn^d *_a node\text{-}assn^k \rightarrow_a clc\text{-}assn$
unfolding *gap-relabel2-def PR-CONST-def*
by *sepref*
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*)
:: $am\text{-}assn^k \rightarrow_a x\text{-}assn \times_a cf\text{-}assn$
unfolding *pp-init-xcf2-def PR-CONST-def*
by *sepref*
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 ($(u,v) \in E \cup E^{-1}$);
  assert ($u \neq v$);
  let Δ = *min* (*x u*) (*cf* (*u*,*v*));
  let Q = (*if* $v \neq s \wedge v \neq t \wedge x\ v = 0$ *then* $Q@[v]$ *else* Q);
  return (($x(u := x\ u - \Delta,\ v := x\ v + \Delta)$,*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*)
  unfolding *fifo-push-def fifo-push2-aux-def*

apply *refine-vcg*
apply (*vc-solve simp*: *xf-rel-def no-self-loop*)
subgoal for *u v*
unfolding *push-precond-def* **using** *cfE-of-ss-invE* **by** *auto*
subgoal for *u v*
proof −
assume [*simp*]: *Labeling c s t f l*
then interpret *Labeling c s t f l* **.**
assume *push-precond f l* (*u*, *v*)
then interpret *l′*: *push-effect-locale c s t f l u v* **by** *unfold-locales*
show *?thesis*
apply (*safe intro*!: *ext*)
using *l′.excess′-if l′.*Δ*-def l′.cf′-alt l′.uv-not-eq*(*1*)
by (*auto*)
qed
done

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 −
have *fifo-push2 x cf Qi ei* ≤ (*fifo-push2-aux x cf Qi ei*)
unfolding *fifo-push2-def fifo-push2-aux-def*
unfolding *x-get-def x-add-def cf-get-def cf-set-def q-enqueue-def*
unfolding *augment-edge-cf-def*
apply (*simp only*: *nres-monad-laws*)
apply *refine-vcg*
using *E-ss-VxV*
by *auto*

also note *fifo-push2-aux-refine*[*OF assms*]
finally show *?thesis* **.**
qed

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*
  unfolding *fifo-gap-relabel-aux-def fifo-gap-relabel-def relabel-def*
    gap-relabel-effect-def l-get-def q-enqueue-def
  apply (*simp only*: *Let-def nres-monad-laws*)
  apply *refine-vcg*
  by *auto*

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*)
  unfolding *fifo-gap-relabel2-def fifo-gap-relabel-aux-def*
  apply (*refine-vcg XCF AM CLC if-bind-cond-refine bind-refine'*)

apply *refine-dref-type*
apply (*vc-solve solve*: *refl*)
subgoal for - *lu*
using *CLC*
unfolding *clc-get-def l-get-def clc-rel-def in-br-conv clc-invar-def*
by (*auto simp*: *refine-pw-simps split*: *prod.splits*)
done

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 −
from *CLC* **have** *LINV*: *l-invar l*
unfolding *clc-rel-def in-br-conv clc-invar-def* **by** *auto*

note *fifo-gap-relabel2-refine-aux*[*OF XCF CLC AM IdI IdI IdI*]
also note *fifo-gap-relabel-aux-refine*[*OF CC LINV*]
finally show *?thesis* **by** *simp*
qed

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* (λ*v*. *cf-of f* (*u*,*v*) ≠ *0*) (*am u*))
using *assms cfE-of-ss-invE*
unfolding *is-adj-map-def Graph.E-def*
by *fastforce*

private 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 –
  from fun-relD[OF AMR] have AM: is-adj-map am
    unfolding is-adj-map-def
    by (auto simp: list-set-rel-def in-br-conv adjacent-nodes-def)

  from AM have AM-SS-V: set (am u) ⊆ V    {u}×set (am u) ⊆ E ∪ E⁻¹
    unfolding is-adj-map-def using E-ss-VxV by auto

  thm nfoldli-refine
  have nfoldli (am ui) cci ffi s0 ≤ ⇓Id (nfoldli (am ui) cc ?ff' s0)
    apply (refine-vcg FR)
    apply (rule list-rel-congD)
    apply refine-dref-type
    using CR
    apply vc-solve
    using AM-SS-V UR by auto
  also have nfoldli (am ui) cc ?ff' s0
          ≤⇓Id (FOREACHc (adjacent-nodes u) cc ?ff' s0)
    by (rule LFOc-refine[OF fun-relD[OF AMR UR]]; simp)
  also have FOREACHc (adjacent-nodes u) cc ?ff' s0
          ≤ FOREACHc {v . (u,v)∈cfE-of f} cc ff s0
    apply (subst am-nodes-as-filter[OF AM])
    apply (subst FOREACHc-filter-deforestation2)
    subgoal using AM unfolding is-adj-map-def by auto
    subgoal
      apply (rule eq-refl)
      apply ((fo-rule cong)+; (rule refl)?)
      subgoal
        using fun-relD[OF AMR IdI[of u]]
        by (auto simp: list-set-rel-def in-br-conv)
      done
    done
  finally show ?thesis using S0R by simp
qed

private definition dis-loop-aux am f₀ 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 f₀ (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)
```

```
  }) (f₀,l,Q)
}

private definition fifo-discharge-aux am f₀ l Q ≡ do {
  (u,Q) ← q-dequeue Q;
  assert (u∈V ∧ u≠s ∧ u≠t);

  (f,l,Q) ← dis-loop-aux am f₀ 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)
  unfolding fifo-discharge-aux-def fifo-discharge-def dis-loop-aux-def
  unfolding q-dequeue-def
  apply (simp only: nres-monad-laws)
  apply (refine-rcg adjacent-nodes-iterate-refine1[OF AM])
  apply refine-dref-type
  apply vc-solve
  subgoal
    using fun-relD[OF AM IdI[of hd Q]]
    unfolding list-set-rel-def by (auto simp: in-br-conv)
  done

private definition dis-loop-aux2 am f₀ 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 (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)
  }) (f₀,l,Q)
}

private lemma dis-loop-aux2-refine:
  shows dis-loop-aux2 am f₀ l Q u ≤⇓Id (dis-loop-aux am f₀ l Q u)
```

```
  unfolding dis-loop-aux2-def dis-loop-aux-def
  apply (intro ASSERT-refine-right ASSERT-refine-left; assumption?)
  apply (rule nfoldli-invar-refine[where
          I=λit1 it2 (f,-,-). ∀ v∈set it2. cf-of f (u,v) = cf-of f₀ (u,v)])
  apply refine-dref-type
  apply vc-solve
  apply (auto simp: pw-leof-iff refine-pw-simps fifo-push-def; metis)
  done

private 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) })
    (λv ((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 ×r Id ×r Id) (dis-loop-aux2 am f l Q u)
  unfolding dis-loop-aux3-def dis-loop-aux2-def
  unfolding x-get-def cf-get-def l-get-def
  apply (simp only: nres-monad-laws nfoldli-to-monadic)
  apply (refine-rcg)
  apply refine-dref-type
  using XF
  by (vc-solve simp: xf-rel-def in-br-conv)

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*)
  unfolding *dis-loop2-def dis-loop-aux3-def am-get-def*
  apply (*simp only*: *nres-monad-laws*)
  apply *refine-rcg*
  apply *refine-dref-type*
  apply (*vc-solve simp*: *CLC*)
  done

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 −
  have [*simp*]:
    ((*Id* $\times_r$ *clc-rel* $\times_r$ *Id*) *O* (*xf-rel* $\times_r$ *Id*)) = *xf-rel* $\times_r$ *clc-rel* $\times_r$ *Id*
    by (*auto simp*: *prod-rel-comp*)

  note *dis-loop2-refine-aux*[*OF IdI IdI IdI IdI IdI IdI CLC*]
  also note *dis-loop-aux3-refine*[*OF IdI IdI IdI IdI XF*]
  also note *dis-loop-aux2-refine*
  finally show *?thesis*
    by (*auto simp*: *conc-fun-chain monoD*[*OF conc-fun-mono*])
qed

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 −
have *fifo-discharge2 C am x cf clc Q*
≤⇓(*xf-rel* $\times_r$ *clc-rel* $\times_r$ *Id*) (*fifo-discharge-aux am f l Q*)
unfolding *fifo-discharge2-def fifo-discharge-aux-def*
unfolding *x-get-def*
apply (*simp only*: *nres-monad-laws*)
apply (*refine-rcg XCF CLC AM IdI*)
apply (*vc-solve simp*: *CC*)
subgoal unfolding *xf-rel-def in-br-conv* **by** *auto*
applyS *assumption*
done
also note *fifo-discharge-aux-refine*[*OF AM IdI IdI IdI*]
finally show *?thesis* **by** *simp*
qed

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 −
from *am-to-adj-nodes-refine*[*OF AM*] **have** *set* (*am s*) ⊆ *V*
using *adjacent-nodes-ss-V*
by (*auto simp*: *list-set-rel-def in-br-conv*)

hence *q-init am* ≤ *RETURN* (*filter* ((≠) *t*) (*am s*))
unfolding *q-init-def q-empty-def q-enqueue-def am-get-def*
apply (*refine-vcg nfoldli-rule*[**where** *I*=λ*l1* - *l*. *l* = *filter* ((≠) *t*) *l1*])
by *auto*
also have ...
≤ (*spec l*. *distinct l* ∧ *set l* = {*v* ∈ *V* − {*s*, *t*}. *excess pp-init-f v* ≠ *0*})
proof −
from *am-to-adj-nodes-refine*[*OF AM*]
have [*simp*]: *distinct* (*am s*) *set* (*am s*) = *adjacent-nodes s*
unfolding *list-set-rel-def*
by (*auto simp*: *in-br-conv*)

show *?thesis*
using *E-ss-VxV*
apply (*auto simp*: *pp-init-x-def adjacent-nodes-def*)
unfolding *Graph.E-def* **by** *auto*
qed
finally show *?thesis* **.**
qed

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*)
unfolding *fifo-push-relabel-aux-def fifo-push-relabel-def*
unfolding *l-init-def pp-init-l-def q-is-empty-def bind-to-let-conv*
apply (*rule specify-left*[*OF init-C-correct*[*OF AM*]])
apply (*refine-rcg q-init-correct*[*OF AM*])
apply *refine-dref-type*

```
  apply vc-solve
  done


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 −
  {
    fix f l n
    assume Height-Bounded-Labeling c s t f l
    then interpret Height-Bounded-Labeling c s t f l .
    have G1: flow-of-cf cf = f by (rule fo-rg-inv)
    have G2: RPreGraph c s t cf by (rule is-RPreGraph)
    note G1 G2
  } note AUX1=this


  have fifo-push-relabel2 am
        ≤ ⇓(br (flow-of-cf) (RPreGraph c s t)) (fifo-push-relabel-aux am)
    unfolding fifo-push-relabel2-def fifo-push-relabel-aux-def
    apply (refine-rcg)
    apply (refine-dref-type)
    apply (vc-solve simp: AM am-to-adj-nodes-refine[OF AM])
    subgoal using AUX1 by (auto simp: in-br-conv xf-rel-def AM)
    done
  also note fifo-push-relabel-aux-refine[OF AM]
  finally show ?thesis .
qed

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
  }
  unfolding fifo-push-relabel-init2-def fifo-push-relabel-run2-def
    fifo-push-relabel2-def pp-init-xcf2-def pp-init-xcf2′-def
```

cf-init-def
by *simp*

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-α* ≡ λ(*L*,*R*). *L*@*rev R*
definition (**in** −) *q-empty-impl* ≡ ([],[])
definition (**in** −) *q-is-empty-impl* ≡ λ(*L*,*R*). *is-Nil L* ∧ *is-Nil R*
definition (**in** −) *q-enqueue-impl* ≡ λ*x* (*L*,*R*). (*L*,*x*#*R*)
definition (**in** −) *q-dequeue-impl*
≡ λ(*x*#*L*,*R*) ⇒ (*x*,(*L*,*R*)) | ([],*R*) ⇒ *case rev R of* (*x*#*L*) ⇒ (*x*,(*L*,[]))

lemma *q-empty-impl-correct*[*simp*]: *q-α q-empty-impl* = []
by (*auto simp*: *q-α-def q-empty-impl-def*)

lemma *q-enqueue-impl-correct*[*simp*]: *q-α* (*q-enqueue-impl x Q*) = *q-α Q* @ [*x*]
by (*auto simp*: *q-α-def q-enqueue-impl-def split*: *prod.split*)

lemma *q-is-empty-impl-correct*[*simp*]: *q-is-empty-impl Q* $\longleftrightarrow$ *q-α Q* = []
  unfolding *q-α-def q-is-empty-impl-def*
  by (*cases Q*) (*auto split*: *list.splits*)

lemma *q-dequeue-impl-correct-aux*:
  $[\![$*q-α Q* = *x#xs*$]\!] \Longrightarrow$ *apsnd q-α* (*q-dequeue-impl Q*) = (*x*,*xs*)
  unfolding *q-α-def q-dequeue-impl-def*
  by (*cases Q*) (*auto split*!: *list.split*)

lemma *q-dequeue-impl-correct*[*simp*]:
  assumes *q-dequeue-impl Q* = (*x*,*Q'*)
  assumes *q-α Q* ≠ []
  shows *x* = *hd* (*q-α Q*) **and** *q-α Q'* = *tl* (*q-α Q*)
  using *assms q-dequeue-impl-correct-aux*[*of Q*]
  by − (*cases q-α Q*; *auto*)+

definition *q-assn* ≡ *pure* (*br q-α* (λ-. *True*))

lemma *q-empty-impl-hnr*[*sepref-fr-rules*]:
  (*uncurry0* (*return q-empty-impl*), *uncurry0 q-empty*) ∈ $unit\text{-}assn^k \rightarrow_a q\text{-}assn$
  apply (*sepref-to-hoare*)
  unfolding *q-assn-def q-empty-def pure-def*
  by (*sep-auto simp*: *in-br-conv*)

lemma *q-is-empty-impl-hnr*[*sepref-fr-rules*]:
  (*return o q-is-empty-impl*, *q-is-empty*) ∈ $q\text{-}assn^k \rightarrow_a bool\text{-}assn$
  apply (*sepref-to-hoare*)
  unfolding *q-assn-def q-is-empty-def pure-def*
  by (*sep-auto simp*: *in-br-conv*)

lemma *q-enqueue-impl-hnr*[*sepref-fr-rules*]:
  (*uncurry* (*return oo q-enqueue-impl*), *uncurry* (*PR-CONST q-enqueue*))
    ∈ $nat\text{-}assn^k *_a q\text{-}assn^d \rightarrow_a q\text{-}assn$
  apply (*sepref-to-hoare*)
  unfolding *q-assn-def q-enqueue-def pure-def*
  by (*sep-auto simp*: *in-br-conv refine-pw-simps*)

lemma *q-dequeue-impl-hnr*[*sepref-fr-rules*]:
  (*return o q-dequeue-impl*, *q-dequeue*) ∈ $q\text{-}assn^d \rightarrow_a nat\text{-}assn \times_a q\text{-}assn$
  apply (*sepref-to-hoare*)
  unfolding *q-assn-def q-dequeue-def pure-def prod-assn-def*
  by (*sep-auto simp*: *in-br-conv refine-pw-simps split*: *prod.split*)

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*)
unfolding *fifo-push2-def PR-CONST-def*
by *sepref*
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*
unfolding *fifo-gap-relabel2-def PR-CONST-def*
by *sepref*
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*
unfolding *dis-loop2-def PR-CONST-def*
by *sepref*
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*
unfolding *fifo-discharge2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $\rightarrow_a$ *nat-assn*
unfolding *init-C-def PR-CONST-def*
by *sepref*
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-assn*^{k} $\rightarrow_a$ *q-assn*
unfolding *q-init-def PR-CONST-def*
by *sepref*
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-assn*^{k} $*_a$ *cf-assn*^{d} $\rightarrow_a$ *x-assn* $\times_a$ *cf-assn*
unfolding *pp-init-xcf2'-def PR-CONST-def*
by *sepref*
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-assn*^{k} $*_a$ *cf-assn*^{d} $\rightarrow_a$ *cf-assn*
unfolding *fifo-push-relabel-run2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $\rightarrow_a$ *cf-assn*
unfolding *fifo-push-relabel-init2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $\rightarrow_a$ *cf-assn*
unfolding *fifo-push-relabel2-alt PR-CONST-def*
by *sepref*
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
    <λcfi. ∃A cf.
        am-assn am ami * cf-assn cf cfi
      * ↑(isMaxFlow (flow-of-cf cf) ∧ RGraph-Impl c s t N cf)>t
proof −
  note fifo-push-relabel2-refine[OF AM]
  also note fifo-push-relabel-correct
  finally have R1:
    fifo-push-relabel2 am
    ≤ ⇓ (br flow-of-cf (RPreGraph c s t)) (SPEC isMaxFlow) .

  have [simp]: nofail (⇓R (RES X)) for R X by (auto simp: refine-pw-simps)

  note R2 = fifo-push-relabel-impl.refine[
            OF Network-Impl-axioms, to-hnr, unfolded autoref-tag-defs]
  note R3 = hn-refine-ref[OF R1 R2, of ami]
  note R4 = R3[unfolded hn-ctxt-def pure-def, THEN hn-refineD, simplified]

  note RGII = rgraph-and-network-impl-imp-rgraph-impl[OF
    RPreGraph.maxflow-imp-rgraph
    Network-Impl-axioms
      ]

  show ?thesis
    by (sep-auto
        heap: R4
        simp: RGII
        simp: pw-le-iff refine-pw-simps in-br-conv)
qed
```

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 −
  interpret *Network c s t* **by** *fact*
  interpret *Network-Impl c s t N* **using** *VN* **by** *unfold-locales*

  from *ABS-PS* **have** [*simp*]: *am u* = [] **if** *u*≥*N* **for** *u*
    unfolding *is-adj-map-def*
    using *E-ss-VxV VN that*
    apply (*subgoal-tac u*∉*V*)
    by (*auto simp del*: *inV-less-N*)

  show *?thesis*
    unfolding *fifo-push-relabel-impl-tab-am-def*
    apply *vcg*
    apply (*rule Hoare-Triple.cons-rule*[
            OF - - fifo-push-relabel-impl-correct[*OF ABS-PS*]])
    subgoal unfolding *am-assn-def is-nf-def* **by** *sep-auto*
    apply (*rule ent-refl*)
    subgoal by *sep-auto*
    done
qed

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

  unfolding fifo-push-relabel-def
  using prepareNet-correct[of el s t]
  by (sep-auto simp: ln-rel-def in-br-conv)
```

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))
      }
  }
  unfolding fifo-push-relabel-def fifo-push-relabel-prepare-impl-def
    fifo-push-relabel-impl-tab-am-def fifo-push-relabel-impl-def
  by (auto split: option.split)
```

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 −
  {
    fix cf N
    assume RGraph-Impl (ln-α el) s t N cf
    then interpret RGraph (ln-α el) s t cf by (rule RGraph-Impl.axioms)
    have f = flow-of-cf cf unfolding f-def by simp
  } note aux=this

  show ?thesis
    unfolding fifo-push-relabel-compute-flow-val-def
    by (sep-auto simp: Network.is-max-flow-val-def aux)

qed

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 −
  have[*simp*]: *am v* = [] **if** *v*∉*V* **for** *v*
  proof −
    from *that* **have** *adjacent-nodes v* = {}
      unfolding *adjacent-nodes-def* **using** *E-ss-VxV* **by** *auto*
    thus *?thesis* **using** *am-to-adj-nodes-refine*[*OF AM*]

    by (*auto simp*: *list-set-rel-def in-br-conv*)
  qed
  show *?thesis*
    unfolding *n-init-def rtf-init-n-def*
    by (*auto*
        simp: *pw-le-iff refine-pw-simps list-set-autoref-empty*
        simp: *am-to-adj-nodes-refine*[*OF AM*])
qed

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)
  unfolding n-at-end-def n-get-hd-def
  apply (simp only: nres-monad-laws)
  apply (cases ni u)
  subgoal
    using fun-relD[OF SR SU] SU UIV fNR
    by (auto simp: list-set-rel-def in-br-conv pw-le-iff refine-pw-simps)

  subgoal for v vs
    using fun-relD[OF SR SU] SU UIV
    using fSR[OF IdI[of v], of vs]
    apply (clarsimp
        simp: list-set-rel-def in-br-conv pw-le-iff refine-pw-simps
        split: option.splits)
    by fastforce
  done

lemma xf-rel-RELATES[refine-dref-RELATES]: RELATES xf-rel
  by (auto simp: RELATES-def)

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 ×r Id ×r (nat-rel → ⟨nat-rel⟩list-set-rel)) (discharge f l n u)
  unfolding discharge2-def discharge-def
  apply (rewrite in monadic-WHILEIT - - ¤ - vcg-intro-frame)
  apply refine-rcg
  apply (vc-solve simp: A NR)
  subgoal by (simp add: xf-rel-def x-get-def)
  subgoal for f l n x cf ni
    apply (subst vcg-rem-frame)
    unfolding n-reset-def cf-get-def l-get-def n-move-next-def
    apply (simp only: nres-monad-laws)
    apply (rule discharge-structure-refine-aux; (refine-vcg AM)?; (assumption)?)
```

```
    apply (vc-solve simp: fun-relD fun-relD[OF AM])
    subgoal for v vs unfolding xf-rel-def Graph.E-def by auto
    subgoal for v vs by (auto simp: list-set-rel-def in-br-conv)
    done
  done
```

8.2.2 Initialization of Queue

```
lemma V-is-adj-nodes: V = { v . adjacent-nodes v ≠ {} }
  unfolding V-def adjacent-nodes-def by auto

definition init-CQ am ≡ do {
  let cardV=0;
  let Q=[];
  nfoldli [0..<N] (λ-. True) (λv (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})
  unfolding init-CQ-def
  apply (refine-vcg
      nfoldli-rule[where
        I=λl1 - (C,Q).
            C = card (V∩set l1) ∧ distinct Q ∧ set Q = (V∩set l1)−{s,t} ]
      )
  apply (clarsimp-all simp: am-to-adj-nodes-refine[OF AM])
  using V-ss by (auto simp: upt-eq-lel-conv Int-absorb2)
```

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 −
define *s-rel*
:: (- × (
capacity-impl flow
× (*nat*⇒*nat*)
× (*node*⇒*node set*)
× *node list*
× *node list*)) *set*
where *s-rel* ≡
xf-rel
$\times_r$ *Id*
$\times_r$ (*nat-rel* → ⟨*nat-rel*⟩*list-set-rel*)
$\times_r$ *br rev* (λ-. *True*)
$\times_r$ *Id*

have [*refine-dref-RELATES*]: *RELATES s-rel* **unfolding** *RELATES-def* **..**

```
{
  fix f l n
  assume neighbor-invar c s t f l n
  then interpret neighbor-invar c s t f l n .
  have G1: flow-of-cf cf = f by (rule fo-rg-inv)
  have G2: RPreGraph c s t cf by (rule is-RPreGraph)
  note G1 G2
} note AUX1=this

have AUXR: do {
    (cardV, L-right) ← init-CQ am;
    xcf ← pp-init-xcf2 am;
    l ← l-init cardV;
    n ← n-init am;
    Fi L-right xcf l n
  }
  ≤ ⇓R (do {
    L-right ← spec l. distinct l ∧ set l = V − {s, t};
    F L-right
  })

if ⋀L-right xcf n.
  ⟦ (xcf,pp-init-f)∈xf-rel; (n,rtf-init-n) ∈ nat-rel → ⟨nat-rel⟩list-set-rel ⟧
  ⟹ Fi L-right xcf pp-init-l n ≤ ⇓R (F L-right)
for Fi F R
  unfolding l-init-def
  apply (rule refine2specI)
  supply pp-init-xcf2-refine
        [OF AM, unfolded conc-fun-RETURN, THEN order-trans, refine-vcg]
  supply n-init-refine[OF AM,THEN order-trans, refine-vcg]
  apply (refine-vcg AM V-ss)
  apply clarsimp
  subgoal for L-right x cf n
    using that[of (x,cf) n L-right]
    unfolding pp-init-l-def
    by (clarsimp simp: pw-le-iff refine-pw-simps; meson)
  done
show ?thesis
  unfolding relabel-to-front2-def relabel-to-front-def Let-def l-get-def
  apply (simp only: nres-monad-laws)
  apply (rule AUXR)
  apply (refine-rcg)
  apply refine-dref-type
  unfolding s-rel-def
  apply (vc-solve
      simp: in-br-conv rev-append-eq xf-rel-def AUX1 fun-relD
      simp: am-to-adj-nodes-refine[OF AM])
  done
```

qed

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*) ∈ $am\text{-}assn^k \rightarrow_a n\text{-}assn$
apply *sepref-to-hoare*
unfolding *am-assn-def n-assn-def n-init-impl-def n-init-def*
by (*sep-auto*)

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*))
$\in n\text{-}assn^k *_a node\text{-}assn^k \rightarrow_a bool\text{-}assn$
apply *sepref-to-hoare* **unfolding** *n-at-end-impl-def n-at-end-def n-assn-def*
by (*sep-auto simp*: *refine-pw-simps split*: *list.split*)

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-assn*^{k} $*_a$ *node-assn*^{k} $\rightarrow_a$ *node-assn*
apply *sepref-to-hoare* **unfolding** *n-get-hd-impl-def n-get-hd-def n-assn-def*
by (*sep-auto simp*: *refine-pw-simps*)

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-assn*^{d} $*_a$ *node-assn*^{k} $\rightarrow_a$ *n-assn*
apply *sepref-to-hoare*
unfolding *n-move-next-impl-def n-move-next-def n-assn-def*
by (*sep-auto simp*: *refine-pw-simps*)

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-reset-impl*, *uncurry2* (*PR-CONST n-reset*))
$\in$ *am-assn*^{k} $*_a$ *n-assn*^{d} $*_a$ *node-assn*^{k} $\rightarrow_a$ *n-assn*
apply *sepref-to-hoare*
unfolding *n-reset-impl-def n-reset-def n-assn-def am-assn-def*
by (*sep-auto simp*: *refine-pw-simps*)

8.3.2 Discharge

sepref-thm *discharge-impl* **is** *uncurry5* (*PR-CONST discharge2*)
:: *am-assn*^{k} $*_a$ *x-assn*^{d} $*_a$ *cf-assn*^{d} $*_a$ *l-assn*^{d} $*_a$ *n-assn*^{d} $*_a$ *node-assn*^{k}
$\rightarrow_a$ (*x-assn* $\times_a$ *cf-assn*) $\times_a$ *l-assn* $\times_a$ *n-assn*
unfolding *discharge2-def PR-CONST-def*
by *sepref*
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-assn*^{k} $\rightarrow_a$ *nat-assn* $\times_a$ *list-assn nat-assn*
unfolding *init-CQ-def PR-CONST-def*
apply (*rewrite HOL-list.fold-custom-empty*)
by *sepref*
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*) :: $am\text{-}assn^k \rightarrow_a cf\text{-}assn$
 unfolding *relabel-to-front2-def PR-CONST-def*
 supply [[*goals-limit = 1*]]
 apply (*rewrite* **in** *Let* [] - *HOL-list.fold-custom-empty*)
 apply (*rewrite* **in** [-] *HOL-list.fold-custom-empty*)
 by *sepref*
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-assn am ami*>
    relabel-to-front-impl c s t N ami
   <λ*cfi*. $\exists_A$*cf*. *cf-assn cf cfi*
      ∗ ↑(*isMaxFlow* (*flow-of-cf cf*) ∧ *RGraph-Impl c s t N cf*)$>_t$
 proof −
  note *relabel-to-front2-refine*[*OF AM*]
  also note *relabel-to-front-correct*
  finally have *R1*:
   relabel-to-front2 am
   ≤ ⇓ (*br flow-of-cf* (*RPreGraph c s t*)) (*SPEC isMaxFlow*) **.**

  have [*simp*]: *nofail* (⇓*R* (*RES X*)) **for** *R X* **by** (*auto simp*: *refine-pw-simps*)

  note *R2* = *relabel-to-front-impl.refine*[
   OF Network-Impl-axioms, *to-hnr*, *unfolded autoref-tag-defs*]
  note *R3* = *hn-refine-ref*[*OF R1 R2*, *of ami*]
  note *R4* = *R3*[*unfolded hn-ctxt-def pure-def*, *THEN hn-refineD*, *simplified*]

  note *RGII* = *rgraph-and-network-impl-imp-rgraph-impl*[*OF*
   RPreGraph.maxflow-imp-rgraph
   Network-Impl-axioms
    ]

  show *?thesis*
   by (*sep-auto heap*: *R4 simp*: *pw-le-iff refine-pw-simps in-br-conv RGII*)
 qed

```
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
    <λcfi. ∃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 −
  interpret Network c s t by fact
  interpret Network-Impl c s t N using VN by unfold-locales

  from ABS-PS have [simp]: am u = [] if u≥N for u
    unfolding is-adj-map-def
    using E-ss-VxV VN that
    apply (subgoal-tac u∉V)
    by (auto simp del: inV-less-N)

  show ?thesis
    unfolding relabel-to-front-impl-tab-am-def
    apply vcg
    apply (rule
        Hoare-Triple.cons-rule[OF - - relabel-to-front-impl-correct[OF ABS-PS]])
    subgoal unfolding am-assn-def is-nf-def by sep-auto
    subgoal unfolding cf-assn-def by sep-auto
    done
qed

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)
    ∗ (∃A cf. asmtx-assn N int-assn cf cfi
          ∗ ↑(RGraph-Impl c s t N cf
            ∧ Network.isMaxFlow c s t (Network.flow-of-cf c cf)))
    ∗ ↑(Graph.is-adj-map c am)
  >t

  unfolding relabel-to-front-def
  using prepareNet-correct[of el s t]
  by (sep-auto simp: ln-rel-def in-br-conv)


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.