Formalizing Push-Relabel Algorithms

Peter Lammich and S. Reza Seifidgar

August 16, 2018

Abstract

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

Our formal proof of the abstract algorithms closely follows a standard textbook proof, and is accessible even without being an expert in Isabelle/HOL—the interactive theorem prover used for the formalization.
Contents

1 Introduction 5

2 Generic Push Relabel Algorithm 5
   2.1 Labeling .................................................. 5
   2.2 Basic Operations ........................................... 6
      2.2.1 Augmentation of Edges ............................. 6
      2.2.2 Push Operation ................................. 10
      2.2.3 Relabel Operation ........................... 11
      2.2.4 Initialization ................................. 11
   2.3 Abstract Correctness .................................... 11
      2.3.1 Maintenance of Invariants ....................... 12
      2.3.2 Maxflow on Termination ......................... 14
   2.4 Convenience Lemmas .................................... 14
   2.5 Complexity ................................................ 18
      2.5.1 Auxiliary Lemmas ................................. 18
      2.5.2 Height Bound ...................................... 18
      2.5.3 Formulation of the Abstract Algorithm .......... 23
      2.5.4 Saturating and Non-Saturating Push Operations .. 24
      2.5.5 Refined Labeled Transition System ............... 25
      2.5.6 Bounding the Relabel Operations ................ 28
      2.5.7 Bounding the Saturating Push Operations ........ 28
      2.5.8 Bounding the Non-Saturating Push Operations ... 33
      2.5.9 Assembling the Final Theorem ................... 38
   2.6 Main Theorem: Correctness and Complexity ............ 39
   2.7 Convenience Tools for Implementation ................ 39
   2.8 Gap Heuristics ............................................ 42
      2.8.1 Termination with Gap Heuristics ................ 43

3 FIFO Push Relabel Algorithm 45
   3.1 Implementing the Discharge Operation ................. 45
   3.2 Main Algorithm ........................................... 51

4 Topological Ordering of Graphs 52
   4.1 List-Before Relation ................................... 52
   4.2 Topological Ordering ................................... 54

5 Relabel-to-Front Algorithm 55
   5.1 Admissible Network ..................................... 56
   5.2 Neighbor Lists .......................................... 59
   5.3 Discharge Operation .................................... 61
   5.4 Main Algorithm .......................................... 65
6 Tools for Implementing Push-Relabel Algorithms

6.1 Basic Operations ................................................. 70
  6.1.1 Excess Map ................................................. 70
  6.1.2 Labeling ..................................................... 70
  6.1.3 Label Frequency Counts for Gap Heuristics .............. 71

6.2 Refinements to Basic Operations ............................... 71
  6.2.1 Explicit Computation of the Excess ..................... 71
  6.2.2 Algorithm to Compute Initial Excess and Flow ........ 72
  6.2.3 Computing the Minimal Adjacent Label ................. 74
  6.2.4 Refinement of Relabel ..................................... 77
  6.2.5 Refinement of Push ....................................... 77
  6.2.6 Adding frequency counters to labeling ............... 79
  6.2.7 Refinement of Gap-Heuristics ........................... 81

6.3 Refinement to Efficient Data Structures ..................... 84
  6.3.1 Registration of Abstract Operations .................... 84
  6.3.2 Excess by Array .......................................... 84
  6.3.3 Labeling by Array ........................................ 85
  6.3.4 Label Frequency by Array ............................... 86
  6.3.5 Combined Frequency Count and Labeling ............... 87
  6.3.6 Push ....................................................... 88
  6.3.7 Relabel .................................................... 88
  6.3.8 Gap-Relabel .............................................. 88
  6.3.9 Initialization ............................................ 89

7 Implementation of the FIFO Push/Relabel Algorithm .......... 89

7.1 Basic Operations ................................................. 89
  7.1.1 Queue ...................................................... 90

7.2 Refinements to Basic Operations ............................... 90
  7.2.1 Refinement of Push ....................................... 90
  7.2.2 Refinement of Gap-Relabel ................................ 92
  7.2.3 Refinement of Discharge .................................. 93
  7.2.4 Computing the Initial Queue ............................. 98
  7.2.5 Refining the Main Algorithm ............................ 99

7.3 Separating out the Initialization of the Adjacency Matrix . 101

7.4 Refinement To Efficient Data Structures ..................... 102
  7.4.1 Registration of Abstract Operations .................... 102
  7.4.2 Queue by Two Stacks ....................................... 102
  7.4.3 Push ....................................................... 104
  7.4.4 Gap-Relabel .............................................. 104
  7.4.5 Discharge .................................................. 104
  7.4.6 Computing the Initial State ............................. 104
  7.4.7 Main Algorithm ........................................... 105

7.5 Combining the Refinement Steps ............................... 106
7.6 Combination with Network Checker and Main Correctness

7.6.1 Justification of Splitting into Prepare and Run Phase 108

7.7 Usage Example: Computing Maxflow Value 109

8 Implementation of Relabel-to-Front 110

8.1 Basic Operations 110

8.1.1 Neighbor Lists 110

8.2 Refinement to Basic Operations 111

8.2.1 Discharge 111

8.2.2 Initialization of Queue 113

8.2.3 Main Algorithm 113

8.3 Refinement to Efficient Data Structures 116

8.3.1 Neighbor Lists by Array of Lists 116

8.3.2 Discharge 117

8.3.3 Initialization of Queue 117

8.3.4 Main Algorithm 118

8.4 Combination with Network Checker and Correctness 118

9 Conclusion 120
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 Maxflow-Lib,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 $l : \text{node} \Rightarrow \text{nat}$ 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 $\Rightarrow$ nat

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

Generalizing validity to paths

lemma gen-valid: $l(u) \leq l(x) + \text{length } p \text{ if } \text{cf.isPath } u \text{ } p \text{ } x$
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: \( \neg \text{isAugmentingPath } p \)

**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 \( \geq \) 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 \( \forall u \in V - \{s,t\}. \text{excess } f u = 0 \)
- 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 \( \Rightarrow \) -

- where augment-edge \( f \equiv \lambda (u,v) \Delta. \)
- if \( (u,v) \in E \) then \( f( (u,v) := f( (u,v) + \Delta ) \)
- else if \( (v,u) \in E \) then \( f( (v,u) := f( (v,u) - \Delta ) \)
- else \( f \)

**lemma** augment-edge-zero[simp]; augment-edge \( f 0 = f \)

**unfolding** augment-edge-def by (auto split: prod.split)
lemma augment-edge-same[simp]: $e \in E \implies$ augment-edge $f e \Delta e = f e + \Delta$
unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-other[simp]: $[\exists e \in E; e' \neq e] \implies$ augment-edge $f e \Delta e' = f e'$
unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-rev-same[simp]:
$v, u \in E \implies$ augment-edge $f (u, v) \Delta (v, u) = f (v, u) - \Delta$
using no-parallel-edge
unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-rev-other[simp]:
$[(u, v) / \in E; e' \neq (v, u)] \implies$ augment-edge $f (u, v) \Delta e' = f e'$
unfolding augment-edge-def by (auto split!: prod.splits)

lemma augment-edge-cf[simp]: $(u, v) \in E \cup E^{-1} \implies$
$\text{cf-of } (\text{augment-edge } f (u, v) \Delta) =$
$(\text{cf-of } f) (u, v) := \text{cf-of } f (u, v) - \Delta, (v, u) := \text{cf-of } f (v, u) + \Delta)$
apply (intro ext; cases $(u, v) \in E$)

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-parallel-edge residualGraph-def)
subgoal by (simp
split!: if-splits prod.splits
add: residualGraph-def augment-edge-def)
done

done

lemma augment-edge-cf': $(u, v) \in \text{cfE-of } f \implies$
$\text{cf-of } (\text{augment-edge } f (u, v) \Delta) =$
$(\text{cf-of } f) (u, v) := \text{cf-of } f (u, v) - \Delta, (v, u) := \text{cf-of } f (v, u) + \Delta)$
proof
assume $(u, v) \in \text{cfE-of } f$
hence $(u, v) \in E \cup E^{-1}$ using cfE-of-ss-invE ..
thus ?thesis by simp
qed

The effect of augmenting an edge on the residual graph

7
definition (in -) augment-edge-cf :: flow ⇒ - where
augment-edge-cf cf
≡ λ(u,v) ∆. (cf)((u,v) := cf(u,v) − ∆, (v,u) := cf(v,u) + ∆)

lemma cf-of-augment-edge:
assumes A: (u,v) ∈ cfE-of f
shows cf-of (augment-edge f (u,v) ∆) = augment-edge-cf (cf-of f) (u,v) ∆
proof
  show cf-of (augment-edge f (u,v) ∆) = augment-edge-cf (cf-of f) (u,v) ∆
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) ∆) ⊆ 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 ∆ 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 ≤ ∆; ∆ ≤ cf (u,v); ∆ ≤ excess f u] ] ⇒ Preflow c s t (augment-edge f (u,v) ∆)
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 ≤ ∆  ∆ ≤ cf (u,v)  ∆ ≤ excess f u
    fix v'
    assume V': v'∈ V  v'≠s  v'≠t

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


next

assume $\Delta \neq 0$ with $(\theta \leq \Delta)$ have $\theta < \Delta$ by auto
with $\Delta \leq cf \ (u,v)$ have $(u,v) \in cf \cdot E$ unfolding Graph.$E$-def by auto

show ?thesis

proof (cases)
assume [simp]: $(u,v) \in E$

hence $AE \cdot$ augment-edge $f \ (u,v) \Delta = f \ (u,v) + \Delta$
unfolding augment-edge-def by auto

have 1: $\forall e \in$ outgoing $v'$. augment-edge $f \ (u,v) \Delta e = f e$ if $v' \neq u$
using that unfolding outgoing-def $AE$ by auto

have 2: $\forall e \in$ incoming $v'$. augment-edge $f \ (u,v) \Delta e = f e$ if $v' \neq v$
using that unfolding incoming-def $AE$ by auto

from $(u,v) \in E$ no-self-loop have $u \neq v$ by blast

{} assume $v' \neq u \quad v' \neq v$

with 1 2 $V'$ no-deficient-nodes have ?thesis by auto

} moreover {
assume [simp]: $v' = v$

have sum (augment-edge $f \ (u, v) \Delta$) (outgoing $v'$)

= sum $f$ (outgoing $v$)
using 1 $(u \neq v) \quad V'$ by auto
also have $\ldots \leq$ sum $f$ (incoming $v$)
using $V'$ no-deficient-nodes by auto
also have $\ldots \leq$ sum (augment-edge $f \ (u, v) \Delta$) (incoming $v$)
apply (rule sum-mono)
using $(\theta \leq \Delta)$
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 $f \ (u, v) \Delta$) (incoming $v'$)

= sum $f$ (incoming $u$)
using 2 $(u \neq v)$ by auto

have $(u,v) \in$ outgoing $u$ using $(u,v) \in E$

by (auto simp: outgoing-def)

note $AUX = \text{sum.remove[OF - this, simplified]}

have $A2$: sum (augment-edge $f \ (u, v) \Delta$) (outgoing $u$)

= sum $f$ (outgoing $u$) + $\Delta$
using $AUX$[of augment-edge $f \ (u,v) \Delta$] $AUX$[of $f$] by auto

from $A1 \ A2 \ (\Delta \leq \text{excess } f \ u$ no-deficient-nodes $V'$ have ?thesis

unfolding excess-def by auto

} ultimately show ?thesis by blast

next

assume [simp]: $(u,v) \notin E$

hence [simp]: $(v,u) \in E$ using $cf \cdot E$-ss-inv $E \ (u,v) \in cf \cdot E$ by auto

from $(u,v) \notin E$ $(v,u) \in E$ have $u \neq v$ by blast
\( AE: \text{augment-edge } f (u,v) \Delta = f (v,u) - \Delta \)

unfolding augment-edge-def by simp

have 1: \( \forall e \in \text{outgoing } v'. \text{augment-edge } f (u,v) \Delta e = f e \) if \( v' \neq v \)
using that unfolding outgoing-def \( AE \) by auto

have 2: \( \forall e \in \text{incoming } v'. \text{augment-edge } f (u,v) \Delta e = f e \) if \( v' \neq u \)
using that unfolding incoming-def \( AE \) by auto

\[
\begin{align*}
\text{have } & \text{1 2 } V' \text{ no-deficient-nodes have } \text{?thesis by auto } \\
\text{moreover } & \{ \\
& \text{assume } [\text{simp}]: v' = u \\
& \text{have } A1: \text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{outgoing } v') \\
& \quad = \text{sum } f (\text{outgoing } u) \\
& \quad \text{using } I (u \neq v) \text{ V' by auto } \\
\text{have } & (v,u) \in \text{incoming } u \\
& \text{using } [(v,u) \in E] \text{ by (auto simp: incoming-def)} \\
\text{note } & \text{AUX } = \text{sum.remove}\{\text{OF - this, simplified} \} \\
\text{have } & A2: \text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{incoming } u) \\
& \quad = \text{sum } f (\text{incoming } u) - \Delta \\
& \quad \text{using } \text{AUX}\{\text{of augment-edge } f (u,v) \Delta\} \text{ AUX}\{\text{of } f\} \text{ by auto } \\
\text{from } & A1 \ A2 \langle \Delta \leq \text{excess } f \rangle \text{ no-deficient-nodes } V' \text{ have } \text{?thesis unfolding excess-def by auto } \\
\text{moreover } & \{ \\
& \text{assume } [\text{simp}]: v' = v \\
& \text{have } \text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{outgoing } v') \\
& \quad \leq \text{sum } f (\text{outgoing } v') \\
& \quad \text{apply (rule sum-mono)} \\
& \quad \text{using } (0 < \Delta) \\
& \quad \text{by (auto simp: augment-edge-def)} \\
& \text{also have } \ldots \leq \text{sum } f (\text{incoming } v) \\
& \quad \text{using no-deficient-nodes } V' \text{ by auto } \\
& \text{also have } \ldots \leq \text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{incoming } v') \\
& \quad \text{using } 2 (u \neq v) \text{ by auto } \\
& \text{finally have } \text{?thesis by simp} \\
\text{} & \text{ultimately show } \text{?thesis by blast} \\
\text{qed} \\
\text{qed} \\
\text{done} \\
\text{end — Network with Preflow} \\
\end{align*}
\]

### 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 \(f l\)
≡ \(\lambda(u,v). \) excess \(f u > 0 \land (u,v) \in cfE-of \ f \land l_u = l_v + 1\)

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 \(f\)
≡ \(\lambda(u,v). \) augment-edge \(f (u,v) (\min (excess \ f u) (cf-of \ f (u,v)))\)

lemma push-precondI [intro?]:
\[ \[ \text{excess } f u > 0; (u,v) \in cfE-of \ f; l_u = l_v + 1 \] \] ⇒ 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 \(f l u\)
≡ \(u \neq t \land \text{excess } f u > 0 \land (\forall v. (u,v) \in cfE-of \ f \rightarrow l_u \neq l_v + 1)\)

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

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

2.2.4 Initialization

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

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

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

definition pp-init-l \(\equiv (\lambda x. \ 0)(s := \text{card } V)\)

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-locale;

  \((\text{auto simp: pp-init-f-def pp-init-l-def cap-non-negative; fail)}\)

  \((\text{intro ballI})?)\))

**proof**

  - fix \(v\)
  
    assume \(v \in V \setminus \{s, t\}\)

  - hence \(\forall e \in \text{outgoing } v. \text{ pp-init-f } e = 0\)
    
      by (auto simp: outgoing-def pp-init-f-def)

  - hence [simp]: \(\text{sum pp-init-f (outgoing } v) = 0\) by auto

  - have \(0 \leq \text{pp-init-f } e \text{ for } e\)
    
      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 \(0 \leq \text{sum pp-init-f (incoming } v)\) by auto

  - thus \(\text{sum pp-init-f (outgoing } v) \leq \text{sum pp-init-f (incoming } v)\)

    by auto

**next**

  - fix \(u, v\)

  - assume \((u, v) \in \text{Graph.E (residualGraph c pp-init-f)}\)

  - thus \(\text{pp-init-l } u \leq \text{pp-init-l } v + 1\)

    unfolding \(\text{pp-init-l-def Graph.E-def pp-init-f-def residualGraph-def}\)

    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 \(?thesis by unfold-locale\)

**qed**

**end** — Network

**context** Labeling

**begin**

Push operations preserve a valid labeling [Cormen 26.16].
**Theorem push-pres-Labeling:**

assumes push-precond \( f \mid l \| e \)

shows Labeling \( c \mid s \mid t \) (push-effect \( f \mid e \)) \( l \)

unfolding push-effect-def

proof (cases e; clarsimp)

fix \( u \| v \)

assume [simp]: \( e = (u, v) \)

let \( ?f' = \) (augment-edge \( f \mid u \| v \) (\( \min \) (excess \( f \mid u \)) (cf \( u \| v \))))

from assms have

\( ACTIVE \): excess \( f \mid u \) > 0

and \( EDGE \): \((u, v) \in cf.E \)

and \( ADM \): \( l_u = l_v + 1 \)

unfolding push-precond-def by auto

interpret cf': Preflow \( c \mid s \mid t \) \( ?f' \)

apply (rule augment-edge-preflow-preserve)

using \( ACTIVE \) resE-nonNegative

by auto

show Labeling \( c \mid s \mid t \) \( ?f' \) \( l \)

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 \( \{ l_v \mid v, (u, v) \in cf.E \} \)

proof –

have \( \{ l_v \mid v, (u, v) \in cf.E \} = l'snd'cf.outgoing u \)

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 \( f \mid l \| u \)

shows relabel-increase-u: relabel-effect \( f \mid l \| u \| u > l_u \) (is \( ?G1 \))

and relabel-pres-Labeling: Labeling \( c \mid s \mid t \) (relabel-effect \( f \mid l \| u \)) (is \( ?G2 \))

proof –

from \( PRE \) have

\( NOT-SINK \): \( u \neq t \)

and \( ACTIVE \): excess \( f \mid u \) > 0

and \( NO-ADM \): \( \forall v. (u, v) \in cf.E \implies l_u \neq l_v + 1 \)

unfolding relabel-precond-def by auto

from \( ACTIVE \) have [simp]: \( s \neq u \) using excess-s-non-pos by auto
from active-has-cf-outgoing[OF ACTIVE] have [simp]: \( \exists v. (u, v) \in cf.E \)
  by (auto simp: cf.outgoing-def)

from NO-ADM valid have \( l u < l v + 1 \) if \((u, v)\in cf.E\) for \( v \)
  by (simp add: nat-less-le that)

hence LU-INCR: \( l u \leq \operatorname{Min} \{ l v \mid v. (u, v) \in cf.E \} \)
  by (auto simp: less-Suc-eq-le)

with valid have \( \forall u'. (u', u) \in cf.E \rightarrow l u' \leq \operatorname{Min} \{ l v \mid v. (u, v) \in cf.E \} + 1 \)
  by (smt ab-semigroup-add-class.add.commute add-le-cancel-left le-trans)

moreover have \( \forall v. (u, v) \in cf.E \rightarrow \min \{ l v \mid v. (u, v) \in cf.E \} + 1 \leq l v + 1 \)
  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)
  subgoal using u' v' using valid by auto
  subgoal using auto
  subgoal using NOT-SINK by auto
  done

lemma relabel-preserve-other: \( u \neq v \implies \text{relabel-effect } f l u v = l v \)
  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: \( \forall (u, v) \in cf.E. \neg \text{push-precond } f l (u, v) \)
  assumes no-relabel: \( \forall u. \neg \text{relabel-precond } f l u \)
  shows isMaxFlow f

proof –
  from assms have \( \forall u \in V - \{t\}. \text{excess } f u \leq 0 \)
    unfolding push-precond-def relabel-precond-def
  by force
  with excess-non-negative have \( \forall u \in V - \{s, t\}. \text{excess } f u = 0 \) by force

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 \textit{PRE}: \textit{push-precond} $f \ l$ ($u, v$)

begin

abbreviation $f' \equiv \textit{push-effect} f$ ($u, v$)

sublocale $l'$: Labeling $c \ s \ t \ f' \ l$

using \textit{push-pres-Labeling}[OF \ \textit{PRE}].

lemma $uv$-$cf$-$edge$[simp, intro!]: ($u, v$)$\in \textit{cf}.E$

using \textit{PRE} unfolding \textit{push-precond-def} by auto

lemma $\textit{excess}$-$u$-$pos$; $\textit{excess}$ $f$ $u > 0$

using \textit{PRE} unfolding \textit{push-precond-def} by auto

lemma $l$-$u$-$eq$[simp]: $l \ u = l \ v + 1$

using \textit{PRE} unfolding \textit{push-precond-def} by auto

lemma $uv$-$edge$-$cases$:

obtains \[\text{(par)} \ (u, v)\in E \ (v, u)\notin E\] \[\text{(rev)} \ (v, u)\in E \ (u, v)\notin E\]

using $uv$-$cf$-$edge$ \textit{cfE}$-$ss$-$invE$ no-parallel-edge by blast

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

using \textit{E}$-$ss$-$VxV$ \textit{cfE}$-$ss$-$invE$ no-parallel-edge by auto

lemma $uv$-$not$-$eq$[simp]: $u \neq v$ $v \neq u$

using \textit{E}$-$ss$-$VxV$ \textit{cfE}$-$ss$-$invE$[THEN set-mp, OF \ \textit{uv}$-$cf$-$edge$] no-parallel-edge by auto

definition $\Delta = \min (\textit{excess} \ f \ u) \ (\textit{cf-of} \ f \ (u, v))$

lemma $\Delta$-$positive$; $\Delta > 0$

unfolding $\Delta$-$def$

using $\textit{excess}$-$u$-$pos$ $uv$-$cf$-$edge$[unfolded \textit{cf.E}$-$def$] \textit{resE}$-$positive$

by auto

lemma $f'$-$alt$: $f' = \textit{augment-edge} f$ ($u, v$) $\Delta$

unfolding \textit{push-effect-def} $\Delta$-$def$ by auto

lemma $l'$-$cf$-$alt$: $l'.cf = \textit{ augment-edge-cf} \ cf$ ($u, v$) $\Delta$

unfolding \textit{push-effect-def} $\Delta$-$def$ \textit{augment-edge-cf-def}$

by (auto simp: \textit{augment-edge-cf'}$'$)

lemma $\textit{excess}$-$'$-$u$[simp]: $\textit{excess}$ $f'$ $u = \textit{excess} \ f \ u - \Delta$

unfolding $\textit{excess}$-$def$[where $f=f'$]

proof

show $\text{sum} \ f' \ (\text{incoming} \ u) - \text{sum} \ f' \ (\text{outgoing} \ u) = \textit{excess} \ f \ u - \Delta$

proof (cases rule: $uv$-$edge$-$cases$)

case [simp]: par

hence $UV$-$ONI:(u,v)\in\text{outgoing} \ u - \text{ incoming} \ u$

by (auto simp: \textit{incoming-def outgoing-def no-self-loop})

have 1: $\text{sum} \ f' \ (\text{incoming} \ u) = \text{sum} \ f \ (\text{incoming} \ u)$
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 \( u \)) + \( \sum_{x \in \text{outgoing } u. \text{ if } x = (u, v) \text{ then } \Delta \text{ else } 0} \)
unfolding f'-alt augment-edge-def sum.distrib[symmetric]
  by (rule sum.cong) auto
also have \( \ldots = \sum f \) (outgoing \( u \)) + \( \Delta \)
  using UV-ONI by (auto simp: sum.delta)
finally show \( \text{thesis using 1 unfolding excess-def by simp} \)
next
case [simp]: rev
have UV-INO:\( (v, u) \in \text{incoming } u - \text{outgoing } u \)
  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)
also have \( \ldots = \sum f \) (outgoing \( u \)) + \( \Delta \)
  using UV-INO by (auto simp: sum.delta)
finally show \( \text{thesis using 1 unfolding excess-def by auto} \)
qed

lemma \( \text{excess'}_v \)[simp]: \( \text{excess'}_v \) \( v \) = \( \text{excess}_v \) + \( \Delta \)
unfolding excess-def[where \( f=f' \)]
proof
  show \( \sum f' \) (incoming \( v \)) - \( \sum f' \) (outgoing \( v \)) = \( \text{excess}_v \) + \( \Delta \)
  proof (cases rule: uv-edge-cases)
  case [simp]: par
  have UV-INO: \( (u, v) \in \text{incoming } v - \text{outgoing } v \)
    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 \( v \)) + \( \sum_{x \in \text{incoming } v. \text{ if } x = (u, v) \text{ then } \Delta \text{ else } 0} \)
  unfolding f'-alt augment-edge-def sum.distrib[symmetric]
  apply (rule sum.cong)
  using UV-INO unfolding f'-alt by auto
  also have \( \ldots = \sum f \) (incoming \( v \)) + \( \Delta \)
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: \((v,u) \in \text{outgoing } v - \text{ incoming } v\)
  by (auto simp: incoming-def outgoing-def no-self-loop)

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

have \(\sum f'(\text{outgoing } v)\)
  \(= \sum f(\text{outgoing } v) + (\sum x \in \text{outgoing } v. \ \text{if } x=(v,u) \ \text{then } - \Delta \ \text{else } 0)\)

unfolding f'\text{-alt augment-edge-def sum.distrib[symmetric]}
apply (rule sum.cong)
using UV-INO unfolding f'\text{-alt by auto}
also have \(\ldots = \sum f(\text{outgoing } v) - \Delta\)
using UV-INO by (auto simp: sum.delta)
finally show ?thesis using 1 by (auto simp: excess-def)

qed

lemma excess'\text{-other}[simp]:
  assumes \(x \neq u \ \ x \neq v\)
  shows \(excess f' x = excess f x\)

proof

  have \(NE: (u,v) \notin \text{incoming } x \ \ (u,v) \notin \text{outgoing } x\)
  
  \(\text{assms unfolding incoming-def outgoing-def by auto}\)

  have \(\sum f'(\text{outgoing } x) = \sum f(\text{outgoing } x)\)
  \(\sum f'(\text{incoming } x) = \sum f(\text{incoming } x)\)
  by (auto

    simp: augment-edge-def f'\text{-alt } NE
    split!: if-split
    intro: sum.cong)

  thus ?thesis
  unfolding excess-def by auto

qed

lemma excess'\text{-if}:
  \(excess f' x = \{
    \text{if } x=u \ \text{then } excess f u - \Delta
    \text{ else if } x=v \ \text{then } excess f v + \Delta
    \text{ else excess } f x\}\)
  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 ≠ 0    card E ≥ Suc 0    card E > 0
proof --
  show card E ≠ 0 by (simp add: E-not-empty)
  thus card E ≥ 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 ≤ 2 * card E
proof --
  have card V ≤ 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 ≤ 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 ≤ 2*card V − 1
begin
  lemma height-bound': u∈V ⇒ l u ≤ 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 c) 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
\[\begin{align*}
\text{have } & (\sum u \in U. \text{excess } f u) \\
& = (\sum u \in U. (\sum v \in U'. f (v, u))) - (\sum u \in U. (\sum v \in U'. f (u, v))) \\
\text{proof } & \\
\text{have } & (\sum u \in U. \text{excess } f u) \\
& = (\sum u \in U. (\sum v \in \text{incoming } u. f v)) - (\sum u \in U. (\sum v \in \text{outgoing } u. f v)) \\
\text{is } & = ?R1 - ?R2 \text{ unfolding excess-def by (simp add: sum-subtractf)} \\
\text{also have } & ?R1 = (\sum u \in U. (\sum v \in V. f (v, u))) \\
\text{using sum-incoming-alt-flow fct1 by (meson subsetCE sum.cong)} \\
\text{also have } & \ldots = (\sum u \in U. (\sum v \in U. f (v, u))) + (\sum u \in U. (\sum v \in U'. f (v, u))) \\
\text{proof } & \\
\text{have } & (\sum v \in V. f (v, u)) = (\sum v \in U. f (v, u)) + (\sum v \in U'. f (v, u)) \text{ for } u \\
\text{using } & U'-\text{def fct1 finite-V} \\
& \text{by (metis ab-semigroup-add-class.add.commute sum_subset-diff)} \\
\text{thus } & \text{?thesis by (simp add: sum.distrib)} \\
\text{qed } & \\
\text{also have } & ?R2 = (\sum u \in U. (\sum v \in V. f (u, v))) \\
\text{using sum-outgoing-alt-flow fct1 by (meson subsetCE sum.cong)} \\
\text{also have } & \ldots = (\sum u \in U. (\sum v \in U. f (u, v))) + (\sum u \in U. (\sum v \in U'. f (u, v))) \\
\text{proof } & \\
\{ \\
\text{fix } & A :: \text{nat set} \\
\text{assume } & \text{finite } A \\
\text{then have } & (\sum u \in A. (\sum v \in A. f (u, v))) = (\sum u \in A. (\sum v \in A. f (v, u))) \\
\text{proof } & \text{(induction card } A \text{ arbitrary: } A) \\
\text{case } & 0 \\
\text{then show } & ?case by auto \\
\text{next } & \\
\text{case } & (\text{Suc } x) \\
\text{then obtain } & A' a \\
\text{where } & o1:A = \text{insert } a A' \land o2:x = \text{card } A' \land o3:\text{finite } A' \\
\text{by (metis card-insert-disjoint card-le-Suc-iff le-refl nat.inject)} \\
\text{then have } & \text{ln}(\sum e \in A. g e) = (\sum e \in A'. g e) + g \text{ a} \\
\text{for } & g :: \text{nat } \Rightarrow 'a \\
\text{using } & \text{Suc.hyps}(2) \\
\text{by (metis card-insert-if n-not-Suc-n} \\
& \text{semiring-normalization-rules(24)} \text{ sum.insert}) \\
\text{have } & (\sum u \in A. (\sum v \in A. f (u, v))) \\
& = (\sum u \in A'. (\sum v \in A. f (u, v))) + (\sum v \in A. f (a, v)) \\
\end{align*}\]
(is - = ?R1 + ?R2) using bm by auto
also have ?R1 = (\(\sum u \in A' \cdot (\sum v \in A' \cdot f(u, v))\)) + (\(\sum u \in A'. f(u, a)\))
also note add.assoc
also have ?R1-2 + ?R2 = (\(\sum u \in A' \cdot f(a, u)\)) + (\(\sum v \in A'. f(v, a)\))
also have ?R1-1 = (\(\sum u \in A'. (\sum v \in A'. f(v, u))\))
also note add.assoc[symmetric]
also have ?R1-1' + ?R1-2' = (\(\sum u \in A'. (\sum v \in A. f(v, u))\))
by (metis (no-types, lifting) sum.cong sum.distrib)
finally show ?thesis using fct1 finite-V finite-subset by auto
qed

moreover have (\(\sum u \in U \cdot excess f u\)) > 0
proof -
  have u \in U using U-def by simp
moreover have u \in U \Rightarrow excess f u \geq 0 for u
moreover have (\(\sum v \in U' \cdot f(v, u)\)) \geq 0
ultimately have ?thesis using assms fct1 finite-V
by (metis Diff-cancel Diff-eq-empty-iff
    Diff-infinite-finite finite-Diff sum-pos2)

qcd
ultimately have
fct2: (\(\sum u \in U. (\sum v \in U'. f(v, u))\)) - (\(\sum u \in U. (\sum v \in U'. f(u, v))\)) > 0
by simp

have fct3: (\(\sum u \in U. (\sum v \in U'. f(v, u))\)) > 0
proof -
  have (\(\sum u \in U. (\sum v \in U'. f(v, u))\)) \geq 0
  using capacity-const by (simp add: sum-nonneg)
  moreover have (\(\sum u \in U. (\sum v \in U'. f(v, u))\)) \geq 0
  using capacity-const by (simp add: sum-nonneg)
ultimately show ?thesis using fct2 by simp

qcd

have \(\exists u' v'. (u' \in U \wedge v' \in U' \wedge f(v', u') > 0)\)
proof (rule contr)
  assume \(\neg (\exists u' v'. u' \in U \wedge v' \in U' \wedge f(v', u') > 0)\)
  then have \(\forall u' v'. (u' \in U \wedge v' \in U' \Rightarrow f(v', u') = 0)\)
  using capacity-const by (metis le-neq-trans)
thus False using fct3 by simp
qcd

then obtain u' v' where u' \in U and v' \in U' and f(v', u') > 0
by blast
obtain \( p_1 \) where \( \text{cf} \).\text{isSimplePath} \( u \) \( p_1 \) \( u' \) using \( U\)-def \( \langle u' \in U \rangle \) by auto

moreover have \( (u', v') \in \text{cf} \) \( E \)

proof –

have \( (v', u') \in E \)

using capacity-const \( \langle f \ (v', u') > 0 \rangle \)

by (metis not-less zero-flow-simp)

then have \( \text{cf} \ (u', v') > 0 \) unfolding \( \text{cf-def} \)

using no-parallel-edge \( \langle f \ (v', u') > 0 \rangle \) by (auto split: if-split)

thus \(?\text{thesis}\) unfolding \( \text{cf} \).\( E\)-def by simp

qed

ultimately have \( \text{cf} \).\text{isPath} \( u \) \( p_1 \) \( @ \ [ (u', v') ] \) \( v' \)

using Graph.\text{isPath-append-edge} Graph.\text{isSimplePath-def} by blast

then obtain \( p_2 \) where \( \text{cf} \).\text{isSimplePath} \( u \) \( p_2 \) \( v' \)

using \( \text{cf} \).\text{isSPath-pathLE} by blast

then have \( v' \in U \) using \( U\)-def by auto

thus \( \text{False} \) using \( \langle v' \in U' \rangle \) and \( U'\)-def by simp

qed

then obtain \( p' \) where \( \text{cf} \).\text{isSimplePath} \( u \) \( p' \) \( s \) using \( U\)-def by auto

thus \(?\text{thesis}\) ..

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 \( c \ s \ t \ f \) \( \text{relabel-effect} \ f \ l \ u \)

proof –

let \(?l' = \text{relabel-effect} \ f \ l \ u \)

from relabel-pres-Labeling[of \ assms] interpret \( l': \text{Labeling} \ c \ s \ t \ f \ ?l' \).

from \assms have excess \( f \ u > 0 \) unfolding relabel-precond-def by auto

with \( l'.\text{excess-imp-source-path} \) obtain \( p \) where \( \text{p-obt}: \text{cf} \).\text{isSimplePath} \( u \) \( p \) \( s \).

have \( u \in V \) using excess-nodes-only \( \langle \text{excess} \ f \ u > 0 \rangle \).

then have length \( p < \text{card} \ V \)

using \( \text{cf} \).\text{simplePath-length-less-V}[\langle \text{of} \ u \ p \rangle] \text{p-obt} \ by auto

moreover have \( ?l' \ u \leq ?l' \ s + \text{length} \ p \)

using \( \text{p-obt} \ l'.\text{gen-valid}[\langle \text{of} \ u \ p \ s \rangle] \text{p-obt} \)

unfolding \( \text{cf} \).\text{isSimplePath-def} \ by auto

moreover have \( ?l' \ s = \text{card} \ V \)

using \( l'.\text{Labeling-axioms} \text{Labeling-def} \text{Labeling-axioms-def} \ by auto \)

ultimately have \( ?l' \ u \leq 2*\text{card} \ V - 1 \) by auto

thus Height-Bounded-Labeling \( c \ s \ t \ f \ ?l' \)

apply unfold-locales

using height-bound relabel-preserve-other

by metis

qed

22
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 \( l = \sum_{v \in V} 2 \cdot \text{card } V - l_v \)

**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 \( l': \text{Height-Bounded-Labeling } c \ s \ t \ f \ ?l' \).

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

- hence V-split: \( V = \text{insert } u \ V\) by auto

- show \(?\text{thesis}\)
  - using relabel-increase-u[OF assms] relabel-preserve-other[of u]
  - using \(?l'.height-bound\)
  - unfolding sum-heights-measure-def
  - apply (rewrite at \( \sum_{v \in \Pi} \cdot \text{-V-split}\)+)
  - apply (subst sum.insert-remove[OF finite-V])+
  - using \(?u \in V\)
    - by auto

**qed**

**end** — Height Bounded Labeling

**lemma (in Network)** sum-height-measure-is-OV2:

- sum-heights-measure \( l \leq 2 \cdot (\text{card } V)^2 \)

- unfolding sum-heights-measure-def

**proof** –

- have \( 2 \cdot \text{card } V - l_v \leq 2 \cdot \text{card } V \) for \( v \) by auto

- then have \( \sum_{v \in \text{V}} 2 \cdot \text{card } V - l_v \leq \sum_{v \in \text{V}} 2 \cdot \text{card } V \)
  - by (meson sum-mono)

- also have \( \sum_{v \in \text{V}} 2 \cdot \text{card } V = \text{card } V \cdot (2 \cdot \text{card } V) \)
  - using finite-V by auto

- finally show \( \sum_{v \in \text{V}} 2 \cdot \text{card } V - l_v \leq 2 \cdot (\text{card } V)^2 \)
  - 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 3 e. push-precond f l e and 3 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)
\[ (u, v) \in \text{cfE-of } f \]
\[ l_u = l_v + 1 \]

**Lemma** push-precond-eq-or-nonsat:

\[
push\text{-precond } f \ l \ e \leftrightarrow \text{sat-push-precond } f \ l \ e \lor \text{nonsat-push-precond } f \ l \ e
\]

**Unfolding** push-precond-def sat-push-precond-def nonsat-push-precond-def

by auto

**Lemma** sat-nonsat-push-disj:

\[
\text{sat-push-precond } f \ l \ e \Rightarrow \neg \text{nonsat-push-precond } f \ l \ e
\]

\[
\text{nonsat-push-precond } f \ l \ e \Rightarrow \neg \text{sat-push-precond } f \ l \ e
\]

**Unfolding** sat-push-precond-def nonsat-push-precond-def

by auto

**Lemma** sat-push-alt: sat-push-precond } f \ l \ e

\[
\Rightarrow \text{push-effect } f \ e = \text{augment-edge } f \ e (\text{cf-of } f \ e)
\]

**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)

\[
\Rightarrow \text{push-effect } f \ (u,v) = \text{augment-edge } f \ (u,v) (\text{excess } f \ u)
\]

**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-\(\Delta\): nonsat-push-precond } f \ l \ (u,v) \Rightarrow \Delta = \text{excess } f \ u

**Unfolding** \(\Delta\)-def nonsat-push-precond-def by auto

**Lemma** sat-push-\(\Delta\): sat-push-precond } f \ l \ (u,v) \Rightarrow \Delta = \text{cf } (u,v)

**Unfolding** \(\Delta\)-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' =

\[
\begin{align*}
\text{is-RELABEL': RELABEL'} \\
\text{is-NONSAT-PUSH': NONSAT-PUSH'} \\
\text{is-SAT-PUSH': SAT-PUSH'} \\
\text{edge}
\end{align*}
\]

**Inductive-set** pr-algo-lts' where
nonsat-push': \([\text{Height-Bounded-Labeling \(c\ s\ t\ f\ l\); nonsat-push-precond \(f\ l\ e\)]} \implies ((f,l),\text{NONSAT-PUSH}'(e,\text{push-effect\(f\ e\ l\)))\in pr-algo-lts')
\]
sat-push': \([\text{Height-Bounded-Labeling \(c\ s\ t\ f\ l\); sat-push-precond \(f\ l\ e\)]} \implies ((f,l),\text{SAT-PUSH}'(e,\text{push-effect\(f\ e\ l\)))\in pr-algo-lts')
\]
relabel': \([\text{Height-Bounded-Labeling \(c\ s\ t\ f\ l\); relabel-precond \(f\ l\ u\)]} \implies ((f,l),\text{RELABEL}'(f,\text{relabel-effect\(f\ l\ u\)))\in pr-algo-lts')
\]

fun project-operation where
  project-operation \text{RELABEL}' = \text{RELABEL}
  project-operation \text{NONSAT-PUSH}' = \text{PUSH}
  project-operation (\text{SAT-PUSH}' \cdot) = \text{PUSH}

lemma is-RELABEL-project-conv[simp]:
  is-RELABEL \circ project-operation = is-RELABEL'
apply (clarsimp intro: ext) subgoal for \(x\) by (cases \(x\)) auto done

lemma is-PUSH-project-conv[simp]:
  is-PUSH \circ project-operation = (\lambda x. is-SAT-PUSH' \ x \lor is-NONSAT-PUSH' \ x)
apply (clarsimp intro: ext) subgoal for \(x\) by (cases \(x\)) auto done

end — Network

class Height-Bounded-Labeling
begin
lemma (in Height-Bounded-Labeling) xfer-run:
  \((f,l),p,(f',l')\) \in trcl \text{pr-algo-lts}'
  \land p = map project-operation p'
proof —
  have \(\exists p'\).
  \text{Height-Bounded-Labeling \(c\ s\ t\ f'\ l'\)}
  \land ((f,l),p,(f',l')) \in trcl \text{pr-algo-lts}'
  \land 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: \((f, l), p, fh, lh\) \in trcl \text{pr-algo-lts}
    \land \text{LAST}: \((fh,lh),a,(f',l')\)\in pr-algo-lts'
    by (auto dest!: trcl-rev-uncons)
  from snoc.brev[OFF PP] obtain \(p'\)
    where HBL: \text{Height-Bounded-Labeling \(c\ s\ t\ fh\ lh\)}
    \land PP': \((f, l), p', fh, lh\) \in trcl \text{pr-algo-lts}'
    \land \[\text{simp}\]: \(p = map project-operation p'\)
    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-NON-SAT-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-NON-SAT-PUSH′ x]
    = length (filter is-SAT-PUSH′ p′) + length (filter is-NON-SAT-PUSH′ p′)
  by (induction p′) auto
have \(\text{length} (\text{filter is-PUSH} \ p)\)
\[ = \text{length} (\text{filter is-SAT-PUSH} \ p') + \text{length} (\text{filter is-NONSAT-PUSH} \ p')\]
by simp
also note \(\text{BOUND-SAT}[\text{rule-format}, \text{OF RUN}]\)
also note \(\text{BOUND-NONSAT}[\text{rule-format}, \text{OF RUN}]\)
finally show \(\text{thesis by simp}\)
qed

end — Height Bounded Labeling

2.5.6 Bounding the Relabel Operations

lemma (in Network) relabel-action-bound';
assumes \(A : \langle fxl,p,fxl' \rangle \in \text{trcl pr-algo-lts}'\)
shows \(\text{length} (\text{filter is-RELABEL} \ p) \leq 2 * \text{card} \ V^2\)
proof —
from \(A\) have \(\text{length} (\text{filter is-RELABEL} \ p) \leq \text{sum-heights-measure} (\text{snd} fxl)\)
apply (induction rule: trcl.induct)
apply (auto elim!: pr-algo-lts'.cases)
apply (drule (1) \text{Height-Bounded-Labeling}\text{.relabel-measure})
apply auto
done
also note \(\text{sum-height-measure-is-OV2}\)
finally show \(\text{length} (\text{filter is-RELABEL} \ p) \leq 2 * \text{card} \ V^2\).
qed

lemma (in Height-Bounded-Labeling) relabel-action-bound;
assumes \(A : \langle (f,l),p,(f',l') \rangle \in \text{trcl pr-algo-lts}\)
shows \(\text{length} (\text{filter is-RELABEL} \ p) \leq 2 * \text{card} \ V^2\)
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 \(\langle (f,l),p,(f',l') \rangle \in \text{trcl pr-algo-lts}'\)
shows \(l u \leq l' u\)
using \texttt{assms}

\textbf{proof} (induction \(p\) arbitrary: \(f\ l\))

\textbf{case} \(\text{Nil}\) thus \(?\text{case}\) by \texttt{simp}

\textbf{next}

\textbf{case} (\(\text{Cons}\ a\ p\))

then obtain \(fh\ lh\)

where FIRST: \(((f,l),\text{a,}(fh,lh))\in\text{pr-algo-lts}'\)

and PP: \(((fh,lh),\text{p,}(f',l'))\): \text{trcl pr-algo-lts}'

by (auto simp: \texttt{trcl-conv})

\begin{verbatim}
from FIRST interpret \texttt{Height-Bounded-Labeling c s t f l}
by cases auto

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

\texttt{qed}

\textbf{lemma} \texttt{edge-reappears-at-increased-labeling:}

assumes \(((f,l),\text{p,}(f',l'))\in\text{trcl pr-algo-lts}'\)

assumes \(l\ u\ \geq\ l\ v\ +\ 1\)

assumes \((u,v)\notin\text{cfE-of f}\)

assumes \(E':\ (u,v)\in\text{cfE-of f'}\)

shows \(l\ v < l'\ v\)

using \texttt{assms(1–3)}

\textbf{proof} (induction \(p\) arbitrary: \(f\ l\))

\textbf{case} \(\text{Nil}\) thus \(?\text{case}\ using\ E'\ by auto

\textbf{next}

\textbf{case} (\(\text{Cons}\ a\ p\))

then obtain \(fh\ lh\)

where FIRST: \(((f,l),\text{a,}(fh,lh))\in\text{pr-algo-lts}'\)

and PP: \(((fh,lh),\text{p,}(f',l'))\): \text{trcl pr-algo-lts}'

by (auto simp: \texttt{trcl-conv})

\begin{verbatim}
from FIRST interpret \texttt{Height-Bounded-Labeling c s t f l}
by cases auto

consider (push) \(u'\ v'\)

where push-precond \(f\ l\ (u',v')\) \(fh=\text{push-effect f}\ (u',v')\) \(lh=l\)

| (relabel) \(u'\)

where relabel-precond \(f\ l\ u'\) \(fh=\text{relabel-effect f}\ l\ u'\)

using FIRST
by (auto elim!: \text{pr-algo-lts'.cases simp: push-precond-eq-sat-or-nonsat})

then show \(?\text{case}\ proof\ cases\)

\textbf{case} push

\texttt{note simp = push(2,3)}
\end{verbatim}

The push operation cannot go on edge \((u,v)\) or \((v,u)\)
from push(1) have \((u',v') \neq (u,v)\) \((u',v') \neq (v,u)\) \((u',v') \in cf.E\)
using \((u \geq l \Rightarrow (u,v) \notin cf.E)\)
by (auto simp: push-precond-def)
hence \(NE': (u,v) \notin cf.E\) using \((u,v) \notin cf.E\)
by (auto simp: push-effect-def)

next
\begin{itemize}
\item case relabel
\item note [simp] = relabel(2)
\item show \(?thesis\)
\item proof (cases \(u' = v\))
\item case False
\item from False relabel(3) relabel-preserve-other have [simp]; \(lh v = l v\)
by auto
from False relabel(3)
\item relabel-preserve-other relabel-increase-u[OF relabel(1)]
\item have \(lh u \geq l u\) by (cases \(u' = u\)) auto
\item with \((l u \geq l v + 1)\) have LHG: \(lh u \geq lh v + 1\) by auto
\end{itemize}

from Cons.IH[OF PP - NE'] \(\forall u \geq l v + 1\) show \(?thesis\) by simp

qed

lemma sat-push-edge-action-bound':
assumes \(((f,l),p,(f',l')) \in trcl pr-algo-lts'\)
shows \(length (filter ((=) (SAT-PUSH' e)) p) \leq 2*card V\)

proof
\begin{itemize}
\item obtain \(u v\) where [simp]; \(e=(u,v)\) by (cases \(e\))
\item have \(length (filter ((=) (SAT-PUSH' (u,v))) p) \leq 2*card V - l v\)
if \(((f,l),p,(f',l')) \in trcl pr-algo-lts'\) for \(p\)
using that
\end{itemize}

proof (induction \(p\) arbitrary: \(fl\) rule: length-induct)
\item case (1 \(p\)) thus \(?case\)
\item proof (cases \(p\))
\item case Nil thus \(?thesis\) by auto
\end{itemize}

next
\begin{itemize}
\item case [simp]: \((Cons a p')\)
\item from 1.prems obtain \(fh lh\)
\end{itemize}

qed
where \( \text{FIRST}: ((f,l),a,(fh,lh)) \in \text{pr-algo-lts}' \)
and \( \text{PP}: ((fh,lh),p',(f',l')) \in \text{trcl pr-algo-lts}' \)
by \( \text{(auto dest!: trcl-unscons)} \)

from \( \text{FIRST} \) interpret \( \text{Height-Bounded-Labeling c s t f l} \)
by cases auto

show \( \text{?thesis} \)
proof (cases \( a \) = \( \text{SAT-PUSH}' (u,v) \))
case \( \text{simp} \): False
from \( \text{1.H PP} \) have
\[
\text{length (filter ((=) (SAT-PUSH' (u, v))) p')} \leq 2 \ast \text{card } V - lh v
\]
by auto
with \( \text{FIRST} \) show \( \text{?thesis} \)
apply (cases; clarsimp)
proof
fix \( ua :: \text{nat} \)
assume \( a1 \): length (filter ((=) (SAT-PUSH' (u, v))) p') \leq 2 \ast \text{card } V - \text{relabel-effect } f l ua v
assume \( a2 \): relabel-precond f l ua
have \( 2 \ast \text{card } V - \text{relabel-effect } f l ua v \leq 2 \ast \text{card } V - l v \)
using \( a1 \) order-trans by blast
then show \( \text{length (filter ((=) (SAT-PUSH' (u, v))) p')} \leq 2 \ast \text{card } V - l v \)
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 \( \text{simp} \): True

from \( \text{FIRST} \) have
\( \text{simp} \): \( fh = \text{push-effect } f (u,v) \) \( lh = l \)
and \( \text{PRE} \): \( \text{sat-push-precond } f l (u,v) \)
by (auto elim : \( \text{pr-algo-lts}' \).cases)

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

have \( \text{UVNEH}: (u,v) \notin \text{cfE-of } fh \)
using \( (u \neq v) \)
apply (simp
add: sat-push-alt[OF \( \text{PRE} \)] augment-edge-cf'[OF \( (u,v) \in \text{cf.E}] \))
unfolding \( \text{Graph.E-def} \) by simp
show \(?thesis\)

proof (cases \(\text{SAT-PUSH}'(u,v) \in \text{set } p'\))
  case False
  hence \([\text{simp}]\): \(\text{filter } ((=) \text{SAT-PUSH}'(u,v)) p' = []\)
    by (induction \(p'\)) auto
  show \(?thesis\)
    using \(\text{bspec}[\text{OF height-bound (} u \in V]\)]\)
    using \(\text{bspec}[\text{OF height-bound (} v \in V]\)]\)
    using \(\text{card-V-ge2}\)
    by simp
  next
  case True
  then obtain \(p1 p2\)
    where \([\text{simp}]\): \(p = p1 @ \text{SAT-PUSH}'(u,v) # p2\)
      and \(\text{NP1: SAT-PUSH}'(u,v) \notin \text{set } p1\)
    using \(\text{in-set-conv-decomp-first[of - } p]\)
    by auto
  from \(\text{NP1}\) have \([\text{simp}]\): \(\text{filter } ((=) \text{SAT-PUSH}'(u,v)) p1 = []\)
    by (induction \(p1\)) auto
  from \(\text{PP}\) obtain \(f2 l2 f3 l3\)
    where \(\text{P1: ((f1, l1), p1, (f2, l2))} \in \text{trcl pr-algo-lts'}\)
      and \(\text{S: ((f2, l2), \text{SAT-PUSH}'(u,v), (f3, l3))} \in \text{pr-algo-lts'}\)
      and \(\text{P2: ((f3, l3), p2, (f', l'))} \in \text{trcl pr-algo-lts'}\)
    by (auto simp: \(\text{trcl-conv}\))
  from \(\text{S}\) have \(\text{(} u,v \text{) } \in )E-of f2\) and \([\text{simp}]\): \(l3 = l2\)
    by (auto elim!: \(\text{pr-algo-lts'}.cases\) simp: \(\text{sat-push-precond-def}\))
  with \(\text{edge-reappears-at-increased-labeling[of } \text{P1} - \text{UVNEH}\)
    \(\text{\langle l u = l v + 1\rangle}\)
  have \(\text{AUX1: } l v < l2 v\)
    by auto
  from \(\text{S}\) interpret \(l2: \text{Height-Bounded-Labeling c s t f2 l2}\)
    by (auto elim!: \(\text{pr-algo-lts'}.cases\))
  from \(\text{spec[of } 1.IH, of SAT-PUSH'(u,v)#p2] S P2 have}\)
    \(\text{Suc (length (filter ((=) SAT-PUSH'(u,v)) p2))} \leq 2 * \text{card } V - l2 v\)
    by (auto simp: \(\text{trcl-conv}\))
  also have \(\ldots \leq 2 * \text{card } V - l v\)
    using \(\text{AUX1}\)
    using \(\text{bspec[of } l2.\text{height-bound (} u \in V]\)}\)
    using \(\text{bspec[of } l2.\text{height-bound (} v \in V]\)}\)
    by auto
  finally show \(?thesis\)
    by simp
qed
qed
qed

32
thus \( \text{thesis using assms by fastforce} \)

qed

lemma sat-push-action-bound\(^{1}\):
  assumes \( A : ((f,l),p,(f',l')) \in \text{trcl pr-algo-lts}' \)
  shows \( \text{length (filter is-SAT-PUSH'} p) \leq 4 * \text{card } V * \text{card } E \)
proof
  from \( A \) have \( \text{IN-E: } e \in E \cup E^{-1} \text{ if SAT-PUSH'} e \in \text{set } p \text{ for } e \)
  using that cfE-of-ss-ine
  apply (induction \( p \) arbitrary: \( f \ l \))
  apply (auto simp: trcl-conv sat-push-precond-def elim!: pr-algo-lts'.cases)
  ; blast)+
done

have \( \text{is-SAT-PUSH'} a = (\exists e \in E \cup E^{-1}. a = \text{SAT-PUSH'} e) \text{ if } a \in \text{set } p \text{ for } a \)
  using \( \text{IN-E} \) that \( \text{by (cases } a \text{) auto} \)
hence \( \text{length (filter is-SAT-PUSH'} p) = \text{length (filter (\lambda a. \exists e \in E \cup E^{-1}. a = \text{SAT-PUSH'} e) p)} \)
  by (auto cong: filter-cong)
also have \( \ldots = (\sum e \in E \cup E^{-1}. \text{length (filter (\lambda a. \exists e \in E \cup E^{-1}. a = \text{SAT-PUSH'} e) p))} \)
  by (auto simp: AUX)
also have \( \ldots \leq (\sum i \in E \cup E^{-1}. 2 * \text{card } V) \)
  using sum-mono[OF sat-push-edge-action-bound[OF A], where \( K=E \cup E^{-1} \)] .
also have \( \ldots \leq 4 * \text{card } V * \text{card } E \) using card-Un-le[of E E^{-1}] by simp
finally show \( \text{length (filter is-SAT-PUSH'} p) \leq 4 * \text{card } V * \text{card } E \).
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)**

```
non-sat-potential f l \equiv \sum_{v \in V. \; \text{excess } f v > 0} l v
```

**context** Height-Bounded-Labeling

begin

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

**lemma** non-sat-potential-bound:

**shows** non-sat-potential $f l \leq 2 \ast (\text{card } V) \ast 2$

**proof** –

have non-sat-potential $f l = (\sum v \in \{v \in V. \; 0 < \text{excess } f v\}. \; l v)$

unfolding non-sat-potential-def by auto
also have \ldots \leq (\sum v \in V. \; l v)

proof –

have $f1: \{v \in V. \; 0 < \text{excess } f v\} \subseteq V$ by auto
thus ?thesis using sum subset-diff[OF $f1$ finite-V, of $l$] by auto

qed
also have \ldots \leq (\sum v \in V. \; 2 \ast \text{card } V - 1)

using height-bound by (meson sum mono)
also have \ldots = \text{card } V \ast (2 \ast \text{card } V - 1) by auto
also have card $V$ \ast (2 \ast \text{card } V - 1) \leq 2 \ast \text{card } V \ast \text{card } V by auto

finally show ?thesis by (simp add: power2 eq square)

qed

A non-saturating push decreases the potential.

**lemma** non-sat-push-decr-non-sat-potential:

**assumes** non-sat-push-precond $f l e$

**shows** non-sat-potential ($push$-effect $f e$) $l <$ non-sat-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

34
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 $(push\text{-}effect\ f\ e)\ l \\leq nonsat-potential\ f\ l + 2 \cdot card\ V$

**Proof**
- obtain $u\ v$ where [simp]: $e = (u, v)$ by (cases $e)$ auto

interpret $push\text{-}effect$-locale $c\ s\ f\ l\ u\ v$
using **PRE**
by unfold-locales (simp add: push-precond-\text{-}eq\text{-}sat\text{-}or\text{-}nonsat)

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

Only target node may get activated

have $\{v\in V. excess\ f'\ v > 0\} \subseteq insert\ v\ \{v\in V. excess\ f\ v > 0\}$
using $\Delta$-positive
by (auto simp: excess'-if)

Thus, potential increases by at most $l\ v$
with sum-mono2 \( OF \) - this, of \( l \)

have \( \text{nonsat-potential } f' l \leq \text{nonsat-potential } f l + l v \)

unfolding \( \text{nonsat-potential-def} \)
by (auto simp: sum.insert-if split: if-splits)

Which is bounded by \( O(V) \)
also note height-bound'[of \( v \)]
finally show \( \text{thesis by simp} \)
qed

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

lemma relabel-nonsat-potential:
assumes \( \text{PRE: relabel-precond } f l u \)
shows \( \text{nonsat-potential } f (\text{relabel-effect } f l u) \leq \text{nonsat-potential } f l + 2 \ast \text{card } V \)

proof -
have [simp, intro!]: \( \text{finite } \{v \in V. \text{excess } f v > 0\} \)
by auto

let \( ?l' = \text{relabel-effect } f l u \)

interpret \( l'\): Height-Bounded-Labeling \( c s t f \ ?l' \)

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

have \( \text{nonsat-potential } f \ ?l' \)
= \( \text{sum } ?l' \ (\{v \in V. 0 < \text{excess } f v\} - \{u\}) + ?l' u \)
unfolding \( \text{nonsat-potential-def} \)
using \( \text{U-ACTIVE} \) by (auto intro: sum-arb)
also have \( \text{sum } ?l' \ (\{v \in V. 0 < \text{excess } f v\} - \{u\}) \)
= \( \text{sum } l \ (\{v \in V. 0 < \text{excess } f v\} - \{u\}) \)
using relabel-preserve-other by auto
also have \( ?l' u \leq l u + 2 \ast \text{card } V \)
using \( l'.\text{height-bound}[\( OF : u \in V\)] \) by auto
finally have \( \text{nonsat-potential } f \ ?l' \)
\leq \( \text{sum } l \ (\{v \in V. 0 < \text{excess } f v\} - \{u\}) + l u + 2 \ast \text{card } V \)
by auto
also have \( \text{sum } l \ (\{v \in V. 0 < \text{excess } f v\} - \{u\}) + l u \)
= \( \text{nonsat-potential } f l \)
unfolding \( \text{nonsat-potential-def} \)
using \( \text{U-ACTIVE} \) by (auto intro: sum-arb[symmetric])
finally show \( \text{thesis} \).
qed

end — Height Bounded Labeling
context Network
begin

lemma nonsat-push-action-bound':
assumes A: ((f,l),p,(f',l')) \in trcl pr-algo-lts'
shows length (filter is-NONSAT-PUSH' p) \leq 18 \ast (\card V)^2 \ast \card E
proof -
have B1: length (filter is-NONSAT-PUSH' p)
  \leq \text{nonsat-potential f l}
  + 2 \ast \card V \ast (\text{length (filter is-SAT-PUSH' p)})
  + 2 \ast \card V \ast (\text{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))\in pr-algo-lts'
  and PP: ((fh,lh),p,(f',l')) \in 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)
\[ \leq 2 \times (\text{card } V)^2 + 8 \times (\text{card } V)^2 \times \text{card } E + 4 \times (\text{card } V)^3 \]

by (simp add: power2-eq-square power3-eq-cube)

also have \((\text{card } V)^3 \leq 2 \times (\text{card } V)^2 \times \text{card } E\)

by (simp add: card-V-est-E power2-eq-square power3-eq-cube)

finally have \(\text{length } (\text{filter is-NONSAT-PUSH' } p) \leq 2 \times (\text{card } V)^2 + 16 \times (\text{card } V)^2 \times \text{card } E\)

by linarith

also have \(2 \times (\text{card } V)^2 \leq 2 \times (\text{card } V)^2 \times \text{card } E\) by auto

finally show \(\text{length } (\text{filter is-NONSAT-PUSH' } p) \leq 18 \times (\text{card } V)^2 \times \text{card } E\)

by linarith

qed

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

**Lemma (in Height-Bounded-Labeling) push-action-bound:**

\(\text{assumes } A : ((f,l),p,(f',l')) \in \text{trcl pr-algo-lts} \)

\(\text{shows } \text{length } (\text{filter is-PUSH } p) \leq 2 \times (\text{card } V)^2 \times \text{card } E \)

\text{apply} (\text{rule order-trans[OF xfer-push-bounds[OF - - A]]; (intro allI impI)}

\text{apply} (\text{erule sat-push-action-bound'; fail)}

\text{apply} (\text{erule nonsat-push-action-bound'; fail)}

\text{apply} (\text{auto simp: power2-eq-square})

\text{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 \Rightarrow nat where**

\(\text{cost-estimate RELABEL } = \text{card } V\)

\(\text{cost-estimate PUSH } = 1\)

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:**

\(\text{assumes } A : ((f,l),p,(f',l')) \in \text{trcl pr-algo-lts} \)

\(\text{shows } (\sum a \leftarrow p. \text{cost-estimate } a) \leq 26 \times (\text{card } V)^2 \times \text{card } E \)

\(\text{proof} - \)

\(\text{have } (\sum a \leftarrow p. \text{cost-estimate } a) = \text{card } V \times \text{length } (\text{filter is-RELABEL } p) + \text{length } (\text{filter is-PUSH } p) \)

\(\text{proof} (\text{induction } p) \)

\(\text{case Nil} \)

\(\text{then show } ?\text{case by simp}\)

\(\text{next} \)

\(\text{case } (\text{Cons } a \text{ } p) \)

\(\text{then show } ?\text{case by } \text{(cases } a\text{ ) auto}\)

\text{qed}

also have \(\text{card } V \times \text{length } (\text{filter is-RELABEL } p) \leq 2 \times (\text{card } V)^3\)
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 $O(V^2E)$ [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: \([\text{Height-Bounded-Labeling } c s t f l; \text{push-precond } f l e]\)
\(\implies (\text{push-effect } f e l),(f,l)) \in \text{pr-algo-rel}\)

| relabel: \([\text{Height-Bounded-Labeling } c s t f l; \text{relabel-precond } f l u]\)
\(\implies ((f,\text{relabel-effect } f l u),(f,l)) \in \text{pr-algo-rel}\)

**lemma** pr-algo-rel-alt: pr-algo-rel =
\[\{ ((\text{push-effect } f e l),(f,l)) | \text{f e l. \(\text{Height-Bounded-Labeling } c s t f l\) \& \text{push-precond } f l e} \}\]
\[\cup \{ ((f,\text{relabel-effect } f l u),(f,l)) | \text{f u l. \(\text{Height-Bounded-Labeling } c s t f l\) \& \text{relabel-precond } f l u} \}\]

by (auto elim!: pr-algo-rel.cases intro: pr-algo-rel.intros)

**definition** pr-algo-len-bound \(\equiv 2 \ast (\text{card V})^2 + 22 \ast (\text{card V})^2 \ast \text{card E}\)

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

**proof**
- have length \(p = \text{length \(\text{filter is-PUSH } p\)} + \text{length \(\text{filter is-RELABEL } p\)}\)
  proof (induction \(p\))
    - case Nil then show \(\theta\) case by simp
  next
    - case (Cons a p) then show \(\theta\) case by (cases a) auto
  qed

also note push-action-bound[OF A]
also note relabel-action-bound[OF A]
finally show \(\theta\) thesis unfolding pr-algo-len-bound-def by simp
qed

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

**proof**
- have FIN-OPS: finite \((\text{UNIV::pr-operation set})\)
  apply (rule finite-subset[where \(B={\text{PUSH,RELABEL}}\)])
  using pr-operation.exhaust by auto

- have \(\{p. \(\exists f' l'. (f,l),p,(f',l') \in \text{trcl pr-algo-lts}\}\) \subseteq \(\{p. \text{length } p \leq \text{pr-algo-len-bound}\}\)
  by (auto simp: pr-algo-lts-length-bound)

also note finite-lists-length-le[OF FIN-OPS, simplified]
finally (finite-subset) show \(\theta\) thesis.

qed

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

assumes \( (f', fl) \in pr\text{-}algo\text{-}rel \)

shows \( pr\text{-}algo\text{-}measure f' < pr\text{-}algo\text{-}measure fl \)

using \assms

proof (cases \( f, f' \); cases \( fl, l' \); simp)

fix \( f, f', l' \)

assume \( A : ((f', l'), (f, l)) \in pr\text{-}algo\text{-}rel \)

then obtain \( a \) where \( LTS\text{-}STEP: ((f, l), a, (f', l')) \in pr\text{-}algo\text{-}lts \)

by cases (auto intro: pr\text{-}algo\text{-}lts\text{-}intros)

from \( A \) interpret Height\text{-}Bounded\text{-}Labeling \( c \) \( s \) \( t \) \( f \) \( l \) by cases auto

from \( pr\text{-}algo\text{-}maintains\text{-}hb\text{-}labeling[\OF \ LTS\text{-}STEP] \)

interpret \( f' : \) Height\text{-}Bounded\text{-}Labeling \( c \) \( s \) \( t \) \( f' \) \( l' \).

let \( ?S1 = \{ \text{length } p \mid p. \exists fx lx. ((f, l), p, fx, lx) \in \text{trcl pr\text{-}algo\text{-}lts} \} \)

let \( ?S2 = \{ \text{length } p \mid p. \exists fx lx. ((f', l'), p, fx, lx) \in \text{trcl pr\text{-}algo\text{-}lts} \} \)

have finite \( ?S1 \) using finite\text{-}image\text{-}set path\text{-}set\text{-}finite by blast

moreover have \( ?S1 \neq \{ \} \) by (auto intro: \text{exI[where } x=[])\)

ultimately obtain \( p \) \( fx \) \( lx \) where

\( \text{length } p = \text{Max } ?S1 \)

\( ((f, l), p, fx, lx) \in \text{trcl pr\text{-}algo\text{-}lts} \)

apply \text{-}

apply (drule (1) \text{Max\text{-}in})

by auto

have finite \( ?S2 \) using finite\text{-}image\text{-}set \( f' \)\text{-}path\text{-}set\text{-}finite by blast

have \( ?S2 \neq \{ \} \) by (auto intro: \text{exI[where } x=[])\)

{ assume \( MG: \text{Max } ?S2 \geq \text{Max } ?S1 \)

from \( \text{Max\text{-}in[\OF \ (\text{finite } ?S2) \ \{ ?S2\neq\{\}] \} } \) \( \text{obtain } p \) \( fx \) \( lx \) where

\( \text{length } p = \text{Max } ?S2 \)

\( ((f', l'), p, fx, lx) \in \text{trcl pr\text{-}algo\text{-}lts} \)

by auto

with \( MG \) \( LTS\text{-}STEP \) have

\( \text{LEN: } \text{length } (a\#p) > \text{Max } ?S1 \)

and \( P : ((f, l), a\#p.(fx, lx)) \in \text{trcl pr\text{-}algo\text{-}lts} \)

by (auto simp: trcl-conv)

from \( P \) have \( \text{length } (a\#p) \in ?S1 \) by blast

from \( \text{Max\text{-}ge[\OF \ (\text{finite } ?S1) \ this] } \) \( \text{LEN have False by simp} \)

\} thus \( \text{pr\text{-}algo\text{-}measure } (f', l') < \text{pr\text{-}algo\text{-}measure } (f, l) \)

unfolding \( \text{pr\text{-}algo\text{-}measure\text{-}def} \) by (rule ccontr) auto

qed

lemma \( \text{wf\text{-}pr\text{-}algo\text{-}rel[\simp, intro!]}: \text{wf pr\text{-}algo\text{-}rel} \)

apply (rule \text{wf\text{-}subset})

apply (rule \text{wf\text{-}measure[where } f=pr\text{-}algo\text{-}measure])
by (auto simp: pr-algo-measure)

end — Network

2.8 Gap Heuristics

class Network

begin

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

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

The gap heuristics preserves a valid labeling.

lemma (in Labeling) gap-pres-Labeling:
\begin{itemize}
\item assumes PRE: gap-precond $l \ k$
\item defines $l' \equiv$ gap-effect $l \ k$
\item shows Labeling c s t f l'$
\end{itemize}

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:
\begin{itemize}
\item assumes PRE: gap-precond $l \ k$
\item defines $l' \equiv$ gap-effect $l \ k$
\item shows Height-Bounded-Labeling c s t f l'$
\end{itemize}

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

**definition** gap-relabel-effect \( f \ l \ u \) \( \equiv \) let \( l' = \) relabel-effect \( f \ l \ u \) in

if (gap-precond \( l' \) \( l \ u \)) then gap-effect \( l' \) \( l \ u \) else \( l' \)

The combined gap-relabel operation preserves a valid labeling.

**lemma** (in Labeling) gap-relabel-pres-Labeling:

assumes \( PRE: \) relabel-precond \( f \ l \ u \)

defines \( l' \equiv \) gap-relabel-effect \( f \ l \ u \)

shows Labeling \( c \ s \ t \ f \ l \)

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 \( l' \equiv \) 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 \( (gap-effect \ l \ k) \) \( \leq \) 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 \( (gap-relabel-effect \ l \ u) \) \( < \) 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 over-approximates the steps of a push-relabel algorithm with gap heuristics.
inductive-set gap-algo-rel where
  push: [[Height-Bounded-Labeling c s t f l; push-precond f l e]]
  ⇒ ((push-effect f e,l),(f,l))∈gap-algo-rel
  | relabel: [[Height-Bounded-Labeling c s t f l; relabel-precond f l u]]
  ⇒ ((f,gap-relabel-effect f l u),(f,l))∈gap-algo-rel

lemma wf-gap-algo-rel[simp, intro!]: wf gap-algo-rel
proof –
  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 
  Maxflow-Lib.Refine-Add-Fofu
  Generic-Push-Relabel
begin
context Network
begin
  definition relabel f l u ≡ { 
    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 ≡ { 
    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 < 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
3 FIFO Push Relabel Algorithm

theory Fifo-Push-Relabel
imports
  Maxflow-Lib, 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 the 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 \neq [])
  let u=hd Q; let Q=tl Q;
  assert (u \in V \land u \neq s \land u \neq t);

  (f,l,Q) \leftarrow \text{FOREACH} \{ v . (u,v) \in cfE-of f_0 \} (\lambda (f,l,Q). \text{excess } f u \neq 0) (\lambda v (f,l,Q)). do {
    if (l u = l v + 1) then do {
      (f',l,Q) \leftarrow fifo-push f l Q (u,v);
      assert (\forall v'. v' \neq v \implies 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 \neq 0 then do {
  (l,Q) \leftarrow 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 \land set Q = \{ v \in V - \{s,t\} . \text{excess } f v \neq 0 \}

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

definition QD-invar u f Q ≡ u \in V - \{s,t\} \land distinct Q \land set Q = \{ v \in V - \{s,t,u\} . \text{excess } f v \neq 0 \}

lemma Q-invar-when-discharged1: \([QD-invar u f Q; \text{excess } f u = 0] \implies Q-invar f Q \)
  unfolding Q-invar-def QD-invar-def by auto

lemma Q-invar-when-discharged2: \([QD-invar u f Q; \text{excess } f u \neq 0] \implies Q-invar f (Q@[u]) \)
  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 $VC$: $s=v \lor t=v \lor excess f v \neq 0$
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 $\Delta$-positive have $excess f v + \Delta \neq 0$ if $v \notin \{s,t\}$
  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 $VC$: $s \neq v \land t \neq v$ 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 $VC$ $INV$ $\Delta$-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 $QNE$: $Q \neq []$
shows fifo-discharge $f l Q \leq SPEC (\lambda(f',l',Q').$
  Height-Bounded-Labeling $c s f' l'$
  \land $Q$-invar $f' Q'$
  \land ((f',l').(f,l)) $\in$ gap-algo-rel$^+$
)  
proof
  from $QNE$ obtain $u Qr$ where [simp]: $Q = u \# Qr$ by (cases $Q$) auto
  from $QINV$ have $U$: $u \in V - \{s,t\}$ \quad $QD$-invar $u f Qr$ and $XU$-orig: $excess f u \neq 0$
by (auto simp: Q-invar-def QD-invar-def)

have [simp, intro!]: finite \{ v, (u, v) \in 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 - - \( \neq 0 \) then \( \) else - vcg-intro-frame)
  apply (refine-vcg FOREACHc-rule[where I=λ it (f′, l′, Q′).
    Height-Bounded-Labeling c s t f′ l′
    ∧ QD-invar u f′ Q′
    ∧ ((f′, l′), (f, l)) \in gap-algo-rel*
    ∧ it \subseteq \{ v, (u, v) \in cfE-of f′ \}
    ∧ (excess f′ u \neq 0 \rightarrow (\forall v \in \{ v, (u, v) \in cfE-of f′ \} \rightarrow it. l′ u \neq l′ v + 1)
    )
  )

apply (vc-solve simp: DINV QINV it-step-insert-iff split del: if-split)
subgoal for v it f′ l′ 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 \neq 0 and UI: u \in V \ u \neq s \ u \neq t
  and QDI: QD-invar u f′ Q′

  assume v \in it and ITSS: it \subseteq \{ v, (u, v) \in l′.cf.E \}
  hence UVE: (u, v) \in l′.cf.E by auto

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

  assume SAT-EDGES: \( \forall v \in \{ v, (u, v) \in cfE-of f′ \} \rightarrow it. l′ u \neq Suc (l′ v) \)

  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 \([\text{simp}]: l' u = \text{Suc} (l' v)\)
assume \(\text{PRE}: \text{push-precond} f' l' (u, v)\)
then interpret \(\text{pe}: \text{push-effect-locale} c s t f' l' u v\) by unfold-locales

have \(\text{UVNE'}: l'.\text{cf} (u, v) \neq 0\)
using \(l'.\text{resE-positive}\) by fastforce

show \(?\text{thesis}\)
apply (rule vcg-rem-frame)
apply refine-vcg
apply (vc-solve simp: l'.\text{push-pres-height-bound}[OF \text{PRE}])
subgoal by unfold-locales
subgoal by (auto simp: \text{pe}.f'-alt \text{augment-edge-cf-def})
subgoal
using l'.\text{push-activate-pres-QD-invar}[OF \text{QDI}\ \text{PRE}]
using l'.\text{push-no-activate-pres-QD-invar}[OF \text{QDI}\ \text{PRE}]
by auto
subgoal
by (meson gap-algo-rel.\text{push} \text{REL}\ \text{PRE} converse-rtrancl-into-rtrancl \text{HBL})

subgoal for \(x\) proof –
assume \(x \in \text{it}\ \ x \neq v\)
with \(\text{ITSS}\) have \((u, x) \in l'.\text{cf}.E\) by auto
thus \(?\text{thesis}\)
using \(x \neq v\)
unfolding pe.f'-alt
apply (simp add: \text{augment-edge-cf}'
unfolding Graph.E-def
by (auto)
qed

subgoal for \(v'\) proof –
assume \(\text{excess} f' u \neq \text{pe}.\Delta\)
hence \(\text{PED}: \text{pe}.\Delta = l'.\text{cf} (u, v)\)
unfolding pe.\Delta-def by auto
hence \(\text{ESS}: \text{pe}.l'.\text{cf}.E \subseteq (l'.\text{cf}.E \cup \{(v, u)\}) - \{(u, v)\}\)
unfolding pe.f'-alt
apply (simp add: \text{augment-edge-cf}')
unfolding Graph.E-def
by auto

assume \(v' \in \text{it} \rightarrow v' = v\) and \(\text{UV'E}: (u, v) \in \text{pe}.l'.\text{cf}.E\) and \(\text{LUSLV'}: l' v = l' v'\)
with \(\text{ESS}\) have \(v' \in \text{it}\) by auto
moreover from \(\text{UV'E} \\text{ESS} \text{pe}.\text{uv-not-eq}(2)\) have \((u, v') \in l'.\text{cf}.E\) by auto
ultimately have \(l' u \neq \text{Suc} (l' v')\) using \(\text{SAT-EDGES}\) by auto
with \(\text{LUSLV'}\) show \(\text{False}\) 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'' 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 \(E''\) − \{t\}!

  (f,l,−) ← whileT (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
    f = λ(f,l,Q). Height-Bounded-Labeling c s t f l ∧ Q-inv f Q
    and R = inv-image (gap-algo-rel\w) (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-inv f l 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-inv f []
  hence ∀ u∈V − \{s,t\}. excess f u = 0 unfolding Q-inv f l by auto

51
thus isMaxFlow \ f \ \text{by} \ \text{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). \ \exists \ l1 \ l2 \ l3. \ \ l = l1 \@ a \# l2 \@ b \# l3 \}\n
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 = \ \{(i!i, i\!\!j). \ i < j \wedge j < length \ l \}\n  unfolding list-before-rel-def
  apply (rule; clarsimp)
subgoal for \ a \ b \ l \ 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: 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 asym. intros irrefl-def list-before-irrefl-eq-distinct list-before-trans transE)

Structural properties on the list

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

lemma list-before-rel-cons: list-before-rel (x#l) = (set x) ∪ 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 < Suc (length l)
      unfolding list-before-rel-alt by blast
      ultimately show ?thesis by blast
  qed

proof
  { assume a=x and b∈set l
    hence a=x#l b∈set l using IDX-BOUND
      by (auto simp: nth-Cons split: nat.splits)
  } moreover { assume i ≠ 0
    with IDX-BOUND have a=x|v(i-l) b=x|v(j-l) i-l < j-l j-l < 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 ∈ x#l j < length l by (auto simp: in-set-conv-nth)
      hence a=x#l|v 0 b = x#l|v 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 \( \text{is-top-sorted} \) to capture the sortedness criterion, but not the completeness criterion, i.e., the list needs not contain all nodes of the graph.

**definition** \( \text{is-top-sorted} \ R \ l \equiv \text{list-before-rel} \ l \cap (R^*)^{-1} = \{\} \)

**lemma** \( \text{is-top-sorted-alt} : \text{is-top-sorted} \ R \ l \iff (\forall \ x \ y. (x,y) \in \text{list-before-rel} \ l \rightarrow (y,x) \notin R^*) \)

**unfolding** \( \text{is-top-sorted-def} \) **by** auto

**lemma** \( \text{is-top-sorted-empty-rel}[\text{simp}]: \text{is-top-sorted} \ \{\} \ l \iff \text{distinct} \ l \)

**by** (auto simp: \( \text{is-top-sorted-def} \ \text{list-before-irrefl-eq-distinct} [\text{symmetric}] \ \text{irrefl-def} \))

**lemma** \( \text{is-top-sorted-empty-list}[\text{simp}]: \text{is-top-sorted} \ R \ [] \)

**by** (auto simp: \( \text{is-top-sorted-def} \))

A topological sorted list must be distinct

**lemma** \( \text{is-top-sorted-distinct} : \)

assumes \( \text{is-top-sorted} \ R \ l \)

shows \( \text{distinct} \ l \)

**proof** (rule contr)

assume \( \neg \text{distinct} \ l \)

with \( \text{list-before-irrefl-eq-distinct[of} \ l \) obtain \( x \) where \( (x,x) \in (\text{list-before-rel} \ l) \)

by (auto simp: \( \text{irrefl-def} \))

with \( \text{assms} \) show False **unfolding** \( \text{is-top-sorted-def} \) **by** auto

**qed**

**lemma** \( \text{is-top-sorted-cons}: \text{is-top-sorted} \ (x \# l) \iff (\{x\} \times \text{set} \ l \cap (R^*)^{-1} = \{\}) \land \text{is-top-sorted} \ R \ l \)

**unfolding** \( \text{is-top-sorted-def} \)

**by** (auto simp: \( \text{list-before-rel-cons} \))

**lemma** \( \text{is-top-sorted-append}: \text{is-top-sorted} \ R \ (l1 @ l2) \iff (\text{set} \ l1 \times \text{set} \ l2 \cap (R^*)^{-1} = \{\}) \land \text{is-top-sorted} \ R \ l1 \land \text{is-top-sorted} \ R \ l2 \)

**by** (induction \( l1 \) ) (auto simp: \( \text{is-top-sorted-cons} \))

**lemma** \( \text{is-top-sorted-remove-elem}: \text{is-top-sorted} \ R \ (l1 @ \# l2) \implies \text{is-top-sorted} \ R \ (l1 @ l2) \)

**by** (auto simp: \( \text{is-top-sorted-cons} \ \text{is-top-sorted-append} \))

Removing edges from the graph preserves topological sorting

**lemma** \( \text{is-top-sorted-antimono}: \)

assumes \( R \subseteq R' \)

assumes \( \text{is-top-sorted} \ R' \ l \)

54
show 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 \( R' \subseteq R \cup \{x\} \times X \) \( R' \cap \text{UNIV} \times \{x\} = \{\} \)
  assumes \( x \notin \text{set } l \)
  assumes is-top-sorted R l
  shows is-top-sorted R' l
proof –
  \{ 
  fix a b
  assume \((a,b) \in R^* \) \( a \neq x \) \( b \neq x \)
  hence \((a,b) \in R^* \)
  proof (induction rule: converse-rtrancl-induct)
    case base
    then show ?case by simp
  next
    case (step y z)
    with assms(1,2) have \( z \neq x \) \( (y,z) \in R \) 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 \( l'.cf.E = \text{insert} (\text{prod.swap} e) \ cf.E - \{e\} \vee l'.cf.E = \ cf.E - \{e\} \)
by auto
thus adm-edges \(?f'\ l = adm-edges f l - \{e\} \)
proof (cases rule: disjE[consumes 1])
  case 1
  from sat have \( e \in adm-edges f l \)
  unfolding sat-push-precond-def adm-edges-def
  by auto
thus adm-edges \(?f'\ l = adm-edges f l - \{e\} \)
  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
next
  case nonsat
  hence \( adm-edges (\text{push-effect} f e) l = adm-edges f l \)
  proof (cases e; simp add: nonsat-push-alt)
    fix \( u \ v \)
    assume \[ \text{simp}; \, e=(u,v) \]
    let \( ?f'=(\text{augment-edge} f \ (u,v) \ (\text{excess} f u)) \)
    interpret \( l':\text{Labeling} \ c \ s \ t \ ?f'\ l \)
    using push-pres-Labeling[OF assms] nonsat-push-alt nonsat
    by auto
  from nonsat have \( e \in adm-edges f l \)
  unfolding nonsat-push-precond-def adm-edges-def by auto
  with adm-edges-inv-disj have \( \text{AUX}: \text{prod.swap} e \notin \, adm-edges f l \)
  by (auto simp: swap-in-iff-inv)
  from nonsat have \( \text{excess} f u < c f (u,v) \quad 0 < \text{excess} f u \)
  and \[ \text{simp}; \, l u = l v + 1 \]
  unfolding nonsat-push-precond-def by auto
  hence \( l'.cf.E \subseteq \text{insert} (\text{prod.swap} e) \ cf.E \quad l'.cf.E \geq \ cf.E \)
  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-cancel1 f-non-negative linorder-not-le)
done

hence \( l' \cdot cf. E = \text{insert} (\text{prod.swap} \ e)\ cf. E \land l' \cdot cf. E = cf. E \)
by auto

thus \( \text{adm-edges} \ ?f' \ l = \text{adm-edges} \ f \ l \) using \( \text{AUX} \)
unfolding \( \text{adm-edges-def} \)
by auto

qed

thus \( ?l' \) by auto

qed

hence \( l' \).

cf. \( E = \{ \text{insert} (\text{prod.swap} \ e)\ \text{cf.} \ l' \}

\lor \ (\text{cf.} \ l) \)

\( \text{cf.} \ E = \text{cf.} \ E \)
by auto

thus \( \text{adm-edges} \ ?f' \ l = \text{adm-edges} \ f \ l \) using \( \text{AUX} \)
unfolding \( \text{adm-edges-def} \)
by auto

qed

thus \( \text{adm-edges} \ ?f' \ l = \text{adm-edges} \ f \ l \) using \( \text{AUX} \)
unfolding \( \text{adm-edges-def} \)
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 [Cor-
men 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 \( \text{PRE: relabel-precond} \ f \ l \ u \)
defines \( l' \equiv \text{relabel-effect} \ f \ l \ u \)
shows \( \text{adm-edges} \ f \ l' \cap \text{cf. outgoing} \ u \neq \{\} \) (is \( ?G1 \))
and \( \text{adm-edges} \ f \ l' \cap \text{cf. incoming} \ u = \{\} \) (is \( ?G2 \))
and \( \text{adm-edges} \ f \ l' - \text{cf. adjacent} \ u = \text{adm-edges} \ f \ l - \text{cf. adjacent} \ u \) (is \( ?G3 \))

proof –

from \( \text{PRE} \) have \( \text{NOT-SINK:} \ u \neq t \)
and \( \text{ACTIVE:} \ \text{excess} \ f \ u > 0 \)
and \( \text{NO-ADM:} \ \forall v. (u,v) \in \text{cf.} \ E \Rightarrow l u \neq l v + 1 \)
unfolding \( \text{relabel-precond-def} \) by auto

have \( \text{NE:} \ \{ l v \mid v. (u,v) \in \text{cf.} \ E \} \neq \{\} \)
using \( \text{active-has-cf-outgoing[OF ACTIVE]} \ \text{cf. outgoing-def} \) by blast

obtain \( v \)
where \( \text{VUE:} (u,v) \in \text{cf.} \ E \) and \( \text{simp:} \ l v = \text{Min} \ \{ l v \mid v. (u,v) \in \text{cf.} \ E \} \)
using \( \text{Min-in[OF finite-min-cf-outgoing[of u]} \ \text{NE} \) by auto

hence \( (u,v) \in \text{adm-edges} \ f \ l' \cap \text{cf. outgoing} \ u \)
unfolding \( l'\text{-def relabel-effect-def} \text{adm-edges-def} \text{cf. outgoing-def} \)
by (auto simp: cf-no-self-loop)
thus \( ?G1 \) by blast

\{
fix \( uh \)
assume \( (uh,u) \in \text{adm-edges} \ f \ l' \)
hence \( l' \ uh = l' \ u + 1 \) and \( \text{UHUE:} \ (uh,u) \in \text{cf.} \ E \)
unfolding \( \text{adm-edges-def} \) by auto
hence \( uh \neq u \) using \( \text{cf-no-self-loop} \) by auto

\}
hence \([\text{simp}]: l' \ uh = l \ uh\) unfolding \(l'\)-def relabel-effect-def by simp
from \(I\) relabel-increase-\(u\)[OF \(PRE\), folded \(l'\)-def] have \(l \ uh > l \ u + 1\)
by simp
with valid[OF \(UHUE\)] have \(\text{False}\) 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 \(n :: \text{node} \Rightarrow \text{node set}\) from each node to the set of target nodes of not yet visited edges.

locale neighbor-invar = Height-Bounded-Labeling +
fixes \(n :: \text{node} \Rightarrow \text{node set}\)
assumes neighbors-adm: \([v \in \text{adjacent-nodes } u - n \ u] \implies (u,v) \notin \text{adm-edges } f\)
assumes neighbors-adj: \(n \ u \subseteq \text{adjacent-nodes } u\)
assumes neighbors-finite[\(\text{simp, intro!}\): 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 \(c\ s\ t\ (\text{push-effect } f\ e)\ l\ n\)
proof (cases \(e\))
  case [\(\text{simp}\): \((\text{Pair } u\ v)\)
    show ?thesis proof simp
      from \(PRE\) interpret push-effect-locale \(c\ s\ t\ f\ l\ u\ v\)
      by unfold-locals simp
      from push-pres-height-bound[OF \(PRE\)]
      interpret \(l'\): Height-Bounded-Labeling \(c\ s\ t\ f'\ l\).

end

59
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\neq u \ b\in adjacent-nodes a \ b \notin n a \ (a,b)\in adm-edges f ?l'
  hence (a,b)\in cf.E unfolding adm-edges-def by auto
  with A relabel-adm-edges(2,3)[OF PRE] neighbors-adm
  show False
    by (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: \[ u\in V; u\neq s \] \implies excess f u \neq 0 \iff excess f u > 0
using excess-non-negative' by force

lemma no-neighbors-relabel-precond:
assumes n u = {} \ u\neq t \ u\neq s \ u\in V \ excess f u \neq 0
shows relabel-precond f l u
using assms neighbors-adm cfEs-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)\notin adm-edges f l
implies 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

class 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 \( \equiv \) do {
  assert (u \in V - \{s,t\});
  while \( T(\lambda(f,l,n).\text{excess}(f,u) \neq 0) \) do {
    v \leftarrow \text{select v. v} \in n(u);
    case v of
    None \Rightarrow do {
      l \leftarrow \text{relabel}(f,l,u);
      return (f,l,n(u := \text{adjacent-nodes}(u)));
    } |
    Some v \Rightarrow do {
      assert (v \in V \wedge (u,v) \in E \cup E^{-1});
      if ((u,v) \in cfE-of f \wedge l(u) = l(v) + 1) then do {
        f \leftarrow \text{push}(f,l,(u,v));
        return (f,l,n);
      } else do {
        assert ((u,v) \notin \text{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 no f l n +
assumes lu-incr: lo u \leq l u
assumes u-node: u \in V - \{s,t\}
assumes no-relabel-adm-edges: lo u = l u \implies \text{adm-edges}(f,l) \subseteq \text{adm-edges}(fo,lo)
assumes no-relabel-excess:
  [lo u = l u; \ u \neq v; excess fo v \neq excess f v] \implies (u,v) \in \text{adm-edges}(fo,lo)
assumes adm-edges-leaving-u: (u',v) \in \text{adm-edges}(f,l) \implies adm-edges(fo,lo) \implies u' = u
assumes relabel-u-no-incoming-adm: lo u \neq l u \implies (v,u) \notin \text{adm-edges}(f,l)
assumes algo-rel: ((f,l),(fo,lo)) \in \text{pr-algo-rel}*
begin

lemma u-node-simp1[simp]: u \neq s \ u \neq t \ s \neq u \ t \neq u using u-node by auto
lemma u-node-simp2[simp, intro!]: u \in V using u-node by auto

61
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 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 \( PRE: \text{relabel-precond} \ f \ l \ u \)
shows \( \text{discharge-invar} c \ s \ t \ u \ f o \ lo \ no \ f \)
\( (\text{relabel-effect} \ f \ l \ u) \ (n(u := \text{adjacent-nodes} \ u)) \)

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

note \( lu-incr \)
also note \( \text{relabel-increase-u}[OF \ PRE] \)
finally have \( \text{INCR:} \ lo \ u < ?l' \ u \).

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

lemma \( \text{push-precondI-nz} \):
\[ \text{excess} \ f \ u \neq 0; (u,v)\in\text{cfE-of} \ f; l \ u = l \ v + 1 \] \implies \( \text{push-precond} \ f \ l \ (u,v) \)
unfolding \( \text{push-precond-def} \) by \( \text{(auto simp: excess-nz-iff-gz)} \)

lemma \( \text{remove-neighbor-pres-dis-invar} \):
assumes \( \text{PRE:} \ (u,v)\notin\text{adm-edges} \ f \ l \)

63
defines \( n' \equiv n \ (u := n \ u - \{v\}) \)
shows discharge-invar c s t u f l n 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: \( v \in n \ u \Rightarrow v \in V \)
using neighbors-adj[of u] E-ss-VxV unfolding adjacent-nodes-def by auto

lemma neighbors-in-E: \( v \in n \ u \Rightarrow (u,v) \in E \cup E^{-1} \)
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 \and u \neq s \and UIV: u \in V
  shows discharge f l n u
  \leq SPEC \( (\lambda(f',l',n'). \text{discharge-invar } c \ s \ t \ u \ f \ l \ n \ f' \ l' \ n') \)
  \and excess f' u = 0)
unfolding discharge-def push-def relabel-def
apply (refine-vcg WHLET-rule|where
  \( I = (\lambda(f',l',n'). \text{discharge-invar } c \ s \ t \ u \ f \ l \ n \ f' \ l' \ n') \)
  \and \( R = \text{inv-image } (pr-algo-rel <\text{lex}> \text{finite-psubset}) \)
  \( \lambda(f',l',n'). ((f',l'),(n',u)) \)\]

64
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-neighbors-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) ← whileT
  (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: \( \forall u \in \text{set} (L-left), \text{excess} f u = 0 \)
assumes L-sorted: is-top-sorted (adm-edges f l) (L-left @ L-right)
assumes L-set: set L-left \cup set L-right = V - \{s,t\}
begin
lemma rtf-is-nbr: neighbor-invar c s t 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 \( \forall u \in V - \{s,t\}, \text{excess} f u = 0 \) by auto
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 \leq SPEC isMaxFlow
unfolding relabel-to-front-def
apply (rewrite in whileT - □ vcg-intro-frame)
apply (refine-vcg
WHILET-rule[where
 l=\lambda(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 alt lex less-than)
(\lambda(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 .

67
assume \( L\text{-right} \neq [] \) then obtain \( u \text{-right}' \)

where [simp]: \( L\text{-right} = u \# L\text{-right}' \) by (cases \( L\text{-right} \)) auto

from \( L\)-set have [simp]: \( u \in V \) \( u \neq s \) \( u \neq t \) \( s \neq u \) \( t \neq u \) by auto

from \( L\)-distinct have [simp]: \( u \notin L\text{-left} \) \( u \notin L\text{-right}' \) by auto

show ?thesis

apply (rule vcg-rem-frame)
apply (rewrite in do \{\(-\)-, \(-\)-, \(-\)-\} 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: \( l' \) \( u \neq l \) \( u \)

have AUX1: \( x = u \) if \((x, u) \in (\text{adm-edges } f' \ l')^* \) for \( x \)
using that \( l'.\text{relabel-u-no-incoming-adm}[OF RELABEL\{symmetric\}] \)
by (auto elim: rtranclE)

have TS1: is-top-sorted (\text{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 \( L\text{-left}@L\text{-right}' \). —
thus, these new edges cannot add effective constraints.
from \( l'.\text{adm-edges-leaving-u} \)
and \( l'.\text{relabel-u-no-incoming-adm}[OF RELABEL\{symmetric\}] \)
have \( \text{adm-edges } f' \ l' \subseteq \text{adm-edges } f \ l \cup \{u\} \times \text{UNIV} \)
and \( \text{adm-edges } f' \ l' \cap \text{UNIV} \times \{u\} = \{\} \) by auto
from is-top-sorted-isolated-constraint[OF this - TS1]
have AUX2: is-top-sorted (\text{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 $l'.\text{algo-rel}$ by (auto dest: rtranclD)
subgoal proof ~
  assume NO-RELABEL[simp]: $l' \equiv l\ u$
  — Intuition: non-zero excess would imply an admissible edge contrary to top-sorted.
  have AUX: $\text{excess } f' \equiv v = 0$ if $v \in \text{set } L\text{-left}$ for $v$
  proof (rule ccontr)
    from that ($u \notin \text{set } L\text{-left}$) have $u \neq v$ by blast
    moreover assume $\text{excess } f' \equiv v = 0$
    moreover from that left-no-excess have $\text{excess } f \equiv v = 0$ by auto
    ultimately have $(u, v) \in \text{adm-edges } f \ l$
      using $l'.\text{no-relabel-excess}[OF NO-RELABEL[symmetric]]$
      by auto
    with $L$-sorted that show False
      by (auto simp: is-top-sorted-append is-top-sorted-cons)
  qed
show rtf-invar $c \ s\ t\ f'\ l'\ n'$ ($L\text{-left } \oplus [u]$) $L\text{-right}'$
  apply unfold-locales
subgoal by (auto simp: AUX)
subgoal
  apply (rule is-top-sorted-antimono)
    OF $l'.\text{no-relabel-adm-edges}[OF NO-RELABEL[symmetric]]$
  using $L$-sorted by simp
subgoal using $L$-set by auto
done
qed
subgoal using $l'.\text{algo-rel}$ by (auto dest: rtranclD)
done
qed
done
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
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 ∆ ≡ do
assert (u∈V);
return (x(u := x u + ∆))

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);
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.

\[
\text{definition cnt-init :: } \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nres}
\]
\[
\text{where cnt-init } C \equiv \{ \begin{align*}
\text{assert } (C < 2 \times N); \\
\text{return } ((\lambda v. 0)(\theta := C - 1, C := 1))
\end{align*}
\}
\]

Get the count for a label value.

\[
\text{definition cnt-get :: } (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nres}
\]
\[
\text{where cnt-get } cnt lv \equiv \{ \begin{align*}
\text{assert } (lv < 2 \times N); \\
\text{return } (cnt lv)
\end{align*}
\}
\]

Increment the count for a label value by one.

\[
\text{definition cnt-incr :: } (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nres}
\]
\[
\text{where cnt-incr } cnt lv \equiv \{ \begin{align*}
\text{assert } (lv < 2 \times N); \\
\text{return } (cnt (lv := cnt lv + 1))
\end{align*}
\}
\]

Decrement the count for a label value by one.

\[
\text{definition cnt-decr :: } (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nres}
\]
\[
\text{where cnt-decr } cnt lv \equiv \{ \begin{align*}
\text{assert } (lv < 2 \times N \land cnt lv > 0); \\
\text{return } (cnt (lv := cnt lv - 1))
\end{align*}
\}
\]

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

\[
\text{definition xf-rel } \equiv \{ ((\text{excess } f, \text{cf-of } f), f) \mid f. \text{ True } \}
\]

\[
\text{lemma xf-rel-RELATES[refine-dref-RELATES]: RELATES xf-rel}
\]
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)
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 \( (x,cf) \) \}

lemma pp-init-xcf2-aux-spec:
shows \( pp-init-xcf2-aux \leq SPEC (\lambda (x,cf). x=pp-init-x \land cf = pp-init-cf) \)
proof 
  have ADJ-S-AUX: adjacent-nodes \( s = \{ v . (s,v) \in E \} \)
  unfolding adjacent-nodes-def using no-incoming-s by auto
  have CSU-AUX: \( c(s,u) = 0 \) if \( u \notin \text{adjacent-nodes } s \) for \( u \)
  using that unfolding adjacent-nodes-def by auto

show \(?thesis \)
  unfolding pp-init-xcf2-aux-def
  apply (refine-vcg FOREACH-rule[where \( I=\lambda it (x,cf) \). 
  \( x s = (\sum v \in \text{adjacent-nodes } s - it, - c(s,v)) \) 
  \land (\forall v \in \text{adjacent-nodes } s . x v = (if v \in it then 0 else c(s,v))) \) 
  \land (\forall v \in - insert s (adjacent-nodes s) . x v = 0) \) 
  \land (\forall v \in \text{adjacent-nodes } s . 
  \quad \text{if } v \notin it \text{ then } cf(s,v) = 0 \land cf(v,s) = c(s,v) \) 
  \quad \text{else } cf(s,v) = c(s,v) \land cf(v,s) = c(v,s) \) 
  \land (\forall u v . u \neq s \land v \neq s \rightarrow cf(u,v) = c(u,v) ) 
  \land (\forall u . u \notin \text{adjacent-nodes } s \rightarrow cf(u,s) = 0 \land 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 \( l=\text{snd} \)] 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-refine2[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 {
  x ← foreach (adjacent-nodes u) (λv x. do {
    assert ((a,v)∈E∪E−1);
    assert (v∈V);
  })

  (x,cf)
}
if \((\text{cf}(u,v) \neq 0)\) then
case \(x\) of
  \(\text{None} \Rightarrow \text{return } (\text{Some } (l v))\)
  \(\text{Some } xx \Rightarrow \text{return } (\text{Some } (\min (l v) (xx)))\)
else
  \text{return } x
}) \text{ None;}
assert \((x \neq \text{None})\);
\text{return } (\text{the } x)
}

\text{lemma (in \text{--}) set-filter-xform-aux:}
\{ f x \mid x = a \lor x \in S \land x \not\in \text{it} \land P x \} = (\text{if } P a \text{ then } \{ f a \} \text{ else } \{ \}) \cup \{ f x \mid x \in S - \text{it} \land P x \}
\text{by auto}

\text{lemma (in \text{Labeling}) min-adj-label-aux-spec:}
\text{assumes \text{PRE}: relabel-precond } f l u
\text{shows min-adj-label-aux } cf l u \leq \text{SPEC } (\lambda x. x = \text{Min } \{ l v \mid v. (u,v) \in \text{cf}.E \})
\text{proof --}
have \text{AUX: } cf (u,v) \neq 0 \longleftrightarrow (u,v) \in \text{cf}.E \text{ for } v \text{ unfolding } \text{cf}.E\text{-def by auto}
have \text{EQ: } \{ l v \mid v. (u,v) \in \text{cf}.E \} = \{ l v \mid v. v \in \text{adjacent-nodes } u \land cf (u,v) \neq 0 \}
\text{unfolding } \text{AUX}
\text{using } \text{cfE-ss-invE}
\text{by } (\text{auto simp: adjacent-nodes-def})
\text{define } \text{Min-option} :: \text{nat set } \mapsto \text{nat}
\text{where } \text{Min-option } X \equiv \text{if } X = \{ \} \text{ then None else } \text{Some } (\text{Min } X) \text{ for } X
\text{from \text{PRE active-has-cf-outgoing have } cf\text{-going } u \neq \{ \}
\text{unfolding } \text{relabel-precond-def by auto}
hence [\text{simp}]: u \in V \text{ unfolding } cf\text{-going-def using } cfE\text{-of-ss-VxV by auto}
\text{from } (cf\text{-going } u \neq \{ \}):
\text{have AUX2: } \exists v. v \in \text{adjacent-nodes } u \land cf (u, v) \neq 0
\text{by } (\text{smt AUX Collect-empty-eq Image-singleton-iff UnCI adjacent-nodes-def}
\text{cf-going-def cf-def converse-iff prod.simps(2)})
\text{show } ?\text{thesis unfolding min-adj-label-aux-def EQ}
\text{apply (refine-vcg)
FOREACH-rule[where}
  I=\lambda it x. x = \text{Min-option}
  \{ l v \mid v. v \in \text{adjacent-nodes } u - \text{it} \land cf (u,v) \neq 0 \}
]\}
\text{apply (vc-solve
simp: Min-option-def it-step-insert-iff set-filter-xform-aux

75
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

definition min-adj-label am cf l u ≡ \{ 
assert (u∈V);
adj ← am-get am u;
x ← nfoldli adj (λ_. True) (λv x. { 
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 Idl] - Idl, refine]
apply (refine-vcg)
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 ∧ cf-of f (u, v) ≠ 0 \} = \{ l v | cf-of f (u, v) ≠ 0 \}
  using cfE-of-ss-VxV[of]
  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:

\[
\left[ (x, cf), f \in xf-rel \right] \Rightarrow push2-aux x cf ei \leq \downarrow xf-rel (push f l e)
\]

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'.\Delta-def l'.cf' alt l'.uv-not-eq(1)
  by (auto)
qed

done

definition push2 x cf \equiv \lambda (u,v). do {
  assert \((u,v) \in E \cup E^{-1}\);
  xu \leftarrow x-get x u;
  cfuv \leftarrow cf-get cf (u,v);
  cfvu \leftarrow cf-get cf (v,u);
  let \Delta = min xu cfuv;
  x \leftarrow x-add x u (-\Delta);
  x \leftarrow x-add x v \Delta;

  cf \leftarrow cf-set cf (u,v) (cfuv - \Delta);
  cf \leftarrow cf-set cf (v,u) (cfvu + \Delta);

  return (x,cf)
}

lemma push2-refine[refine]:
assumes \((x,cf),f \in xf-rel \ (ei,e) \in Id \times Id\)
shows push2 x cf ei \leq \downarrow xf-rel (push f l e)
proof
  have push2 x cf ei \leq (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
6.2.6 Adding frequency counters to labeling

**Definition** \( l\text{-invar} \equiv \forall v. \ l \ v \neq 0 \implies v \in V \)

**Definition** \( clc\text{-invar} \equiv \lambda (cnt,l). \)
\[ (\forall lv. \ cnt \ lv = \text{card} \ \{ u \in V . \ l \ u = lv \} \]
\[ \land \ (\forall u. \ l \ u < 2*N) \land l\text{-invar} \]

**Definition** \( clc\text{-rel} \equiv \text{br snd clc\text{-invar}} \)

**Definition** \( clc\text{-init} \ C \equiv \)
\[
\{ \ l \leftarrow l\text{-init} \ C; \\
\ cnt \leftarrow cnt\text{-init} \ C; \\
\ return \ (cnt,l) \}
\]

**Definition** \( clc\text{-get} \equiv \lambda (cnt,l) \ u. \ l\text{-get} \ l \ u \)

**Definition** \( clc\text{-set} \equiv \lambda (cnt,l) \ u \ a. \)
\[
\{ \ assert \ (a < 2*N); \\
\ lu \leftarrow l\text{-get} \ l \ u; \\
\ cnt \leftarrow cnt\text{-decr} \ cnt \ lu; \\
\ l \leftarrow l\text{-set} \ l \ u \ a; \\
\ lu \leftarrow l\text{-get} \ l \ u; \\
\ cnt \leftarrow cnt\text{-incr} \ cnt \ lu; \\
\ return \ (cnt,l) \}
\]

**Definition** \( clc\text{-has-gap} \equiv \lambda (cnt,l) \ u. \)
\[
\{ \ nlu \leftarrow cnt\text{-get} \ cnt \ lu; \\
\ return \ (nlu = 0) \}
\]

**Lemma** \( \text{cardV-le-N} \): \( \text{card} \ V \leq N \) using \( \text{card-mono[OF - V-ss]} \) by auto

**Lemma** \( N\text{-not-Z} \): \( N \neq 0 \) using \( \text{card-V-ge2 cardV-le-N} \) by auto

**Lemma** \( N\text{-ge-2} \): \( 2 \leq N \) using \( \text{card-V-ge2 cardV-le-N} \) by auto

**Lemma** \( clc\text{-init-refine[refine]} \)
\[
\{ \ \text{assumes} \ [\text{simplified,simp}]: (Ci,C)\in\text{nat-rel} \\
\ \ \text{assumes} \ [\text{simp}]: C = \text{card} \ V \\
\ \ \text{shows} \ clc\text{-init} \ Ci \leq \psi \text{clc\text{-rel} (l\text{-init})} \}
\]

**Proof**
\[
\{ \ \text{have} \ AUX \ : \ \{ u. \ u \neq s \land u \in V \} = V - \{s\} \ by \ auto \\
\ \ \text{show} \ \psi \text{thesis} \}
\]
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]:
  \[(clc,l) \in clc-rel; (ui,u) \in nat-rel \] \implies clc-get clc ui \leq 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 \Rightarrow nat) \Rightarrow node \Rightarrow nat nres

where l-get-rlx l u 
  assert (u < N);
  return (l u)

definition clc-get-rlx \equiv \lambda (cnt,l) u. l-get-rlx l u

lemma clc-get-rlx-refine[refine]:
  \[(clc,l) \in clc-rel; (ui,u) \in nat-rel \] \implies clc-get-rlx clc ui \leq 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:
  \[ finite Y; X = insert x Y; x \notin Y \] \implies card X = Suc (card Y)
by auto

lemma clc-set-refine[refine]:
  \[(clc,l) \in clc-rel; (ui,u) \in nat-rel; (ai,a) \in nat-rel \] \implies clc-set clc ui ai \leq Id 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]:
  \[ ((\text{clc}, l) \in \text{clc-rel}; k < 2 + N) \Rightarrow \text{clc-has-gap clc k} \leq (\text{spec r. r} \mapsto \text{gap-precond l k}) \]
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 \equiv do 
  nfoldl \[0..<N\] (\lambda. True) (\lambda v l. do 
    lv \leftarrow \text{L-get-rlx} l v;
    if (k < lv \land lv < C) then do 
      assert (C + 1 < 2 \times N);
      l \leftarrow \text{L-set} l v (C + 1);
      return l
    } else return l
  } l
}

lemma gap-effect-invar[simp]: l-invar l \Rightarrow l-invar (gap-effect l k)
unfolding gap-effect-def l-invar-def
by auto

lemma relabel-effect-invar[simp]: \[ [l-invar l; u \in V] \Rightarrow l-invar (relabel-effect f l u) \]
unfolding relabel-effect-def l-invar-def by auto

lemma gap-aux-correct[THEN order-trans, refine-vcg]:
  \[ l-invar l; C = \text{card} V \Rightarrow \text{gap-aux clc l k} \leq \text{SPEC (\lambda r. r} = \text{gap-effect l k) \]
unfolding gap-aux-def l-get-rlx-def l-set-def
apply (simp only: nres-monad-laws)
apply (refine-vcg nfoldli-rule[where I = \lambda i t1 i t2 l'. \forall u. if u \in \text{set} it2 then l' u = l u else l'u = \text{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

81
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 ($k < lv \land lv < C$) then do {
      clc ← clc-set clc v (C+1);
      return clc
    } else return clc
  }) clc
}

lemma gap2-refine[refine]:
assumes [simplified simp]: $(Ci,C) \in \text{nat-rel} \quad (ki,k) \in \text{nat-rel}$
assumes CLC: $(clc,l) \in \text{clc-rel}$
shows gap2 $Ci clc ki \leq \llbracket \text{clc-rel} \mid \text{gap-aux} \ C l k \rrbracket$
unfolding gap2-def gap-aux-def
apply (refine-reg 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]: $C = \text{card V} \quad l\text{-invar l}$
shows gap-relabel-aux $C f l u \leq \text{gap-relabel} f l u$
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 $(-,l) \Rightarrow \text{min-adj-label} am cf l u$

definition clc-relabel2 $am cf clc u$ ≡ do {
assert \((u \in V - \{s,t\})\);

\(nl \leftarrow \min\text{-adj-label-clc} \ am \ am \ clc \ u\);

\(clc \leftarrow \text{clc-set} \ clc \ u \ (nl+1)\);

return clc
}

**Lemma clc-relabel2-refine**

**Assumes**
- \(XF: ((x, cf), f) \in xf-rel\)
- \(AM: (am, adjacent-nodes) \in \text{nat-rel} \rightarrow (\text{nat-rel}) \rightarrow (\text{list-set-rel})\)
- \([\text{simplified}, \text{simp}]: (ui, u) \in Id\)

**Shows**
- \(clc\text{-relabel2} \ am \ cf \ clc \ ui \leq \downarrow clc\text{-rel} \ (\text{relabel} f l u)\)

**Proof**

Have \(clc\text{-relabel2} \ am \ cf \ clc \ ui \leq \downarrow clc\text{-rel} \ (\text{relabel} f l u)\)

Unfolding \(clc\text{-relabel2-def} \ relabel2\text{-def}\)

Apply \(\text{refine}\text{-rcg}\)

Apply \(\text{refine}\text{-dref-type}\)

Apply \(\text{vc-solve simp: CLC}\)

Subgoal

Using CLC

Unfolding \(clc\text{-rel-def} \ in\text{-br-conv} \ min\text{-adj-label-clc-def}\)

By \(\text{auto split: prod.split}\)

Done

Also Note \(\text{relabel2-refine}\)[OF \(XF\ AM\), of \(l l ui u]\)

Finally Show \(\boxempty\text{thesis by simp}\)

**QED**

definition gap-relabel2 C am cf clc u \equiv do {
lu \leftarrow \text{clc-get} \ clc \ u;

clc \leftarrow \text{clc-relabel2} \ am \ cf \ clc \ u;

has-gap \leftarrow \text{clc-has-gap} \ clc \ lu;

if has-gap then gap2 C clc lu
else
    RETURN clc
}

**Lemma gap-relabel2-refine-aux**

**Assumes**
- \(XCF: ((x, cf), f) \in xf-rel\)
- \(AM: (am, adjacent-nodes) \in \text{nat-rel} \rightarrow (\text{nat-rel}) \rightarrow (\text{list-set-rel})\)
- \([\text{simplified}, \text{simp}]: (Ci, C) \in Id \quad (ui, u) \in Id\)

**Shows**
- \(gap\text{-relabel2} \ Ci \ am \ cf \ clc \ ui \leq \downarrow clc\text{-rel} \ (\text{gap\text{-relabel2-def} gap\text{-relabel2-def}})\)

Apply \(\text{refine\text{-vcg XCF AM CLC if\text{-bind-cond-refine bind\text{-refine}}}'}\)

Apply \(\text{vc\text{-solve solve: refl}}\)

Subgoal for - lu

Using CLC

Unfolding \(\text{clc\text{-get-def} l\text{-get-def} clc\text{-rel-def} in\text{-br-conv} clc\text{-invar-def}}\)
by \((auto\ simp:\ refine-pw-simps\ split: prod.splits)\)
done

lemma gap-relabel2-refine\[refine\]:
assumes \(XCF\): \(\{(x, cf), f\} \in xf-rel\)
assumes \(CLC\): \((clc,t)\in clc-rel\)
assumes \(AM\): \((am,adjacent-nodes)\in nat-rel\rightarrow\{nat-rel\}list-set-rel\)
assumes \(\text{simplified}, simp\): \((ui,u)\in Id\)
assumes \(CC\): \(C = \text{card } V\)
shows gap-relabel2 \(C\ am\ cf\ clc\ ui \leq clc-rel\ (gap-relabel f l u)\)
proof –
  from \(CLC\) have \(LIN\): \(\text{linvar } l\ unfolding\ clc-rel-def\ in-br-conv\ clc-invar-def\)
  by \(auto\)
  note gap-relabel2-refine-aux\[OF\ XCF\ CLC\ AM\ Id\ Id\]
  also note gap-relabel-aux-refine\[OF\ CC\ LIN\]
  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 \(\equiv\ is-nf\ N\ (0::\text{capacity-impl})\)

lemma x-init-hnr\[sepref-fr-rules\]:
  \((uncurry0\ (Array.new\ N\ 0),\ uncurry0\ x\text{-init})\in unit-assn^k\ \rightarrow\ a\ x\text{-assn}\)
apply sepref-to-hoare unfolding \(x\)-assn-def \(x\)-init-def

84
by \((\text{sep-auto heap: } \text{nfil-rule})\)

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

**Definition** (in \(-\)) \(x\)-add-impl \(x u \Delta \equiv \{\)
\[
x u \leftarrow \text{Array}.\text{nth } x u;
x \leftarrow \text{Array}.\text{upd } u (x u + \Delta) \ x;
\text{return } x
\}

**Lemma** \(x\)-add-hnr[sepref-fr-rules]:
\[
(\text{uncurry2 } \text{x-add-impl}, \text{uncurry2 } (\text{PR-CONST } \text{x-add}))
\in x\text{-assn}^d \circ_a \text{node-assn}^k \circ_a \text{cap-assn} \rightarrow_a x\text{-assn}
\]
\text{apply sepref-to-hoare}
\text{unfolding } x\text{-assn-def } x\text{-add-impl-def } x\text{-add-def by (sep-auto simp: refine-pw-simps)}

### 6.3.3 Labeling by Array

**Definition** \(l\)-assn \(\equiv \text{is-nf } N (0::\text{nat})\)

**Definition** (in \(-\)) \(l\)-init-impl \(N s \text{ cardV} \equiv \{\)
\[
l \leftarrow \text{Array}.\text{new } N (0::\text{nat});
l \leftarrow \text{Array}.\text{upd } s \text{ cardV } l;
\text{return } l
\}

**Lemma** \(l\)-init-hnr[sepref-fr-rules]:
\[
(\text{l-init-impl } N s, (\text{PR-CONST } \text{l-init})) \in \text{nat-assn}^k \rightarrow_a l\text{-assn}
\]
\text{apply sepref-to-hoare}
\text{unfolding } l\text{-assn-def } l\text{-init-def } l\text{-init-impl-def by (sep-auto heap: nf-init-rule)}

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

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

**Lemma** \(l\)-set-hnr[sepref-fr-rules]:
\[
(\text{uncurry2 } (\lambda a i x. \text{Array}.\text{upd } i x a), \text{uncurry2 } (\text{PR-CONST } \text{l-set}))
\]

85
6.3.4 Label Frequency by Array

Definition cnt-assn \( f::\text{node} \Rightarrow \text{nat} \) \( a \) ≡ \( \exists \ A l. \ a \mapsto \lambda \ (\text{length } l = 2*N \land (\forall i < 2*N. \ l!i = f i) \land (\forall i \geq 2*N. \ f i = 0)) \)

Definition (in −) cnt-init-impl \( N \ C \) ≡ do 
\( a \leftarrow \text{Array}.\text{new} (2*N) (0::\text{nat}); \)
\( a \leftarrow \text{Array}.\text{upd} 0 (C-1) \ a; \)
\( a \leftarrow \text{Array}.\text{upd} C 1 \ a; \)
\( \text{return } a \)

Definition (in −) cnt-incr-impl \( a \ k \) ≡ do 
\( \text{freq} \leftarrow \text{Array}.\text{nth} a k; \)
\( a \leftarrow \text{Array}.\text{upd} k (\text{freq}+1) \ a; \)
\( \text{return } a \)

Definition (in −) cnt-decr-impl \( a \ k \) ≡ do 
\( \text{freq} \leftarrow \text{Array}.\text{nth} a k; \)
\( a \leftarrow \text{Array}.\text{upd} k (\text{freq}-1) \ a; \)
\( \text{return } a \)

Lemma cnt-init-hnr[sepref-fr-rules]: \( (\text{cnt-init-impl } N, \text{PR-CONST cnt-init}) \in \text{nat-assn}^k \rightarrow_a \text{cnt-assn} \)
Apply sepref-to-hoare
Unfolding cnt-init-def cnt-init-impl-def cnt-assn-def
By (sep-auto simp: refine-pw-simps split: prod.split)

Lemma cnt-get-hnr[sepref-fr-rules]: \( (\text{uncurry Array}.\text{nth}, \text{uncurry (PR-CONST cnt-get)}) \in \text{cnt-assn}^k \rightarrow_a \text{nat-assn} \)
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]: \( (\text{uncurry cnt-incr-impl}, \text{uncurry (PR-CONST cnt-incr)}) \in \text{cnt-assn}^d \rightarrow_a \text{cnt-assn} \)
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]: \( (\text{uncurry cnt-decr-impl}, \text{uncurry (PR-CONST cnt-decr-hnr[sepref-fr-rules])}) \in \text{cnt-assn}^d \rightarrow_a \text{cnt-assn} \)
Apply sepref-to-hoare
Unfolding cnt-decr-def cnt-decr-impl-def cnt-assn-def
By (sep-auto simp: refine-pw-simps)

86
cnt-decr) ∈ cnt-assn \ast_a nat-assn \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)

\textbf{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 \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


sepref-thm clc-get-impl is uncurry (PR-CONST clc-get)
:: clc-assn \ast_a node-assn \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 \gamma f,\cdot \cdot)∈··


sepref-thm clc-get-rlx-impl is uncurry (PR-CONST clc-get-rlx)
:: clc-assn \ast_a node-assn \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 \gamma f,\cdot \cdot)∈··


sepref-thm clc-set-impl is uncurry2 (PR-CONST clc-set)
:: clc-assn \ast_a node-assn \ast_a nat-assn \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 \gamma f,\cdot \cdot)∈··


sepref-thm clc-has-gap-impl is uncurry (PR-CONST clc-has-gap)
:: clc-assn \ast_a nat-assn \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 \gamma f,\cdot \cdot)∈··

lemmas [sepref-fr-rules] = clc-has-gap-impl.refine[OF Network-Impl-axioms]
6.3.6 Push

\text{sepref-thm} \text{push-impl} \text{ is } \text{uncurry2} (PR-CONST push2)
\[ x \text{-assn}^d \circ_a \text{cf-assn}^d \circ_a \text{edge-assn}^k \rightarrow_a (x \text{-assn} \times_a \text{cf-assn}) \]
\text{unfolding} push2-def PR-CONST-def
\text{by} \text{ sepref}
\text{concrete-definition (in \(-\)) push-impl}
\text{uses} Network-Impl.push-impl.refine-raw \text{ is } (\text{uncurry2 } ?f,-\circ)-\leq-
\text{lemmas} [\text{sepref-fr-rules} = \text{push-impl.refine}[OF Network-Impl-axioms]}

6.3.7 Relabel

\text{sepref-thm} \text{min-adj-label-impl} \text{ is } \text{uncurry3} (PR-CONST min-adj-label)
\[ am \text{-assn}^k \circ_a \text{cf-assn}^k \circ_a \text{l-assn}^k \circ_a \text{node-assn}^k \rightarrow_a \text{nat-assn} \]
\text{unfolding} min-adj-label-def PR-CONST-def
\text{by} \text{ sepref}
\text{concrete-definition (in \(-\)) min-adj-label-impl}
\text{uses} Network-Impl.min-adj-label-impl.refine-raw \text{ is } (\text{uncurry3 } ?f,-\circ)-\leq-
\text{lemmas} [\text{sepref-fr-rules} = \text{min-adj-label-impl.refine}[OF Network-Impl-axioms]}

6.3.8 Gap-Relabel

\text{sepref-thm} \text{gap-impl} \text{ is } \text{uncurry2} (PR-CONST gap2)
\[ \text{nat-assn}^k \circ_a \text{cf-assn}^k \circ_a \text{l-assn}^d \circ_a \text{node-assn}^k \rightarrow_a \text{l-assn} \]
\text{unfolding} gap2-def PR-CONST-def
\text{by} \text{ sepref}
\text{concrete-definition (in \(-\)) gap-impl}
\text{uses} Network-Impl.gap-impl.refine-raw \text{ is } (\text{uncurry2 } ?f,-\circ)-\leq-
\text{lemmas} [\text{sepref-fr-rules} = \text{gap-impl.refine}[OF Network-Impl-axioms]}

\text{sepref-thm} \text{min-adj-label-clc-impl} \text{ is } \text{uncurry3} (PR-CONST min-adj-label-clc)
\[ am \text{-assn}^k \circ_a \text{cf-assn}^k \circ_a \text{clc-assn}^d \circ_a \text{node-assn}^k \rightarrow_a \text{nat-assn} \]
\text{unfolding} min-adj-label-clc-def PR-CONST-def clc-assn-def
\text{by} \text{ sepref}
\text{concrete-definition (in \(-\)) min-adj-label-clc-impl}
\text{uses} Network-Impl.min-adj-label-clc-impl.refine-raw \text{ is } (\text{uncurry3 } ?f,-\circ)-\leq-
\text{lemmas} [\text{sepref-fr-rules} = \text{min-adj-label-clc-impl.refine}[OF Network-Impl-axioms]}

\text{sepref-thm} \text{clc-relabel-impl} \text{ is } \text{uncurry3} (PR-CONST clc-relabel2)
\[ am \text{-assn}^k \circ_a \text{cf-assn}^k \circ_a \text{clc-assn}^d \circ_a \text{node-assn}^k \rightarrow_a \text{clc-assn} \]

88
unfolding clc-relabel2-def PR-CONST-def
by sepref
concrete-definition (in −) clc-relabel-impl
uses Network-Impl.clc-relabel-impl.refine is (uncurry3 \(f, .\)) ∈-

sepref-thm gap-relabel-impl is uncurry4 (PR-CONST gap-relabel2)
:: nat-assn\(k\) \ast_a am-assn\(k\) \ast_a cf-assn\(k\) \ast_a clc-assn\(d\) \ast_a node-assn\(k\)
→_a clc-assn
unfolding gap-relabel2-def PR-CONST-def
by sepref
concrete-definition (in −) gap-relabel-impl
uses Network-Impl.gap-relabel-impl.refine is (uncurry4 \(f, .\)) ∈-

6.3.9 Initialization

sepref-thm pp-init-xcf2-impl is (PR-CONST pp-init-xcf2)
:: am-assn\(k\) →_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 is (\(f, .\)) ∈-

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 ∈ V);
    return (Q @\{v\})

Dequeue a node.

definition q-dequeue :: node list ⇒ (node × node list) nres where
  q-dequeue Q ≡ do
    assert (Q ≠ []);
    return (hd Q, tl Q)

end — Network Implementation Locale

7.2 Refinements to Basic Operations

context Network-Impl
begin

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

7.2.1 Refinement of Push

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

lemma fifo-push2-aux-refine:
[((x,cf),f)∈xf-rel; (ei,e)∈Id×,Id; (Qi,Q)∈Id]  
⇒ fibonacci-push2-aux x cf Qi ei ≤ β(xf-rel ×, Id) (fibonacci-push f l Q v)
unfolding fibonacci-push-def fibonacci-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 ∆ = min xu cfuv;
  x ← x-add x u (−∆);
  x ← x-add x v ∆;

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

  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×Id (Qi,Q)∈Id
  shows fifo-push2 x cf Qi ei ≤⇓(xf-rel ×, 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: ares-monad-laws)
  apply refine-vcg
  using E-ss-VxV
  by auto
also note \texttt{fifo-push2-aux-refine[OF assms]}
finally show \texttt{thesis}.
\begin{proof}

\subsection{Refinement of Gap-Relabel}

\textbf{Definition} \texttt{fifo-gap-relabel-aux} \( C \ f \ l \ Q \ u \equiv \{ \)
\begin{align*}
Q & \leftarrow q\text{-enqueue} \ u \ Q; \\
l u & \leftarrow l\text{-get} \ l \ u; \\
l & \leftarrow \text{relabel} \ f \ l \ u; \\
\text{if} \ & \text{gap-precond} \ l \ lu \ \text{then do} \{ \\
& \ l \leftarrow \text{gap-aux} \ C \ l \ lu; \\
& \ \text{return} \ (l,Q) \\
\} \ \text{else return} \ (l,Q) \\
\}
\end{align*}

\textbf{Lemma} \texttt{fifo-gap-relabel-aux-refine}:
\begin{align*}
\text{assumes} \ \& \ \text{simp}: \ C &= \text{card} \ V \ l\text{-invar} \ l \\\n\text{shows} \ & \texttt{fifo-gap-relabel-aux} \ C \ f \ l \ Q \ u \leq \texttt{fifo-gap-relabel} \ f \ l \ Q \ u \\
\text{unfolding} \ & \texttt{fifo-gap-relabel-aux-def} \texttt{fifo-gap-relabel-def} \texttt{relabel-def} \\
& \texttt{gap-relabel-effect-def} \texttt{l-get-def} \texttt{q-enqueue-def} \\
\text{apply} \ (\text{simp only}: \text{Let-def} \ \text{nres-monad-laws}) \\
& \text{apply} \ \texttt{refine-vcg} \text{by} \ \text{auto} \\
\end{align*}

\textbf{Definition} \texttt{fifo-gap-relabel2} \( C \ am \ cf \ clc \ Q \ u \equiv \{ \)
\begin{align*}
Q & \leftarrow q\text{-enqueue} \ u \ Q; \\
l u & \leftarrow \text{clc-get} \ clc \ u; \\
clc & \leftarrow \text{clc-relabel2} \ am \ cf \ clc \ u; \\
\text{has-gap} & \leftarrow \text{clc-has-gap} \ clc \ lu; \\
\text{if} \ \text{has-gap} \ \text{then do} \{ \\
& \ clc \leftarrow \text{gap2} \ C \ clc \ lu; \\
& \ \text{RETURN} \ (clc,Q) \\
\} \ \text{else} \\
& \ \text{RETURN} \ (clc,Q) \\
\}
\end{align*}

\textbf{Lemma} \texttt{fifo-gap-relabel2-refine-aux}:
\begin{align*}
\text{assumes} \ XCF: \ ((x, \ cf), f) \in \text{xf-rel} \\
\text{assumes} \ CLC: \ (\text{clc},l)\in\text{clc-rel} \\
\text{assumes} \ AM: \ (am,\text{adjacent-nodes})\in\text{nat-rel}\rightarrow(\text{nat-rel})\text{list-set-rel} \\
\text{assumes} \ \& \ \text{simp}: \ (C_i,C)\in\text{Id} \ (Qi,Q)\in\text{Id} \ (ui,u)\in\text{Id} \\\n\text{shows} \ & \texttt{fifo-gap-relabel2} \ C_i \ am \ cf \ clc \ Qi \ ui \\
\leq \ & \upharpoonright (\text{clc-rel} \times, \text{Id}) \ (\texttt{fifo-gap-relabel-aux} \ C \ f \ l \ Q \ u) \\
\text{unfolding} \ & \texttt{fifo-gap-relabel2-def} \texttt{fifo-gap-relabel-aux-def} \\
\text{apply} \ \text{refine-vcg} \ \text{XCF} \ AM \ CLC \ \text{if-bind-cond-refine} \ \text{bind-refine}')
\end{align*}

\end{proof}
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) \in xf-rel
  assumes CLC: (clc,l)\in clc-rel
  assumes AM: (am,adjacent-nodes)\in nat-rel\rightarrow\langle nat-rel\rangle list-set-rel
  assumes [simplified,simp]: (Qi,Q)\in Id (ui,u)\in Id
  assumes CC: C = card V
  shows fifo-gap-relabel2 C am cf clc Qi ui
  \leq_{\langle clc-rel \times_r Id \rangle} \langle fifo-gap-relabel f l Q u \rangle
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 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)\in cfE-of f \} = set (filter (\lambda v. cf-of f (u,v) \neq 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 f f u f
  assumes AMR: (am,adjacent-nodes)\in Id \rightarrow \langle Id\rangle list-set-rel
  assumes CR: \\\wedge s si. (si,s)\in Id \Rightarrow cci si \leftrightarrow cc s
  assumes FR: \\\wedge v vi s si. [(vi,v)\in Id; v\in V; (u,v)\in E\cup E^{-1}; (si,s)\in Id] \Rightarrow
  \\\\\\{ f v si \leq_{\langle Id \rangle} \langle \text{if (cf-of f (u,v) \neq 0) then f f s else RETURN s \rangle \}
  (is \\\\\\langle vi\text{ v si}\rangle \in \text{Id} [::::] \Rightarrow - \leq_{\langle Id \rangle} (?ff' v s))
  assumes S0R: (s0i,s0)\in Id
  assumes UR: (ui,u)\in Id

93
shows \( \text{nfoldli} \ (am \ u i) \ cci \ ffm \ s0i \leq \Id \ (\text{FOREACHc} \ \{v . (u,v) \in \text{cfE-of f} \} \ cci \ ffm \ s0) \)

proof

from \( \text{fun-relD[OF AMR]} \) 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) \subseteq V \ (\{u\} \times \text{set (am u)} \subseteq E \cup E^{-1})

unfolding is-adj-map-def using E-ss-VxV by auto

thm nfoldli-refine

have nfoldli (am u i) cci ffm s0 \leq \Id (nfoldli (am u i) cci ffm' 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 u i) cci ffm' s0 \leq \Id (\text{FOREACHc (adjacent-nodes u)} cci ffm' s0)

by (rule LFOc-refine[OF fun-relD[OF AMR UR]; simp])

also have \( \text{FOREACHc (adjacent-nodes u)} cci ffm' s0 \)

apply (subst am-nodes-as-filter[OF AM])

apply (subst \text{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 \( \text{fun-relD[OF AMR Id[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 f_0 l Q u \equiv do {
assert (u \in V - \{s,t\});
assert (distinct (am u));
nfoldli (am u) (\lambda(f,l,Q). \text{excess f u} \neq 0) (\lambda v (f,l,Q). \text{do } {
assert ((u,v) \in E \cup E^{-1} \land v \in V); 
if (\text{cf-of f}_0 (u,v) \neq 0) then do {
if (l u = l v + 1) then do {
(f',Q) \leftarrow \text{fifo-push f l Q (u,v)};
assert (\forall v'. v' \neq v \rightarrow \text{cf-of f'} (u,v') = \text{cf-of f} (u,v'));
return (f',l,Q)
} else return (f,l,Q)
} else return (f,l,Q)
}) else return (f,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-reg 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
  \( = \lambda \text{if1 it2 (f, \cdot, \cdot). } \forall v \in \text{set it2}. \text{cf-of } f (u, v) = \text{cf-of } f_0 (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 \((u \in V \land \text{distinct } (am u))\);
  monadic-nfoldli (am u)
  \((\lambda (x,cf),l,Q). \text{do } \{ \text{xu } \leftarrow \text{x-get } x u ; \text{return } (xu \neq 0) \} \})
  \((\lambda v ((x,cf),l,Q). \text{do } \{ \text{cfuv } \leftarrow \text{cf-get cf } (u,v) ; \text{if } (\text{cfuv } \neq 0) \text{ then do } \{ \text{lu } \leftarrow \text{l-get } l u ; \text{lv } \leftarrow \text{l-get } l v ; \text{if } (\text{lu } = \text{lv } + 1) \text{ then do } \{ \text{((x,cf),Q } \leftarrow \text{fifo-push2 } x cf Q (u,v) ; \text{return } ((x,cf),l,Q) \} \text{ else return } ((x,cf),l,Q) \} \text{ else return } ((x,cf),l,Q) \} \}) (\text{cfv}) (\text{cf}) (\text{lh}) (\text{Q}) (\text{u}) (\text{v})\)
}

private lemma dis-loop-aux3-refine:
assumes [simplified,simp]: \((\text{ami,am}) \in \text{Id } (li,l) \in \text{Id } (Qi,Q) \in \text{Id } (ui,u) \in \text{Id}\)
assumes XF: \(((x,cf),f) \in \text{xf-rel}\)
shows dis-loop-aux3 am x cf li Qi ui \(\leq \downarrow (\text{xf-rel} \times_{\text{Id}} \text{Id} \times_{\text{Id}} \text{Id}) \) (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-reg)
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 \((\text{distinct } (am u))\);
  amu ← am-get am u;
  monadic-nfoldli amu
  \((\lambda ((x,cf),clc,Q). \text{do } \{ \text{xu } \leftarrow \text{x-get } x u ; \text{return } (xu \neq 0) \} \})
  \((\lambda v ((x,cf),clc,Q). \text{do } \{ \text{cfuv } \leftarrow \text{cf-get cf } (u,v) ; \text{if } (\text{cfuv } \neq 0) \text{ then do } \{ \text{lu } \leftarrow \text{clc-get } clc u ; \text{lv } \leftarrow \text{clc-get } clc v ; \} \} \}) (\text{cfv}) (\text{cf}) (\text{li}) (\text{l}) (\text{Q}) (\text{u}) (\text{v})\)

96
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)
}} ((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 cf clc Qi ui
≤⇓ (Id × r clc-rel × 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) ∈ zf-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
≤⇓ (zf-rel × r clc-rel × r Id) (dis-loop-aux am f l Q u)
proof –
have [simp]:
  ((Id × r clc-rel × r Id) O (zf-rel × r Id)) = zf-rel × r clc-rel × r Id
  by (auto simp: prod-rel-comp)

note dis-loop2-refine-aux[OF Idl Idl Idl Idl Idl Idl CLC]
also note dis-loop-aux3-refine[OF Idl Idl Idl Idl XF]
also note dis-loop-aux2-refine
finally show ?thesis
  by (auto simp: conc-fun-chain monoD[OF conc-fun-monotonic])
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;
  }
}
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 ×_r clc-rel ×_r Id) (fifo-discharge f l Q)
proof −
  have fifo-discharge2 C am x cf clc Q ≤⇓ (xf-rel ×_r clc-rel ×_r Id) (fifo-discharge-aux am f l Q)
    unfolding fifo-discharge2-def fifo-discharge-aux-def
    unfolding x-get-def
    apply (simp only: ares-monad-laws)
    apply (refine-rcg XCF CLC AM IdI)
    apply (vc-solve simp: CC)
    subgoal unfolding xf-rel-def in-br-conv by auto
    apply 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 (λv 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 \( q\text{-init }am \leq \text{RETURN } (\neg f) \) \( (am \ s) \)

unfolding \( q\text{-init-def }q\text{-empty-def }q\text{-enqueue-def }am\text{-get-def} \)
apply (refine-vcg nfoldli-rule[where \( I=\lambda l. l = \text{filter } ((\neg f) \ l) \)])
by auto
also have ... \[
\leq (\spec l. \text{distinct } l \land \text{set } l = \{ v \in V - \{ s, t \}. \text{excess } pp\text{-init}-f v \neq 0 \})
\]
proof –
from \( am\text{-to-adj-nodes-refine}[OF AM] \)
have [simp]: \( \text{distinct } (am \ s) \land \text{set } (am \ s) = \text{adjacent-nodes } s \)
unfolding \( \text{list-set-rel-def} \)
by (auto simp: in-br-conv)

definition \( \text{fifo-push-relabel-aux } am \equiv \{ \)
\( \text{cardV }\leftarrow \text{init-C } am; \)
\( \text{assert } (\text{cardV }= \text{card } V); \)
\( \text{let } f = \text{pp-init-f}; \)
\( l \leftarrow \text{l-init } \text{cardV}; \)
\( Q \leftarrow \text{q-init } am; \)
\( (f,l,-) \leftarrow \text{monadic-WHILEIT } (\lambda -. \text{True}) \)
\( (\lambda (f,l,Q). \text{do } \{ qe \leftarrow \text{q-is-empty } Q; \text{return } (\neg qe)\}) \)
\( (\lambda (f,l,Q). \text{do } \{ \)
\( \text{fifo-discharge } f \ l \ Q \)
\( \}) \)
\( (f,l,Q); \)
\( \text{assert } (\text{Height-Bounded-Labeling } c \ s \ t \ f \ l); \)
\( \text{return } f \)
\}

lemma \( \text{fifo-push-relabel-aux-refine}: \)
assumes \( AM: \text{is-adj-map } am \)
shows \( \text{fifo-push-relabel-aux } am \leq \Downarrow \text{Id } \) \((\text{fifo-push-relabel}) \)
unfolding \( \text{fifo-push-relabel-aux-def } \text{fifo-push-relabel-def} \)
unfolding \( \text{l-init-def } pp\text{-init-l-def }q\text{-is-empty-def bind-to-let-conv} \)
apply (rule specify-left[OF init-C-correct[OF AM]])
apply (refine-reg q-init-correct[OF AM])
apply refine-dref-type

7.2.5 Refining the Main Algorithm

definition \( \text{fifo-push-relabel-aux } am \equiv \{ \)
\( \text{cardV }\leftarrow \text{init-C } am; \)
\( \text{assert } (\text{cardV }= \text{card } V); \)
\( \text{let } f = \text{pp-init-f}; \)
\( l \leftarrow \text{l-init } \text{cardV}; \)
\( Q \leftarrow \text{q-init } am; \)
\( (f,l,-) \leftarrow \text{monadic-WHILEIT } (\lambda -. \text{True}) \)
\( (\lambda (f,l,Q). \text{do } \{ qe \leftarrow \text{q-is-empty } Q; \text{return } (\neg qe)\}) \)
\( (\lambda (f,l,Q). \text{do } \{ \)
\( \text{fifo-discharge } f \ l \ Q \)
\( \}) \)
\( (f,l,Q); \)
\( \text{assert } (\text{Height-Bounded-Labeling } c \ s \ t \ f \ l); \)
\( \text{return } f \)
\}

lemma \( \text{fifo-push-relabel-aux-refine}: \)
assumes \( AM: \text{is-adj-map } am \)
shows \( \text{fifo-push-relabel-aux } am \leq \Downarrow \text{Id } \) \((\text{fifo-push-relabel}) \)
unfolding \( \text{fifo-push-relabel-aux-def } \text{fifo-push-relabel-def} \)
unfolding \( \text{l-init-def } pp\text{-init-l-def }q\text{-is-empty-def bind-to-let-conv} \)
apply (rule specify-left[OF init-C-correct[OF AM]])
apply (refine-reg q-init-correct[OF AM])
apply refine-dref-type

99
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-rg)
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)
}) (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
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 ≡ λx (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 \iff \ q-\alpha Q = []
unfolding q-\alpha-def q-is-empty-impl-def
by (cases Q) (auto split: list.splits)

lemma q-dequeue-impl-correct-aux:
[\ q-\alpha Q = x#xs ] \iff \ apsnd q-\alpha (q-dequeue-impl Q) = (x, xs)
unfolding q-\alpha-def q-dequeue-impl-def
by (cases Q) (auto split: list.splits)

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

definition q-assn \equiv \ pure (br q-\alpha (\lambda - . True))

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

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

lemma q-enqueue-impl-hnr[sepref-fr-rules]:
(uncurry (return oo q-enqueue-impl), uncurry (PR-CONST q-enqueue))
\in \ nat-assn^{b} \times_{a} q-assn^{d} \rightarrow_{a} q-assn
apply (sepref-to-hoare)
unfolding q-assn-def q-enqueue-def pure-def
by (sep-auto simp: in-br-cone refine-pw-simps)

lemma q-dequeue-impl-hnr[sepref-fr-rules]:
(return o q-dequeue-impl, q-dequeue) \in \ q-assn^{d} \rightarrow_{a} nat-assn \times_{a} q-assn
apply (sepref-to-hoare)
unfolding q-assn-def q-dequeue-def pure-def prod-assn-def
by (sep-auto simp: in-br-cone refine-pw-simps split: prod.split)

103
7.4.3 Push

sepref-thm fifo-push-impl is uncurry3 (PR-CONST fifo-push2)
:: x-assn \* a cf-assn \* a q-assn \* a edge-assn
→ 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 \* a am-assn \* a cf-assn \* a clc-assn \* a q-assn \* a node-assn
→ 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,\_)∈-

7.4.5 Discharge

sepref-thm fifo-dis-loop-impl is uncurry5 (PR-CONST dis-loop2)
:: am-assn \* a x-assn \* a cf-assn \* a clc-assn \* a q-assn \* a node-assn
→ 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,\_)∈-

sepref-thm fifo-fifo-discharge-impl is uncurry5 (PR-CONST fifo-discharge2)
:: nat-assn \* a am-assn \* a x-assn \* a cf-assn \* a clc-assn \* a q-assn
→ 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,\_)∈-

7.4.6 Computing the Initial State

sepref-thm fifo-init-C-impl is (PR-CONST init-C)
:: am-assn \* a x-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 → 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 → a x-assn ×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-run)
:: am-assn k *a cf-assn d → 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, -)∈ -

sepref-thm fifo-push-relabel-init-impl
is uncurry0 (PR-CONST fifo-push-relabel-init)
:: unit-assn k → 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, -)∈ -

sepref-thm fifo-push-relabel-impl is (PR-CONST fifo-push-relabel2)
:: am-assn k → a cf-assn
unfolding fifo-push-relabel2-alts PR-CONST-def
by sepref
concrete-definition (in −) fifo-push-relabel-impl
uses Network-Impl.fifo-push-relabel-impl.refine-raw is (?f, -)∈ -
lemmas \([\text{sepref-fr-rules}] = \text{fifo-push-relabel-impl}.\text{refine}[\text{OF Network-Impl-axioms}]\)

end — Network Impl. Locale

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

7.5 Combining the Refinement Steps

theorem (in Network-Impl) \(\text{fifo-push-relabel-impl-correct}[\text{sep-heap-rules}]\):
assumes \(\text{AM: is-adj-map am}\)
shows
\(<\text{am-assn am ami}>\)
\(\text{fifo-push-relabel-impl } c s t N \text{ ami}\)
\(<\lambda cf. \exists A cf.\)
\(\text{am-assn am ami} * \text{cf-assn cf cf}\)
\(* \uparrow(\text{isMaxFlow (flow-of-cf cf)} \land \text{RGraph-Impl c s t N cf})>_{1}\)

proof —

note \(\text{fifo-push-relabel2-refine}[\text{OF AM}]\)
also note \(\text{fifo-push-relabel-correct}\)
finally have \(R1: \)
\(\text{fifo-push-relabel2 am} \leq \downarrow (\text{br flow-of-cf (RPreGraph c s t)}) (\text{SPEC isMaxFlow}) .\)

have [simp]: nofail \((\downarrow R (\text{RES X}))\) for \(R X\) by (auto simp: refine-pw-simps)

note \(R2 = \text{fifo-push-relabel-impl}.\text{refine}[\)
\(\text{OF Network-Impl-axioms, to-hnr, unfolded autoref-tag-defs}\]

note \(R3 = \text{hn-refine-ref}[\text{OF R1 R2, of ami}\]

note \(R4 = R3[\text{unfolded hn-ctzt-def pure-def, THEN hn-refineD, simplified}]\)

note \(RGII = \text{rgraph-and-network-impl-imp-rgraph-impl}[\text{OF}
\text{RPreGraph, maxflow-imp-rgraph}
\text{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 \(\text{fifo-push-relabel-impl-tab-am } c s t N \text{ am} \equiv \text{do }\{
ami \leftarrow \text{Array.make } N \text{ am}; \quad \text{— TODO/DUP: Called init-ps in Edmonds-Karp}
impl\)
\[ \text{fifo-push-relabel-impl } c s t N \ami; \]
\[ \text{return } (\ami, \text{cfi}) \]

**Theorem** \(\text{fifo-push-relabel-impl-tab-am-correct}[\text{sep-heap-rules}]:\)

**Assumes** \(\text{NW: Network } c s t\)

**Assumes** \(\text{VN: Graph. } V c \subseteq \{0..<N\}\)

**Assumes** \(\text{ABS-PS: Graph.is-adj-map } c \ami\)

**Shows**

\[ \text{fifo-push-relabel-impl-tab-am } c s t N \ami \]
\[ \lambda(\ami, \text{cfi}) \exists A \cf: \]
\[ \text{am-assn } N \ami \ami \ast \text{cf-assn } N \cf \text{cfi} \]
\[ \ast \downarrow (\text{Network.isMaxFlow } c s t \text{(Network.flow-of-cf } c \cf) \]
\[ \land \text{RGraph-Impl } c s t N \cf \]
\[ \rangle t \]

**Proof**

- **Interpret** \(\text{Network } c s t \) **by** **fact**
- **Interpret** \(\text{Network-Impl } c s t N \) **using** \(\text{VN} \) **by** **unfold-locales**

**From** \(\text{ABS-PS have} \) **simp**: \( \ami u = [] \) **if** \( u \geq N \) **for** \( u \)

**Unfolding** \(\text{is-adj-map-def}\)

**Using** \(\text{E-ss-VxV VN} \) **that**

**Apply** (**subgoal-tac** \( u \in V \))

**By** (**auto simp del**: \( \text{inV-less-N} \))

**Show** ?thesis

**Unfolding** \(\text{fifo-push-relabel-impl-tab-am-def}\)

**Apply** \(\text{vcg}\)

**Apply** (**rule Hoare-Triple.cons-rule[**

\[ \text{OF } \text{- fifo-push-relabel-impl-correct}[\text{OF } \text{ABS-PS}] \]

**Subgoal Unfolding** \(\text{am-assn-def is-nf-def} \) **by** **sep-auto**

**Apply** (**rule ent-refl**)  
**Subgoal by** **sep-auto**

**done**

**Qed**

**Definition** \(\text{fifo-push-relabel } el s t \equiv \) **do**

**Case** \(\text{prepareNet } el s t \) **of**

**None** \( \Rightarrow \) **return** **None**

**| Some (c,am,N) \Rightarrow do**

\( (\ami, \cf) \leftarrow \text{fifo-push-relabel-impl-tab-am } c s t N \ami; \)

**return** \( (\text{Some } (c,\ami,N,\cf)) \)

**Export-code** \(\text{fifo-push-relabel checking SML-imp}\)

Main correctness statement:
• If \textit{fifo-push-relabel} returns \texttt{None}, the edge list was invalid or described an invalid network.

• If it returns \texttt{Some} \((c, am, N, cf)\), then the edge list is valid and describes a valid network. Moreover, \(cf\) 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 \(N\).

\textbf{Theorem} \textit{fifo-push-relabel-correct}:\\
\texttt{fifo-push-relabel} \(el \ s \ t\)\\
\texttt{<\lambda}\\
\begin{align*}
\texttt{None} & \Rightarrow \uparrow (\neg \text{ln-invar} \ el \lor \neg \text{Network} \ (\text{ln-\alpha} \ el) \ s \ t) \\
\texttt{Some} \ (c, am, N, cf) & \Rightarrow \\
& \uparrow (c = \text{ln-\alpha} \ el \land \text{ln-invar} \ el \land \text{Network} \ c \ s \ t) \\
& \ast (\exists A \ am cf. \ am-assn N am ami \ast cf-Assn N cf cfi \\
& \ast \uparrow (RGraph-Impl c s t N cf \land \text{Graph.is-adj-map} \ c am \\
& \land \text{Network.isMaxFlow} c s t (\text{Network.flow-of-cf} c cf)) \\
\end{align*}
\texttt{>} \\
\texttt{t}\\
\textbf{Unfolding} \textit{fifo-push-relabel-def}\\
\texttt{using} \textit{prepareNet-correct[of el s t]}\\
\texttt{by} \texttt{(sep-auto simp: ln-rel-def in-br-cone)}

\textbf{7.6.1 Justification of Splitting into Prepare and Run Phase}

\textbf{Definition} \textit{fifo-push-relabel-prepare-impl} \(el \ s \ t\) \equiv \{\\
\texttt{case prepareNet} \ el \ s \ t \ of\\
\texttt{None} & \Rightarrow \texttt{return None} \\
\texttt{Some} \ (c, am, N) & \Rightarrow \{\\
& \texttt{ami} \leftarrow \text{Array.make} N am; \\
& \texttt{cfi} \leftarrow \text{fifo-push-relabel-init-impl} \ c \ N; \\
& \texttt{return} \ (\texttt{Some} \ (N, ami, c, cfi)) \\
\}\}

\textbf{Theorem} \textit{justify-fifo-push-relabel-prep-run-split}:
\texttt{fifo-push-relabel} \(el \ s \ t\) =
\texttt{do} \\
\texttt{pr} & \leftarrow \texttt{fifo-push-relabel-prepare-impl} \ el \ s \ t; \\
\texttt{case} \texttt{pr} \texttt{of}\\
\texttt{None} & \Rightarrow \texttt{return} \texttt{None} \\
\texttt{Some} \ (N, ami, c, cf) & \Rightarrow \{\\
& \texttt{cf} \leftarrow \texttt{fifo-push-relabel-run-impl} \ s \ t \ N \ ami \ cf; \\
& \texttt{return} \ (\texttt{Some} \ (c, ami, N, cf)) \\
\}\}
7.7 Usage Example: Computing Maxflow Value

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

**Definition**

```plaintext
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**

```plaintext
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
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 \( v \notin V \) for \( v \)

proof –
from that have adjacent-nodes \( v = {} \)
unfolding adjacent-nodes-def using \( E \text{-ss} \cdot V \times V \) by auto
thus \( \neg \text{thesis} \) using \( \text{am-to-adj-nodes-refine} \cdot \text{OF AM} \)
by (auto simp: list-set-rel-def in-br-conv)

qed

show \( \neg \text{thesis} \)
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 \cdot \text{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);
lv ← l-get l v;
if (cfuv ≠ 0 ∧ lv = 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: \( (ni,n) \in \text{nat-rel} \rightarrow \langle \text{nat-rel} \rangle \cdot \text{list-set-rel} \)
assumes SU: \( (ui,u) \in \text{Id} \)
assumes fNR: \( fN_i \leq \downarrow R fN \)
assumes UIV: \( u \in V \setminus \{s,t\} \)
assumes fSR: \( \forall v vi vs. [ \)
\[(vi, v) \in \text{Id}; \, v \in n \in u; \, \text{ni} \in u \Rightarrow (v \# vs, (v \# vs, n \in u)) \in \langle \text{nat-rel} \rangle \text{list-set-rel}\]

shows
\[
\begin{array}{l}
\begin{array}{l}
\text{(do } \{ \atend \leftarrow \text{n-at-end ni ui}; \\
\text{if } \atend \text{ then } fNi \\quad \text{else do } \{ \v \leftarrow \text{n-get-hd ni ui}; \\
\quad fSi v\}) \}) \leq \downarrow R (fS v)
\end{array}
\end{array}
\]

\[
\text{(do } \{ v \leftarrow \text{select v. } v \in n \in u; \\
\text{case } v \quad \text{of } \text{None } \Rightarrow fN \quad \text{| Some } v \Rightarrow fS v\}) (\text{is } ?\text{lhs } \leq \downarrow R ?\text{rhs})
\]

unfolding \text{n-at-end-def n-get-hd-def}

apply (\text{simp only: } \text{nres-monad-laws})

apply (\text{cases ni u})

subgoal

using \text{fun-relD}[\text{OF SR SU}] \text{ SU UIV fNR}

by (\text{auto simp: } \text{list-set-rel-def in-br-conv pw-le-iff refine-pw-simps})

subgoal for \text{v vs}

using \text{fun-relD}[\text{OF SR SU}] \text{ SU UIV}

using \text{FSR}[\text{OF IdI[of v], of vs}]

apply (\text{clarsimp simp: } \text{list-set-rel-def in-br-conv pw-le-iff refine-pw-simps split: option.splits})

by \text{fastforce}

done

lemma \text{xf-rel-RELATES}[\text{refine-dref-RELATES}]: \text{RELATES xf-rel}

by (\text{auto simp: } \text{RELATES-def})

lemma \text{discharge2-refine}[\text{refine}]:

assumes \text{A: } ((x, cf), f) \in xf-rel

assumes \text{AM: } (\text{am, adjacent-nodes}) \in \text{nat-rel} \rightarrow \langle \text{nat-rel} \rangle \text{list-set-rel}

assumes \text{[simplified simp]: } (li, l) \in \text{Id} \quad (ui, u) \in \text{Id}

assumes \text{NR: } (ni, n) \in \text{nat-rel} \rightarrow \langle \text{nat-rel} \rangle \text{list-set-rel}

shows \text{discharge2 am x cf li ni ui}

\leq \downarrow ((xf-rel \times_r \text{Id} \times_r (\text{nat-rel} \rightarrow \langle \text{nat-rel} \rangle \text{list-set-rel})) \text{ (discharge f l n u})

unfolding \text{discharge2-def discharge-def}

apply (\text{rewrite in monadic-WHILEIT - - \square - veg-intro-frame})

apply \text{refine-reg}

apply (\text{vc-solve simp: } \text{A NR})

subgoal by (\text{simp add: } \text{xf-rel-def x-get-def})
subgoal for /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)?)
apply (vc-solve simp: fun-relD fun-relD[OF AM])
subgoal for v vs
unfolding xf-rel-def Graph.E-def
by auto

8.2.2 Initialization of Queue

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

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

lemma init-CQ-correct[THEN order-trans, refine-vcg]:
assumes AM: is-adj-map am
shows init-CQ am â© SPEC (λ(C,Q). C = card V ∧ distinct Q ∧ set Q = V−{s,t})
unfolding init-CQ-def
apply (refine-vcg
nfoldl-rule[where
I=M1 - (C,Q),
C = card (V∩set I) ∧ distinct Q ∧ set Q = (V∩set I)−{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 â© do {

113
(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,u,L-left,L-right) ← whileT
(λ((x,cf),l,u,L-left,L-right). L-right ≠ [])
(λ((x,cf),l,u,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 {
    — Go to 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-front 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
\texttimes_r \text{Id} \\
\texttimes_r (\text{nat-rel} \rightarrow (\text{nat-rel})\text{list-set-rel}) \\
\texttimes_r \text{br rev} (\lambda \cdot \text{True}) \\
\texttimes_r \text{Id} \\

\textbf{have} \ [\text{refine-dref}-\text{RELATES}]: \ \text{RELATES} \ s \text{-rel} \ \text{unfolding} \ \text{RELATES-def} \ .

{ 

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

\textbf{have} \ AUXR: \ \text{do} \ { 

\begin{align*}
&\text{(cardV, L-right)} \leftarrow \text{init-CQ} \ am; \\
&\text{xcf} \leftarrow \text{pp-init-xcf2} \ am; \\
&l \leftarrow \text{l-init cardV}; \\
&n \leftarrow \text{n-init am}; \\
&\text{Fi L-right xcf l n} \\
&\} \\
&\leq \Downarrow R \ (\text{do} \ { 

\begin{align*}
&L-right \leftarrow \text{spec l. distinct l } \land \text{set l } = \text{V } - \{s, t\}; \\
&F \ L-right \\
&})
\end{align*}
\}) \\

\text{if} \ \bigwedge \text{L-right xcf n} . \\
\quad \begin{array} \{ (xcf, \text{pp-init-f}) \in \text{xcf-rel}; (n, \text{nrf-init-n}) \in \text{nat-rel} \rightarrow (\text{nat-rel})\text{list-set-rel} \end{array} \\
\implies \text{Fi L-right xcf pp-init-l n} \leq \Downarrow R \ (F \ L-right) \\
\text{for} \ \text{Fi} \ F \ R \\
\text{unfolding} \ l\text{-init-def} \\
\text{apply} \ \text{(rule refine2specI)} \\
\text{supply} \ \text{pp-init-xcf2-refine} \\
\quad \begin{array} \{ \text{OF AM, unfolded conc-fun-RETURN, THEN order-trans, refine-vcg} \end{array} \\
\text{supply} \ n\text{-init-refine}[\text{OF AM, THEN order-trans, refine-vcg}] \\
\text{apply} \ \text{(refine-vcg AM V-ss)} \\
\text{apply} \ \text{clarsimp} \\
\text{subgoal for} \ L-right x cf n \\
\quad \begin{array} \{ \text{of} \ (x, cf) \ n \ L-right \} \\
\text{unfolding} \ pp\text{-init-l-def} \\
\quad \text{by} \ \text{(clarsimp simp: pw-le-iff refine-pw-simps; meson)} \\
\text{done}
\end{array}
\text{show} \ \text{?thesis} \\
\text{unfolding} \ \text{relabel-to-front2-def relabel-to-front-def Let-def l-get-def} \\
\text{apply} \ \text{(simp only: ares-monad-laws)} \\
\text{apply} \ \text{(rule AUXR)} \\
\text{apply} \ \text{(refine-vcg)}
\end{align*}

115
apply refine-dref-type
unfolding s-rel-def
apply (vc-solve
  simp: in-br-conv rev-append-eq zf-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-assn⁺ →ₐ n-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))
∈ n-assn⁺ *ₐ node-assn⁺ →ₐ bool-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 \( n \ u \equiv \) do \\
\( \nu \leftarrow \text{Array}\ .\ \text{nth}\ n\ u; \) \\
\( \text{return} (\text{hd} \ \nu) \) \\
}\n
lemma [sepref-fr-rules]: \\
\((\text{uncurry} \ n\text{-get-hd-impl}, \text{uncurry} (\text{PR-CONST} \ n\text{-get-hd}))\) \\
\( \in \text{n-assn}^k \ *_a \ \text{node-assn}^k \rightarrow_a \ \text{node-assn} \) \\
apply \text{sepref-to-hoare} unfolding \( n\text{-get-hd-impl-def} \ n\text{-get-hd-def} \ n\text{-assn-def} \) \\
by (sep-auto simp: refine-pw-simps)

definition (in −) \( n\)-move-next-impl \( n \ u \equiv \) do \\
\( \nu \leftarrow \text{Array}\ .\ \text{nth}\ n\ u; \) \\
\( n \leftarrow \text{Array}\ .\ \text{upd}\ u \ (\text{tl} \ \nu) \ n; \) \\
\( \text{return} n \) \\
}\n
lemma [sepref-fr-rules]: \\
\((\text{uncurry} \ n\text{-move-next-impl}, \text{uncurry} (\text{PR-CONST} \ n\text{-move-next}))\) \\
\( \in \text{n-assn}^d \ *_a \ \text{node-assn}^k \rightarrow_a \ \text{n-assn} \) \\
apply \text{sepref-to-hoare} unfolding \( n\text{-move-next-impl-def} \ n\text{-move-next-def} \ n\text{-assn-def} \) \\
by (sep-auto simp: refine-pw-simps)

definition (in −) \( n\)-reset-impl \( a m \ n \ u \equiv \) do \\
\( \nu \leftarrow \text{Array}\ .\ \text{nth}\ a m\ u; \) \\
\( n \leftarrow \text{Array}\ .\ \text{upd}\ u \ \nu\ n; \) \\
\( \text{return} n \) \\
}\n
lemma [sepref-fr-rules]: \\
\((\text{uncurry2} \ n\text{-reset-impl}, \text{uncurry2} (\text{PR-CONST} \ n\text{-reset}))\) \\
\( \in \text{am-assn}^k \ *_a \ \text{node-assn}^k \ ightarrow_a \ \text{am-assn} \) \\
apply \text{sepref-to-hoare} unfolding \( n\text{-reset-impl-def} \ n\text{-reset-def} \ am\text{-assn-def} \) \\
by (sep-auto simp: refine-pw-simps)

8.3.2 Discharge

sepref-thm discharge-impl is \( \text{uncurry5} \ (\text{PR-CONST discharge2}) \) \\
:: \( \text{am-assn}^k \ *_a \ \text{x-assn}^d \ *_a \ \text{cf-assn}^d \ *_a \ \text{l-assn}^d \ *_a \ \text{n-assn}^d \ *_a \ \text{node-assn}^k \) \\
\rightarrow_a (\text{x-assn} \times_a \ \text{cf-assn}) \times_a \ \text{l-assn} \times_a \ \text{n-assn} \\
unfolding \( \text{discharge2-def} \ \text{PR-CONST-def} \) \\
by sepref

concrete-definition (in −) discharge-impl \\
uses Network-Impl.discharge-impl.refine-raw is (uncurry5 \ ?f,\cdot\cdot)\in\cdot\cdot\cdot\cdot\\
lemmas [sepref-fr-rules] = discharge-impl.refine\{OF Network-Impl-axioms\}

8.3.3 Initialization of Queue

sepref-thm init-CQ-impl is \( \text{PR-CONST} \ \text{init-CQ} \) \\
:: \( \text{am-assn}^k \rightarrow_a \ \text{nat-assn} \times_a \ \text{list-assn} \ \text{nat-assn} \) \\
unfolding \( \text{init-CQ-def} \ \text{PR-CONST-def} \)
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-assn ↦ a cf-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, -) ∈-

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
<cfi. ∃ A cf. cf-assn cf efi
  * (isMaxFlow (flow-of-cf cf) ∧ RGraph-Impl c s t N cf)>t
proof —
ote 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
end

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 <λ cfí. ∃ A cfí.
       asmtx-assn N id-assn cfí
      * ∨ (Network.isMaxFlow c s t (Network.flow-of-cf c cfí)
           ∧ RGraph-Impl c s t N cfí)
    )≥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 vug
    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 \((\text{Some} \ (c,am,N,cf))\) |
|} |

**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:**

\(<\text{emp}>\)

\(\text{relabel-to-front} el\ s\ t\)

\(<\lambda \)

\(\text{None} \Rightarrow ↑(\neg \text{ln-invar} el \lor \neg \text{Network} \ (\text{ln-}α\ el) s t)\)

| Some \((c,am,N,cfi)\) ⇒ |
| ↑\((c = \text{ln-}α\ el \land \text{ln-invar} el)\) |
| (∃_{A cf}. asmtx-assn N int-assn cf cfi |
| * ↑\((\text{RGraph-Impl} c\ s\ t\ N\ cf)\) |
| * ↑\((\text{Network.isMaxFlow} c\ s\ t\ (\text{Network.flow-of-cf} c\ cf)))\) |
| * ↑\((\text{Graph.is-adj-map} c\ am)\) |

\(> t\)

**unfolding relabel-to-front-def**

**using prepareNet-correct [of el s t]**

**by** \((\text{sep-auto simp: ln-rel-def in-br-conv})\)

**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


