Formalizing Push-Relabel Algorithms

Peter Lammich and S. Reza Sefidgar

April 20, 2020

Abstract

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

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

7.7 Usage Example: Computing Maxflow Value

8 Implementation of Relabel-to-Front

8.1 Basic Operations

8.1.1 Neighbor Lists

8.2 Refinement to Basic Operations

8.2.1 Discharge

8.2.2 Initialization of Queue

8.2.3 Main Algorithm

8.3 Refinement to Efficient Data Structures

8.3.1 Neighbor Lists by Array of Lists

8.3.2 Discharge

8.3.3 Initialization of Queue

8.3.4 Main Algorithm

8.4 Combination with Network Checker and Correctness

9 Conclusion
1 Introduction

Computing the maximum flow of a network is an important problem in graph theory. Many other problems, like maximum-bipartite-matching, edge-disjoint-paths, circulation-demand, as well as various scheduling and resource allocating problems can be reduced to it.

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

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

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

2 Generic Push Relabel Algorithm

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

2.1 Labeling

The central idea of the push-relabel algorithm is to add natural number labels $l : \text{node} \Rightarrow \text{nat}$ 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: $(u,v) \in E \Rightarrow l(u) \leq l(v) + 1$
  assumes lab-src [simp]: $l(s) = \text{card } V$
  assumes lab-sink [simp]: $l(t) = 0$
begin

Generalizing validity to paths

lemma gen-valid: $l(u) \leq l(x) + \text{length } p$ if $\text{cf.isPath } u p x$
  using that by (induction $p$ arbitrary: $u$; fastforce dest: valid)
In a valid labeling, there cannot be an augmenting path [Cormen 26.17]. The proof works by contradiction, using the validity constraint to show that any augmenting path would be too long for a simple path.

**Theorem no-augmenting-path:** \( \neg \text{isAugmentingPath } p \)

**Proof**

Assume \( \text{isAugmentingPath } p \)

Hence \( \text{SP: } \text{cf.isSimplePath } s \ p \ t \) unfolding \( \text{isAugmentingPath-def} \).

Hence \( \text{cf.isPath } s \ p \ t \) unfolding \( \text{cf.isSimplePath-def} \) by auto

From \( \text{gen-valid[OF this]} \) have \( \text{length } p \geq \text{card } V \) by auto

With \( \text{cf.simplePath-length-less-}\ V[\text{OF - SP}] \) show \( \text{False} \) 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 \( \text{isMaxFlow } f \)

**Proof**

From \( \text{assms interpret NFlow} \)

Apply \( \text{unfold-locales} \)

Using \( \text{no-deficient-nodes unfolding excess-def} \) by auto

From \( \text{noAugPath-iff-maxFlow no-augmenting-path show isMaxFlow } f \) by auto

qed

---

**2.2 Basic Operations**

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

**2.2.1 Augmentation of Edges**

**Context** Network

**Begin**

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

**Definition augment-edge :: 'capacity flow ⇒ -**

Where \( \text{augment-edge } f \equiv \lambda (u,v) \Delta. \)

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]**: \( \text{augment-edge } f 0 = f \)

**Unfolding augment-edge-def** by (auto split: prod.split)
The effect of augmenting an edge on the residual graph
definition (in -) augment-edge-cf :: - flow ⇒ - where
augment-edge-cf cf
≡ \( \lambda (u,v) \Delta. \) (cf)((u,v) := cf (u,v) − Δ, (v,u) := cf (v,u) + Δ)

lemma cf-of-augment-edge:
assumes A: \((u,v) \in \text{cfE-of } f\)
shows cf-of (augment-edge f (u,v) Δ) = augment-edge-cf (cf-of f) (u,v) Δ
proof –
show cf-of (augment-edge f (u,v) Δ) = augment-edge-cf (cf-of f) (u,v) Δ
  by (simp add: augment-edge-cf-def A augment-edge-cf′)
qed

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

end — Network

context NPreflow begin

Augmenting an edge \((u,v)\) with a flow \(\Delta\) 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: \([ \theta \leq \Delta; \Delta \leq cf (u,v); \Delta \leq excess f u ]\) ⇒ Preflow c s t (augment-edge f (u,v) Δ)
apply unfold-locales
subgoal
  unfolding residualGraph-def augment-edge-def
  using capacity-const
  by (fastforce split!: if-splits)
subgoal
proof (intro ballI; clarsimp)
assume \(\theta \leq \Delta\) \(\Delta \leq cf (u,v)\) \(\Delta \leq excess f u\)
fix \(v′\)
assume \(V′: v′ \in V\) \(v′ \neq s\) \(v′ \neq t\)
show \(\text{sum} (\text{augment-edge } f (u,v) \Delta) (\text{outgoing } v′) \leq \text{sum} (\text{augment-edge } f (u,v) \Delta) (\text{incoming } v′)\)
proof (cases)
assume \(\Delta = 0\)
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 $\Delta \leq \text{cf} \ (u,v)$ have $(u,v) \in \text{cf}.E$ unfolding Graph.E-def by auto

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

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

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

have 2: $\forall e \in \text{incoming } v'$. 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 $\exists$thesis by auto
} moreover { assume [simp]: $v'=v$
have sum (augment-edge $f$ $(u,v)$ $\Delta$) (outgoing $v'$)
  $=$ sum $f$ (outgoing $v$)
using 1 $(u \neq v)$ $V'$ by auto
also have ... $\leq$ sum $f$ (incoming $v$)
using $V'$ no-deficient-nodes by auto
also have ... $\leq$ 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-def split!: if-split)
finally have $\exists$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 $(u,v) \in \text{outgoing } u$ using $(u,v) \in \text{E}$
by (auto simp: outgoing-def)
note $\text{AUX} = \text{sum.remove[OF - this, simplified]}$

have $A2$: sum (augment-edge $f$ $(u,v)$ $\Delta$) (outgoing $u$)
  $=$ sum $f$ (outgoing $u$) + $\Delta$
using $\text{AUX[of augment-edge f (u,v) \Delta]} \ \text{AUX[of \ f]}$ by auto
from $A1 \ A2$ $(\Delta \leq \text{excess } f w)$ no-deficient-nodes $V'$ have $\exists$thesis
unfolding excess-def by auto
}
ultimately show $\exists$thesis by blast

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

hence [simp]: $(v,u) \in E$ using $\text{cfE-ss-invE} \ (u,v) \in \text{cf}.E$ by auto
from $(u,v) \notin E$ $(v,u) \in E$ have $u \neq v$ by blast

9
have \(AE\): augment-edge \(f (u,v) \Delta = f (v,u) - \Delta\)

unfolding augment-edge-def by simp

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

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

\{
assume \(v' \neq u\) \(v' \neq v\)
with 1 \(2\) \(V'\) no-deficient-nodes have \(\text{thesis}\) by auto
\}

moreover \{
assume \([simp]: v'=u\)
have A1: \(\text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{outgoing } v')\)
\(= \text{sum } f (\text{outgoing } u)\)
using 1 \(\langle u \neq v\rangle\) \(V'\) by auto

have \((v,u) \in \text{incoming } u\)
using \((v,u) \in E\) by (auto simp: incoming-def)
note AUX = sum.remove[OF - this, simplified]

have A2: \(\text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{incoming } u)\)
\(= \text{sum } f (\text{incoming } u) - \Delta\)
using AUX\([\text{of augment-edge } f (u,v) \Delta]\) AUX\([\text{of } f]\) by auto

from A1 A2 \(\Delta \leq \text{excess } f \subseteq \text{no-deficient-nodes } V'\) have \(\text{thesis}\)
unfolding excess-def by auto
\}

moreover \{
assume \([simp]: v'=v\)
have \(\text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{outgoing } v')\)
\(\leq \text{sum } f (\text{outgoing } v')\)
apply (rule sum-mono)
using \(0 < \Delta\)
by (auto simp: augment-edge-def)
also have \(\ldots \leq \text{sum } f (\text{incoming } v)\)
using no-deficient-nodes \(V'\) by auto
also have \(\ldots \leq \text{sum } (\text{augment-edge } f (u,v) \Delta) (\text{incoming } v')\)
using 2 \(\langle u \neq v\rangle\) by auto
finally have \(\text{thesis}\) by simp
\}
ultimately show \(\text{thesis}\) by blast
qed
qed
done
end — Network with Preflow

### 2.2.2 Push Operation

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

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

**definition** push-precond :: 'capacity flow ⇒ labeling ⇒ edge ⇒ bool

**where** push-precond f l

\[\equiv \lambda(u,v). \text{excess } f \ u > 0 \land (u,v) \in cfE-of f \land l_u = l_v + 1\]

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

\[\equiv \lambda(u,v). \text{augment-edge } f \ (u,v) \ (\text{min } \text{excess } f \ u \ (\text{cf-of } f \ (u,v)))\]

**lemma** push-precondI

\[\begin{align*}
\text{[} \text{excess } f \ u > 0; (u,v) \in cfE-of f; l_u = l_v + 1\text{]} & \implies \text{push-precond } f \ l \ (u,v) \\
\text{unfolding push-precond-def by auto}
\end{align*}\]

### 2.2.3 Relabel Operation

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

**definition** relabel-precond :: 'capacity flow ⇒ labeling ⇒ node ⇒ bool

**where** relabel-precond f l u

\[\equiv u \neq t \land \text{excess } f \ u > 0 \land (\forall v. (u,v) \in cfE-of f \rightarrow l_u \neq l_v + 1)\]

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

\[\equiv l(u := \text{Min} \ \{ \ l(v) \ | \ v. (u,v) \in cfE-of f \ \} + 1)\]

### 2.2.4 Initialization

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

**definition** pp-init-f \[\equiv \lambda(u,v). \text{if } (u=s) \text{ then } c \ (u,v) \text{ else } 0\]

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-locales;
  ((auto simp: pp-init-f-def pp-init-l-def cap-non-negative; fail)
    | (intro ballI)?)

**proof** –

fix v

assume v∈V − {s,t}

**hence** ∀ e∉outgoing v. pp-init-f e = 0

by (auto simp: outgoing-def pp-init-f-def)

**hence** [simp]: sum pp-init-f (outgoing v) = 0 by auto

**have** 0 ≤ pp-init-f e for e

by (auto simp: pp-init-f-def cap-non-negative split: prod.split)

**from** sum-bounded-below[of incoming v 0 pp-init-f, OF this]

**have** 0 ≤ sum pp-init-f (incoming v) by auto

**thus** sum pp-init-f (outgoing v) ≤ sum pp-init-f (incoming v)

by auto

next

fix u v

assume (u, v) ∈ Graph.E (residualGraph c pp-init-f)

**thus** pp-init-l u ≤ pp-init-l v + 1

**unfolding** pp-init-l-def Graph.E-def pp-init-f-def residualGraph-def

by (auto split: if-splits)

qed

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

**proof** –

from pp-init-invar interpret Labeling c s t pp-init-f pp-init-l .

**show** ?thesis by unfold-locales

qed

end — Network

**context** Labeling

**begin**

Push operations preserve a valid labeling [Cormen 26.16].
**Theorem** push-pres-Labeling:
assumes push-precond \( f \) \( l \) \( e \)
shows Labeling \( c \) \( s \) \( t \) (push-effect \( f \) \( e \)) \( l \)

**Proof** (cases \( e \); clarsimp)

\( \text{fix } a \ b \) 
assume \[
\text{simp: } e = (a, b)
\]
let \( \bar{f}' = (\text{augment-edge } f \ (a, b) \ (\min \ (\text{excess } f \ u) \ (\text{cf } (a, b)))) \)

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

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

qed

**Lemma** finite-min-cf-outgoing [simp, intro!]: finite \( \{ l \ v | v. \ (u, v) \in \text{cf}.E \} \)

**Proof** –

have \( \{ l \ v | v. \ (u, v) \in \text{cf}.E \} = \text{snd}' \text{cf.outgoing } u \)
by (auto simp: cf.outgoing-def)
moreover have finite (\text{snd}' cf.outgoing \( u \)) by auto
ultimately show \(?thesis by auto

qed

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

**Theorem**
assumes PRE: relabel-precond \( f \) \( l \) \( u \)
shows relabel-increase-u: relabel-effect \( f \) \( l \) \( u \) \( u \) \( l \) u \( (\text{is} \ ?G1) \)
and relabel-pres-Labeling: Labeling \( c \) \( s \) \( t \) \( f \) (relabel-effect \( f \) \( l \) \( u \)) \( (\text{is} \ ?G2) \)

**Proof** –

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

from ACTIVE have [simp]: \( s \neq u \) using excess-s-non-pos by auto

13
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: \( l_u \leq \min \{ l_v | v. (u,v) \in cf.E \} \)
by (auto simp: less-Suc-eq-le)

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

moreover have \( \forall (u,v) \in cf.E \rightarrow \min \{ l_v | v. (u,v) \in cf.E \} + 1 \leq l_v + 1 \)
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)
apply (unfold-locales)

subgoal using \( u' \) \( v' \) using valid by auto

subgoal by auto

subgoal using NOT-SINK by auto

done

qed

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
with no-excess-imp-maxflow show \( ?\text{thesis} \).

qed

eyd — Labeling

2.4 Convenience Lemmas

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

locale push-effect-locale = Labeling +
fixes u v
assumes PRE: push-precond f l (u,v)

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

lemma uv-cf-edge[simp, intro!]: (u,v)∈ cf.E
  using PRE unfolding push-precond-def by auto

lemma excess-u-pos; excess f u > 0
  using PRE unfolding push-precond-def by auto

lemma l-u-eq[simp]: l u = l v + 1
  using PRE unfolding push-precond-def by auto

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

lemma uv-nodes[simp, intro!]: u∈ V v∈ V
  using E-ss-VxV cfE-ss-invE by auto

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

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

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

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

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

lemma excess'-u[simp]: excess f' u = excess f u − Δ
  unfolding excess-def[where f=f']

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

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

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

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

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

finally show ?thesis using 1 by (auto simp: excess-def)

next

  case [simp]: rev

  have UV-INO:(v,u)∈outgoing v – incoming v
    by (auto simp: incoming-def outgoing-def no-self-loop)

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

  have sum f' (outgoing v)
    = sum f (outgoing v) + (∑ x∈outgoing v. if x=(v,u) then − ∆ else 0)
  unfolding f'--alt augment-edge-def sum.distrib[symmetric]
  apply (rule sum.cong)
  using UV-INO unfolding f'--alt by auto

  also have ... = sum f (outgoing v) − ∆
    using UV-INO by (auto simp: sum.delta)

  finally show ?thesis using 1 by (auto simp: excess-def)

qed

qed

lemma excess'--other[simp]:
  assumes x ≠ u  x ≠ v
  shows excess f' x = excess f x

proof –

  have NE: (u,v)∉incoming x  (u,v)∉outgoing x
           (v,u)∉incoming x  (v,u)∉outgoing x
    using assms unfolding incoming-def outgoing-def by auto

  have sum f' (outgoing x) = sum f (outgoing x)
  sum f' (incoming x) = sum f (incoming x)
    by (auto
      simp: augment-edge-def f'--alt NE
      split!: if-split
      intro: sum.cong)

  thus ?thesis
  unfolding excess-def by auto

qed

lemma excess'--if:
  excess f' x = ( 
    if x=u then excess f u − ∆
    else if x=v then excess f v + ∆
    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:
context Height-Bounded-Labeling
begin

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

lemma push-pres-height-bound:
  assumes push-precond f l e
  shows Height-Bounded-Labeling c s t (push-effect f e) l
proof
  from push-pres-Labeling[OF assms]
  interpret l': Labeling c s t push-effect f e .
  show ?thesis using height-bound by unfold-locales
qed

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

lemma (in Labeling) excess-imp-source-path:
  assumes excess f u > 0
  obtains p where cf.isSimplePath u p s
proof
  obtain U where U-def: U = {v|p v. cf.isSimplePath u p v} by blast
  have fct1: U ⊆ V
  proof
    fix v
    assume v ∈ U
    then have (u, v) ∈ cf.E*
    using U-def cf.isSimplePath-def cf.isPath-rtc by auto
    then obtain u' where u = v ∨ ((u, u') ∈ cf.E* ∧ (u', v) ∈ cf.E)
    by (meson rtranclE)
    thus v ∈ V
  proof
    assume u = v
    thus ?thesis using excess-nodes-only[OF assms] by blast
  next
    assume (u, u') ∈ cf.E* ∧ (u', v) ∈ cf.E
    then have v ∈ cf.V unfolding cf.V-def by blast
    thus ?thesis by simp
  qed
  qed

have s ∈ U
proof(rule ccontr)
  assume s /∈ U
  obtain U' where U'-def: U' = V - U by blast
have \( \sum_{u \in U} \text{excess } f u \)
= \( \sum_{u \in U} \left( \sum_{v \in U'} f (v, u) \right) - \left( \sum_{u \in U} \left( \sum_{v \in U'} f (u, v) \right) \right) \)

\begin{align*}
\text{proof} -
\quad & \text{have } \left( \sum_{u \in U} \text{excess } f u \right)
= \left( \sum_{u \in U} \left( \sum_{v \in u} \text{incoming } f v \right) \right) - \left( \sum_{u \in U} \left( \sum_{v \in u} \text{outgoing } f v \right) \right)
\quad \text{(is - = ?R1 - ?R2) unfolding excess-def by (simp add: sum-subtractf)}
\quad \text{also have } ?R1 = \left( \sum_{u \in U} \left( \sum_{v \in V} f (v, u) \right) \right)
\quad \text{using sum-outgoing-alt-flow fct1 by (meson subsetCE sum.cong)}
\quad \text{also have } \ldots = \left( \sum_{u \in U} \left( \sum_{v \in U} f (v, u) \right) \right) + \left( \sum_{u \in U} \left( \sum_{v \in U'} f (v, u) \right) \right)
\end{align*}

\begin{align*}
\quad & \text{proof} -
\quad \text{have } \left( \sum_{v \in V} f (v, u) \right)
= \left( \sum_{v \in U} f (v, u) \right) + \left( \sum_{v \in U'} f (v, u) \right) \text{ for } u
\quad \text{using } U'-\text{def fct1 finite-V}
\quad \text{by (metis ab-semgroup-add-class.add.commute sum.subset-diff)}
\quad \text{thus } ?\text{thesis by (simp add: sum.distrib)}
\end{align*}

\text{qed}

\text{also have } ?R2 = \left( \sum_{u \in U} \left( \sum_{v \in V} f (u, v) \right) \right)
\text{using sum-outgoing-alt-flow fct1 by (meson subsetCE sum.cong)}
\text{also have } \ldots = \left( \sum_{u \in U} \left( \sum_{v \in U} f (u, v) \right) \right) + \left( \sum_{u \in U} \left( \sum_{v \in U'} f (u, v) \right) \right)

\begin{align*}
\quad & \text{proof} -
\quad \text{have } \left( \sum_{v \in V} f (u, v) \right)
= \left( \sum_{v \in U} f (u, v) \right) + \left( \sum_{v \in U'} f (u, v) \right) \text{ for } u
\quad \text{using } U'-\text{def fct1 finite-V}
\quad \text{by (metis ab-semgroup-add-class.add.commute sum.subset-diff)}
\quad \text{thus } ?\text{thesis by (simp add: sum.distrib)}
\end{align*}

\text{qed}

\text{also have } \left( \sum_{u \in U} \left( \sum_{v \in U} f (u, v) \right) \right) = \left( \sum_{u \in U} \left( \sum_{v \in U} f (v, u) \right) \right)
\text{proof -}

\begin{align*}
& \{ \\
\text{fix } A :: \text{nat set} \\
\text{assume finite } A \\
\text{then have } \left( \sum_{u \in A} \left( \sum_{v \in A} f (u, v) \right) \right) = \left( \sum_{u \in A} \left( \sum_{v \in A} f (v, u) \right) \right)
\text{proof (induction card } A \text{ arbitrary: } A) \\
\text{case } 0 \\
\text{then show ?case by auto} \\
\text{next} \\
\text{case } (Suc x) \\
\text{then obtain } A' \\
\text{where } o1 : A = \text{insert } a \text{ A' and } o2 : x = \text{card } A' \text{ and } o3 : \text{finite } A' \\
\text{by (metis card-insert-disjoint card-le-Suc-iff le-refl nat.inject)} \\
\text{then have } \text{ln} : \left( \sum_{e \in A} (g e) \right) = \left( \sum_{e \in A'} (g e) \right) + g 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) sum.insert)} \\
\text{have } \left( \sum_{u \in A} \left( \sum_{v \in A} f (u, v) \right) \right) \\
= \left( \sum_{u \in A'} \left( \sum_{v \in A} f (u, v) \right) \right) + \left( \sum_{v \in A} f (a, v) \right)
\end{align*}
is - = ?R1 + ?R2) using ln by auto
also have ?R1 = (∑u∈A'. (∑v∈A'. f(u, v))) + (∑u∈A'. f(u, a))
(is - = ?R1-1 + ?R1-2) using ln sum.distrib by force
also note add.assoc
also have ?R1-2 + ?R2 = (∑u∈A'. f(a, u)) + (∑v∈A. f(v, a))
(is - = ?R1-2' + ?R2') using ln by auto
also have ?R1-1 = (∑u∈A'. (∑v∈A'. f(v, u)))
(is - = ?R1-1') using Suc.hyps(1)[of A'] o2 o3 by auto
also note add.assoc[symmetric]
also have ?R1-1' + ?R1-2' = (∑u∈A'. (∑v∈A. f(v, u)))
by (metis (no-types, lifting) ln sum.cong sum.distrib)
finally show ?thesis using ln[symmetric] by auto
qed

}\note this[of U]

thus ?thesis using fct1 finite-V finite-subset by auto
qed
finally show ?thesis by arith

qed

moreover have (∑u∈U. excess f u) > 0

proof –

have u ∈ U using U-def by simp
moreover have u ∈ U ⇒ excess f u ≥ 0 for u
  using fct1 excess-non-negative' (s ∉ U) by auto
ultimately show ?thesis using assms fct1 finite-V
  by (metis Diff-cancel Diff-eq-empty-iff
    Diff-infinite-finite finite-Diff sum-pos2)

qed

ultimately have

fct2: (∑u∈U. (∑v∈U'. f(v, u))) - (∑u∈U. (∑v∈U'. f(u, v))) > 0
by simp

have fct3: (∑u∈U. (∑v∈U'. f(v, u))) > 0

proof –

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

qed

have ∃u' v'. (u' ∈ U ∧ v' ∈ U' ∧ f(v', u') > 0)

proof (rule contr)
assume ¬(∃u' v'. u' ∈ U ∧ v' ∈ U' ∧ f(v', u') > 0)
than have (∀u' v'. (u' ∈ U ∧ v' ∈ U' ⇒ f(v', u') = 0))
  using capacity-const by (metis le-neq-trans)
thus False using fct3 by simp

qed

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

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

proof —
  have \( (v', u') \in E \)
    using capacity-const 'if \( (v', u') > 0 \)
    by (metis not-less zero-flow-simp)
  then have \( \text{cf} \ (u', v') > 0 \) unfolding \( \text{cf-def} \)
    using no-parallel-edge 'if \( f(v', u') > 0 \) 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.isPath-append-edge Graph.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 \( \text{U-def} \) by auto
thus \( \text{False} \) using \( (u' \in U' \text{ and } U'\-def) \) by simp

qed

Relabel operations preserve the height bound [Cormen 26.20].

lemma relabel-pres-height-bound: assumes relabel-precond \( f \ l \ u \)
shows \( \text{Height\-Bounded\-Labeling} \ c \ s \ t \ f \ (\text{relabel\-effect} \ f \ l \ u) \)

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 \( \text{excess} \ f \ u > 0 \) 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 \( \text{excess\-nodes\-only} \ (\text{excess} \ f \ u > 0) \).
then have \( \text{length} \ p < \text{card} \ V \)
  using \( \text{cf}. \text{simplePath\-length\-less\-V}[o f \ u \ p] \) \( \text{p\-obt} \) by auto
moreover have \( ?l' \ u \leq ?l' \ s + \text{length} \ p \)
  using \( \text{p\-obt} \ l'.\text{gen\-valid}[o f \ u \ p \ s] \) \( \text{p\-obt} \)
  unfolding \( \text{cf}. \text{isSimplePath\-def} \) by auto
moreover have \( ?l' \ s = \text{card} \ V \)
  using \( l'.\text{Labeling\-axioms Labeling\-def Labeling\-axioms\-def} \) by auto
ultimately have \( ?l' \ u \leq 2*\text{card} \ V - 1 \) by auto
thus \( \text{Height\-Bounded\-Labeling} \ c \ s \ t \ f \ ?l' \)
  apply unfold-locales
  using height-bound relabel\-preserve\-other
  by metis

qed
Thus, the total number of relabel operations is bounded by $O(V^2)$ [Cormen 26.21]. We express this bound by defining a measure function, and show that it is decreased by relabel operations.

definition (in Network) sum-heights-measure $l \equiv \sum_{v \in V} 2 \cdot \text{card } V - l v$

corollary relabel-measure:
  assumes relabel-precond $f \mid l \mid u$
  shows sum-heights-measure (relabel-effect $f \mid l \mid u$) $< \text{sum-heights-measure } l$

proof
  let $?l' = \text{relabel-effect } f \mid l \mid u$
  from relabel-pres-height-bound[of assms]
  interpret $l': \text{Height-Bounded-Labeling } c \mid s \mid t \mid f \mid ?l'$.

  from assms have $u \in V$
    by (simp add: excess-nodes-only relabel-precond-def)
  hence V-split: $V = \text{insert } u \mid V$ by auto

  show $?thesis
    using relabel-increase-u[of assms] relabel-preserve-other[of $u$]
    using $?l'.height-bound
    unfolding sum-heights-measure-def
    apply (rewrite at $\sum \cdot \in \Pi - \text{V-split}$)+
    apply (subst sum.insert-remove[of $\text{finite-V}$])+
    using $\langle u \in V \rangle$
    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 $(\sum v \in V. 2 \cdot \text{card } V - l v) \leq (\sum v \in V. 2 \cdot \text{card } V)$
    by (meson sum-mono)
  also have $(\sum v \in V. 2 \cdot \text{card } V) = \text{card } V \cdot (2 \cdot \text{card } V)$
    using finite-V by auto
  finally show $(\sum v \in V. 2 \cdot \text{card } V - l v) \leq 2 \cdot (\text{card } V)^2$
    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 \( \exists e \). push-precond f l e and \( \exists u \). 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)
\((u,v) \in E \land l_u = l_v + 1\)

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

\[\text{push-precond } f \triangleq \text{sat-push-precond } f \lor \text{nonsat-push-precond } f\]

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

**By** auto

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

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

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

**By** auto

**Lemma** sat-push-alt: sat-push-precond 
\[f \triangleq \text{push-effect } f = \text{augment-edge } f \text{ (cf-off } f)\]

**Unfolding** push-effect-def push-precond-eq-or-nonsat sat-push-precond-def

**By** (auto simp: min-absorb2)

**Lemma** nonsat-push-alt: nonsat-push-precond 
\[f \triangleq \text{push-effect } f (u,v) = \text{augment-edge } f (u,v) \text{ (excess } f)\]

**Unfolding** push-effect-def push-precond-eq-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 
\[f \triangleq \Delta = \text{excess } f\]

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

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

**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' =

\[\text{is-RELABEL'}: \text{RELABEL'}\]
\[| \text{is-NONSAT-PUSH'}: \text{NONSAT-PUSH'}\]
\[| \text{is-SAT-PUSH'}: \text{SAT-PUSH'}\]

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

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

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

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

end — Network

context Height-Bounded-Labeling

begin

lemma (in Height-Bounded-Labeling) xfer-run:
assumes ((f,l),p,(f',l')) ∈ trcl pr-algo-lts
obtains p' where ((f,l),p,(f',l')) ∈ trcl pr-algo-lts'
and p = map project-operation p'
proof —
have ∃p'.
  Height-Bounded-Labeling c s t f' l'
  ∧ ((f,l),p,(f',l')) ∈ trcl pr-algo-lts'
  ∧ p = map project-operation p'
  using assms
proof (induction p arbitrary: f' l' rule: rev-induct)
case Nil thus ?case using Height-Bounded-Labeling-axioms by simp
next
case (snoc a p)
  from snoc.prems obtain fh lh
  where PP: ((f, l), p, fh, lh) ∈ trcl pr-algo-lts
  and LAST: ((fh,lh),a,(f',l'))∈pr-algo-lts
  by (auto dest!: trcl-rev-uncons)
  from snoc.IH[OF PP] obtain p'
  where HBL: Height-Bounded-Labeling c s t fh lh
  and PP': (f, l), p', fh, lh) ∈ trcl pr-algo-lts'
  and [simp]: p = map project-operation p'
  by blast

26
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 length (filter is-PUSH \( p \)) \\
= length (filter is-SAT-PUSH \( p' \)) + length (filter is-NONSAT-PUSH \( p' \)) \\
by simp
also note BOUND-SAT[rule-format,OF RUN]
also note BOUND-NONSAT[rule-format,OF RUN]
finally show \( \exists \)thesis by simp

end — Height Bounded Labeling

2.5.6 Bounding the Relabel Operations

lemma (in Network) relabel-action-bound:
assumes \( A: (fxl,p,fxl') \in trcl pr-algo-lts' \)
shows length (filter (is-RELABEL') \( p \)) \leq 2 * (\text{card } V)^2

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

qed

lemma (in Height-Bounded-Labeling) relabel-action-bound:
assumes \( A: ((f,l),p,(f',l')) \in trcl pr-algo-lts \)
shows length (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 \( ((f,l),p,(f',l')) \in trcl pr-algo-lts' \)
shows \( l u \leq l' u \)
using assms

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

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

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

lemma edge-reappears-at-increased-labeling:
  assumes ((f,l),p,(f',l')) ∈ trcl pr-algo-lts'
  assumes l u ≥ l v + 1
  assumes (u,v) ∉ cfE-of f
  assumes E': (u,v) ∈ cfE-of f'
  shows l v < l' v
  using assms(1-3)

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

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

consider
  (push) u' v'
  where push-precond f l (u',v') fh = push-effect f (u',v') lh=l
  | (relabel) u'
  where relabel-precond f l u' fh=f lh=relabel-effect f l u'
  using FIRST
  by (auto elim!: pr-algo-lts'.cases simp: push-precond-eq-sat-or-nonsat)

then show ?case proof cases
  case push
  note [simp] = push(2,3)

The push operation cannot go on edge (u,v) or (v,u)
from push(1) have \((u',v')\neq (u,v)\) \((u',v')\neq (v,u)\) \((u',v')\in cf.E\)
using \((l u \geq l v + 1)\) \(\langle (u,v)\notin cf.E\rangle\)
by \(\text{(auto simp: push-precond-def)}\)
hence \(NE': (u,v)\notin cfE-of fh\) using \(\langle (u,v)\notin cf.E\rangle\)
by \(\text{(auto simp: push-effect-def)}\)

next

case relabel

note \([\text{simp}]=\text{relabel}(2)\)

show \(?thesis\)
proof \(\text{(cases u'=v)}\)
  case False
  from False relabel \(3\) relabel-preserve-other have \([\text{simp}]\): \(lh v = l v\)
  by \(\text{auto}\)
  from False relabel \(3\)
  relabel-preserve-other relabel-increase-u \(\text{OF relabel}(1)\)
  have \(lh u \geq l u\) by \(\text{(cases u'=u)}\) \(\text{auto}\)
  with \(\langle l u \geq l v + 1\rangle\) \(\text{have LG}: lh u \geq lh v + 1\) \(\text{by} \ \text{auto}\)

from \(\text{Cons.IH}(OP PP \ - \ NE')\ \langle a \geq l v + 1\rangle\) \(\text{show} \ ?thesis \text{by simp}\)

next

case True

note \([\text{simp}]=\text{relabel}(3)\)

from True relabel-increase-u \(\text{OF relabel}(1)\)
have \(lh v < lh v\) by \(\text{simp}\)
also note labels-path-increasing \(\text{OF PP, of } v\)
finally show \(?thesis\) by \(\text{simp}\)

qed

qed

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

proof –

obtain \(u v\) where \([\text{simp}]: e=(u,v)\) by \(\text{(cases e)}\)

have \(\text{length} \ \text{(filter} \ (\text{=} \ \text{(SAT-PUSH'} (u,v))) \ p\) \(\leq 2*\text{card} \ V - l v\)
if \((f,l),p,(f',l')\) \(\in \text{trcl pr-algo-lts'} \text{ for} \ p\)
using that

proof \(\text{(induction} \ p \text{ arbitrary: f l rule: length-induct)}\)

case \(1 p\) thus \(?case\)

proof \(\text{(cases} \ p\)
  case Nil thus \(?thesis\) by \(\text{auto}\)

next

case \([\text{simp}]: (Cons a p')\)
from \(1-prems\) obtain \(fh\) \(lh\)
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 (auto dest!: trcl-uncs)

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

show \(?thesis\)
proof (cases \( a = \text{SAT-PUSH}' (u,v) \))
case [simp]: False
from \ref{IH PP} have
length (filter ((=) (SAT-PUSH' (u, v))) p')
\leq 2 * card V - lh v
by auto
with \( \text{FIRST} \) show \(?thesis\)
apply (cases; clarsimp)
proof -
fix ua :: nat
assume a1: length (filter ((=) (SAT-PUSH' (u, v))) p')
\leq 2 * card V - relabel-effect f l ua v
assume a2: relabel-precond f l ua
have 2 * card V - relabel-effect f l ua v \leq 2 * card V - l v
\rightarrow length (filter ((=) (SAT-PUSH' (u, v))) p') \leq 2 * card V - l v

using a1 order-trans by blast
then show length (filter ((=) (SAT-PUSH' (u, v))) p')
\leq 2 * card V - l v
using a2 a1 by (metis (no-types) Labeling.relabel-increase-u Labeling-axioms diff-le-mono2 nat-less-le relabel-preserve-other)

qed
next
case [simp]: True

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

from \( \text{PRE} \) have \( (u,v)\in cf.E \quad l u = l v + 1 \)

unfolding sat-push-precond-def by auto
hence \( u\in V \quad v\in V \quad u\neq v \) using cfE-ss-invE E-ss-VxV by auto

have UVNEH: \( (u,v)\notin cfE-of fh \)
using \( (u\neq v) \)
apply (simp

unfolding Graph.E-def by simp

31
show \( ?\text{thesis} \)

**proof** (cases SAT-PUSH' \((u,v)\) \(\in\) set \(p'\))

- **case** False
  
  hence \([\text{simp}]:\) filter \((=)\) \((\text{SAT-PUSH}' (u,v))\) \(p' = []\)

  by (induction \(p'\)) auto

- **show** \( ?\text{thesis} \)

  using \( \text{bspec}\{OF height-bound \ (u\in V)\} \)

  using \( \text{bspec}\{OF height-bound \ (v\in V)\} \)

  using card-V-ge2 by simp

  by simp

**next**

- **case** True

  then obtain \(p1\) \(p2\)

  where \([\text{simp}]:\) \(p'=p1 \# \text{SAT-PUSH}' (u,v)\)#\(p2\)

  and \(NP1:\) SAT-PUSH' \((u,v)\) \(\not\in\) set \(p1\)

  using in-set-conv-decomp-first[of - \(p\)'] by auto

  from \(NP1\) have \([\text{simp}]:\) filter \((=)\) \((\text{SAT-PUSH}' (u,v))\) \(p1 = []\)

  by (induction \(p1\)) auto

  from \(PP\) obtain \(f2\) \(l2\) \(f3\) \(l3\)

  where \(\text{P1:\) }((fh,lh),p1,(f2,l2))\) \(\in\) trcl pr-algo-lts'

  and \(S:\) \((f2,l2),\text{SAT-PUSH}' (u,v),(f3,l3))\) \(\in\) pr-algo-lts'

  and \(P2:\) \((f3,l3),p2,(f',l'))\)\(\in\)trcl pr-algo-lts'

  by (auto simp: trcl-conv)

  from \(S\) have \((u,v)\)\(\in\)c\(E\)-of \(f2\) and \([\text{simp}]:\) \(l3=l2\)

  by (auto elim!: pr-algo-lts'.cases simp: sat-push-precond-def)

  with edge-reappears-at-increased-labeling\{OF \(P1\) - UVNEH\}

  \((l\ u = l\ v + 1)\)

  have \(AUX1\): \(l\ v < l2\ v\) by auto

  from \(S\) interpret \(l2\): Height-Bounded-Labeling c s t \(f2\ l2\)

  by (auto elim!: pr-algo-lts'.cases)

  from spec\{OF \(1.1H\), of \ SAT-PUSH' \((u,v)\)#\(p2\)\} \(S\) \(P2\) have

  Suc (length (filter \((=)\) \((\text{SAT-PUSH}' (u,v))\) \(p2\)))

  \(\leq 2 \ast\) card \(V\) \(-\) \(l2\ v\)

  by (auto simp: trcl-conv)

  also have \(\ldots\ + 1 \leq 2\ast\) card \(V\) \(-\) \(l\ v\)

  using \(AUX1\)

  using \(\text{bspec}\{OF \(l2\).height-bound \ (u\in V)\}\)

  using \(\text{bspec}\{OF \(l2\).height-bound \ (v\in V)\}\)

  by auto

  finally show \( ?\text{thesis} \)

  by simp

  qed

  qed

  qed
thus ?thesis using assms by fastforce
qed

lemma sat-push-action-bound' :
  assumes A : ((f,l),p,(f',l')) ∈ trcl pr-algo-lts'
  shows length (filter is-SAT-PUSH' p) ≤ 4 * card V * card E
proof
  from A have IN-E: e∈E∪E−1 if SAT-PUSH' e ∈ set p for e
    using that cfE-of-ss-invE
  apply (induction p arbitrary: f l)
  apply (auto simp: trcl-conv sat-push-precond-def
          elim!: pr-algo-lts'.cases
          ; blast)+
  done
have AUX: length (filter (λa. (∃ e∈S. a = SAT-PUSH' e)) p)
  = (∑ e∈S. length (filter ((=) (SAT-PUSH' e)) p)) if finite S for S
  using that
  apply induction
  apply simp
  apply clarsimp
  apply (subst length-filter-disj-or-conv; clarsimp)
  apply (fo-rule arg-cong)
  subgoal premises by (induction p) auto
  done
have is-SAT-PUSH' a = (∃ e∈E∪E−1. a = SAT-PUSH' e) if a∈set p for a
  using IN-E that by (cases a) auto
hence length (filter is-SAT-PUSH' p)
  = length (filter (λa. (∃ e∈E∪E−1. a = SAT-PUSH' e)) p)
  by (auto cong: filter-cong)
also have ... = (∑ e∈E∪E−1. length (filter ((=) (SAT-PUSH' e)) p))
  by (auto simp: AUX)
also have ... ≤ (∑ i∈E ∪ E−1. 2 * card V)
  using sum-mono[OF sat-push-edge-action-bound'[OF A], where K=E∪E−1] .
also have ... ≤ 4 * card V * card E using card-Un-le[of E E−1] by simp
finally show length (filter is-SAT-PUSH' p) ≤ 4 * card V * card E .
qed

end — Network

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

• A non-saturating push deactivates the source node and may activate
the target node. As the source node’s label is higher, the potential decreases.

- A saturating push may activate a node, thus increasing the potential by $O(V)$.
- A relabel operation may increase the potential by $O(V)$.

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

This argumentation corresponds to [Cormen 26.23].

Sum of heights of all active nodes

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

**context** Height-Bounded-Labeling

begin

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

**lemma** nonsat-potential-bound:

- **shows** nonsat-potential $f l \leq 2 \ast (\text{card } V)^2$
- **proof** –
  - have nonsat-potential $f l = (\sum v \in \{v \in V. 0 < \text{excess } f v\}. l v)$
  - unfolding nonsat-potential-def by auto
  - also have ... $\leq (\sum v \in V. l v)$
  - 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 ... $\leq (\sum v \in V. 2 \ast \text{card } V - 1)$
    - using height-bound by (meson sum_mono)
  - also have \(\ldots = \text{card } V \ast (2 \ast \text{card } V - 1)\) by auto
  - also have \(\text{card } V \ast (2 \ast \text{card } V - 1) \leq 2 \ast \text{card } V \ast \text{card } V\) by auto
  - finally show ?thesis by (simp add: power2_eq_square)
  - qed

A non-saturating push decreases the potential.

**lemma** nonsat-push-decr-nonsat-potential:

- **assumes** nonsat-push-precond $f l e$
- **shows** nonsat-potential \((\text{push-effect } f e) l < \text{nonsat-potential } f l\)
- **proof** (cases $e$)
  - case [simp]: (Pair $u v$)
    - show ?thesis
    - proof simp
      - interpret push-effect-locale $c s t f l u v$
      - apply unfold_locales using assms

34
by (simp add: push-precond-eq-sat-or-nonsat)

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

define S where S={x∈V. x≠u ∧ x≠v ∧ 0<excess f x}

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

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

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

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

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

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

lemma sat-push-nonsat-potential:
  assumes PRE: sat-push-precond f l e
  shows nonsat-potential (push-effect f e) l
    ≤ nonsat-potential f l + 2 * card V
proof –
  obtain u v where [simp]: e = (u, v) by (cases e) auto

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

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

  Only target node may get activated
  have {v∈V. excess f’ v > 0} ⊆ insert v {v∈V. excess f v > 0}
    using Δ-positive
    by (auto simp: excess’-if)

Thus, potential increases by at most l v
A relabeling increases the potential by at most $O(V)$.

**Lemma relabel-nonsat-potential**:

* Assumes `PRE: relabel-precond f l u`
* Shows `nonsat-potential f (relabel-effect f l u) \leq nonsat-potential f l + 2 \cdot \text{card } V`

**Proof** —

* Have `[simp, intro!]: finite \{v \in V. \text{excess } f v > 0\}` by `auto`

  * Let $?l' = \text{relabel-effect } f l u$


  * From `PRE` have `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 `nonsat-potential f ?l' = \text{sum } \{v \in V. \text{excess } f v\} - \{u\} + ?l' u`

    * Unfolding `nonsat-potential-def` using `U-ACTIVE` by `(auto intro: sum-arb)`

    * Also have `\text{sum } \{v \in V. \text{excess } f v\} - \{u\}`

      * = `\text{sum } l \{v \in V. \text{excess } f v\} - \{u\}`

    * Using `relabel-preserve-other` by `auto`

    * Also have `$?l' u \leq l u + 2 \cdot \text{card } V$

    * Using `$l'.height-bound[OF :u\in V]\]$ by `auto`

    * Finally have `nonsat-potential f ?l' \leq \text{sum } l \{v \in V. \text{excess } f v\} - \{u\} + l u + 2 \cdot \text{card } V` by `auto`

    * Also have `\text{sum } l \{v \in V. \text{excess } f v\} - \{u\}`

      * = `nonsat-potential f l`

    * Unfolding `nonsat-potential-def` using `U-ACTIVE` by `(auto intro: sum-arb[symmetric])`

    * Finally show `?thesis`.

**QED**

end — Height Bounded Labeling
context Network

begin

lemma nonsat-push-action-bound':
  assumes A: ((f,l),p,(f',l')) ∈ trcl pr-algo-lts'
  shows length (filter is-NONSAT-PUSH' p) ≤ 18 * (card V)^2 * card E

proof

  have B1: length (filter is-NONSAT-PUSH' p)
       ≤ nonsat-potential f l
      + 2 * card V * (length (filter is-SAT-PUSH' p))
      + 2 * card V * (length (filter is-RELABEL' p))
    using A

  proof (induction p arbitrary: f l)
  case Nil thus ?case by auto
  next
case [simp]: (Cons a p)
  then obtain fh lh
  where FIRST: ((f,l),a,(fh,lh))∈pr-algo-lts'
    and PP: ((fh,lh),p,(f',l')) ∈ trcl pr-algo-lts'
  by (auto simp: trcl-conv)
  note IH = Cons.IH[OF PP]

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

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

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)

  37
\[ \leq 2 \times (\text{card } V)^2 + 8 \times (\text{card } V)^2 \times \text{card } E + 4 \times (\text{card } V)^3 \]

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

by linarith

qed

end — Network

2.5.9 Assembling the Final Theorem

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

lemma (in Height-Bounded-Labeling) push-action-bound:

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

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

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

apply (erule sat-push-action-bound'; fail)

apply (erule nonsat-push-action-bound'; fail)

apply (auto simp: power2-eq-square)

done

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

fun (in Network) cost-estimate :: pr-operation \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:

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

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

proof —

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

proof (induction p)

case Nil

then show ?case by simp

next

case (Cons a p)

then show ?case by (cases a) auto

qed

also have \(\text{card } V \times \text{length} \left( \text{filter is-RELABEL} p \right) \leq 2 \times (\text{card } V)^3\)
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.

proof
  show ?G1 using pp-init-height-bound Height-Bounded-Labeling
  proof
    from A interpret Height-Bounded-Labeling c s t f l
    apply (induction p arbitrary: f l rule: rev-induct)
    done
  from pr-algo-term-maxflow show ?G2 by simp
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**: \([ \text{Height-Bounded-Labeling } c s t f l; \text{push-precond } f l e ] \)
  \[ \Rightarrow ((\text{push-effect } f e l), (f, l)) \in \text{pr-algo-rel} \]
- **relabel**: \([ \text{Height-Bounded-Labeling } c s t f l; \text{relabel-precond } f l u ] \)
  \[ \Rightarrow ((f, \text{relabel-effect } f l u), (f, l)) \in \text{pr-algo-rel} \]

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

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

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

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

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

**qed**

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

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

**proof**
- **have** \( \text{FIN-OPS} : \text{finite } (\text{UNIV::pr-operation set}) \)
  - **apply** (rule finite-subset[where \( B = \{ \text{PUSH, RELABEL} \} \])
  - **using** \( \text{pr-operation.exhaust by auto} \)
- **have** \{ \( p. \exists f' l'. ((f, l), p, (f', l')) \in \text{trcl } \text{pr-algo-lts} \} \subseteq \{ \text{p. length } p \leq \text{pr-algo-len-bound} \}
  - **by** (auto simp: pr-algo-lts-length-bound)
- **also note** \( \text{finite-lists-length-le[OF FIN-OPS, simplified]} \)
- **finally** (finite-subset) show \( \text{?thesis} \).
**qed**

**definition pr-algo-measure**
\[ \equiv \lambda (f, l). \text{Max} \{ \text{length } p | p. \exists aa ba. ((f, l), p, (f', l')) \in \text{trcl } \text{pr-algo-lts} \} \]
lemma pr-algo-measure:
assumes \((f',fl) \in pr-algo-rel\)
shows \(pr-algo-measure f' < pr-algo-measure fl\)
using assms
proof (cases fl'; cases fl; simp)
fix \(f l f' l'\)
assume \(A: ((f',l'),(f,l)) \in pr-algo-rel\)
then obtain \(a\) where LTS-STEP: \(((f,l),a,(f',l'))\in pr-algo-lts\)
by cases (auto intro: pr-algo-lts.intros)
from \(A\) interpret Height-Bounded-Labeling \(c s t f l\)
by cases auto
from pr-algo-maintains-hb-labeling[OF LTS-STEP]
interpret \(f': Height-Bounded-Labeling\ c s t f' l'\).

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

have finite \(?S1\) using finite-image-set path-set-finite by blast
moreover have \(?S1 \neq \{\}\\) by (auto intro: exI[where \(x=\{\}\)])
ultimately obtain \(p fx lx\) where length \(p = \text{Max } ?S1\)
apply -
apply (drule (1) Max-in)
by auto

have finite \(?S2\) using finite-image-set f'.path-set-finite by blast
have \(?S2 \neq \{\}\\) by (auto intro: exI[where \(x=\{\}\)])
{
  assume MG: Max ?S2 \geq Max ?S1
from Max-in[OF \(\text{finite } ?S2\), \(?S2 \neq \{\}\)] obtain \(p fx lx\) where length \(p = \text{Max } ?S2\)
apply -
apply (auto simp: trcl-conv)
by auto
with MG LTS-STEP have
LEN: length \((a\#p)\) > Max ?S1
and \(P: ((f,l),\#p,(fx, lx)) \in trcl pr-algo-lts\)
by (auto simp: trcl-conv)
from \(P\) have length \((a\#p)\) \(\in ?S1\) by blast
from Max-ge[OF \(\text{finite } ?S1\) this] LEN have False by simp
} thus \(pr-algo-measure \((f', l') < pr-algo-measure \((f, l)\)\)
unfolding pr-algo-measure-def by (rule ccontr) auto
qed

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

end — Network

2.8 Gap Heuristics

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 \( l k \equiv \lambda v. \text{iff} k < l_v \land l_v < \text{card} V \text{ then card } V + 1 \text{ else } l_v \)

The gap heuristics preserves a valid labeling.

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

  have \( l'\text{-inc} : l' v \geq l_v \) for \( v \) unfolding \( l'\)-def gap-effect-def by auto

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

The gap heuristics also preserves the height bounds.

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

  show \( ?\text{thesis} \)
    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 \( f \ l \ u \) ≡ let \( \ell' = \text{relabel-effect} \ f \ l \ u \) in
if \( \text{gap-precond} \ (\ell' \ (l \ u)) \) then gap-effect \( \ell' \ (l \ u) \) else \( \ell' \)

The combined gap-relabel operation preserves a valid labeling.

**lemma** (in Labeling) gap-relabel-pres-Labeling:
assumes \( \text{PRE: relabel-precond} \ f \ l \ u \)
defines \( \ell' \equiv \text{gap-relabel-effect} \ f \ l \ u \)
shows Labeling \( c \ s \ t \ f \ \ell' \)
unfolding \( \ell'-\text{def gap-relabel-effect-def} \)
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 \( \text{PRE: relabel-precond} \ f \ l \ u \)
defines \( \ell' \equiv \text{gap-relabel-effect} \ f \ l \ u \)
shows Height-Bounded-Labeling \( c \ s \ t \ f \ \ell' \)
unfolding \( \ell'-\text{def gap-relabel-effect-def} \)
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 \( \text{sum-heights-measure} \ (\text{gap-effect} \ l \ k) \leq \text{sum-heights-measure} \ l \)
unfolding gap-effect-def sum-heights-measure-def
by (auto intro!: sum-mono)

**lemma** (in Height-Bounded-Labeling) gap-relabel-measure:
assumes \( \text{PRE: relabel-precond} \ f \ l \ u \)
shows \( \text{sum-heights-measure} \ (\text{gap-relabel-effect} \ f \ l \ u) < \text{sum-heights-measure} \ l \)
unfolding gap-relabel-effect-def
by (fastforce simp: Let-def)

Analogously to \( \text{pr-algo-rel} \), we provide a well-founded relation that overapproximates the steps of a push-relabel algorithm with gap heuristics.
inductive-set gap-algo-rel where
push: ![Height-Bounded-Labeling c s t f l; push-precond f l e]
⇒ ((push-effect f e,l),(f,l))∈gap-algo-rel
| relabel: ![Height-Bounded-Labeling c s t f l; relabel-precond f l u]
⇒ ((f,gap-relabel-effect f l u),(f,l))∈gap-algo-rel

lemma wf-gap-algo-rel[simp, intro!]: wf gap-algo-rel

proof –

have gap-algo-rel ⊆ inv-image (less-than <*lex*> less-than) (λ(f,l), (sum-heights-measure l, pr-algo-measure (f,l)))

using pr-algo-measure

using Height-Bounded-Labeling.gap-relabel-measure

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

thus ?thesis

by (rule-tac wf-subset; auto)

qed

end — Network

end

theory Prpu-Common-Inst

imports
Flow-Networks.Refine-Add-Fofu
Generic-Push-Relabel

begin

context Network

begin

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

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

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

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

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

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

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

3.1 Implementing the Discharge Operation

context Network
begin

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

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

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

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

}
return \((l, Q)\)
}

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

definition fifo-discharge \(f_0 l Q\) \(\equiv\) do {
  assert \((Q \neq [])\);
  let \(u = \text{hd} Q\); let \(Q = l l Q\);
  assert \((u \in V \land u \neq s \land u \neq t)\);

  \((f, l, Q) \leftarrow \text{FOREACH}c \{ v . (u, v) \in \text{cfE-of} f_0 \} (\lambda(f, l, Q). \text{excess} f u \neq 0) (\lambda v (f, l, Q)). \text{do} \}
  if \((l u = l v + 1)\) then do {
    \((f', l, Q) \leftarrow \text{fifo-push} f l Q (u, v);\)
    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 \((f, l, Q)\)
}

if \text{excess} f u \neq 0 then do {
  \((l, Q) \leftarrow \text{fifo-gap-relabel} f l Q u;\)
  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 \(f Q\) \(\equiv\) \(\text{distinct} Q \land \text{set} Q = \{ v \in V - \{s, t\}. \text{excess} f v \neq 0 \}\)

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

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

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

lemma Q-invar-when-discharged2: \( [\text{QD-invar} u f Q; \text{excess} f u \neq 0] \implies \text{Q-invar} f (Q[u]) \)
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 $\text{thesis}$
  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 $\text{simp}$: $excess f v = 0$
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 $\text{thesis}$
  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-vec):
assumes $DINV$: Height-Bounded-Labeling $c s t f l$
assumes $QINV$: $Q$-invar $f Q$ and $QNE$: $Q \neq []$
shows $\text{fifo-discharge } f l Q \leq \text{SPEC } (\lambda(f',l',Q').$
  Height-Bounded-Labeling $c s t f' l'$
  $\land Q$-invar $f' Q'$
  $\land ((f',l'),(f,l)) \in \text{gap-algo-rel}^+$
)

proof –
  from $QNE$ obtain $u Qr$ where $\text{simp}$: $Q = u \# Qr$ by (cases $Q$) auto
  from $QINV$ have $U$: $u \in V - \{s,t\}$ $QD$-invar $u f Qr$ and $XU$-orig: $excess f u \neq 0$
  qed
by \(\text{(auto simp: Q-invar-def QD-invar-def)}\)

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

show \(?\text{thesis}\)
using \(U\)
unfolding \(\text{fifo-discharge-def fifo-push-def fifo-gap-relabel-def}\)
apply \(\text{(simp only: split nres-monad-laws)}\)
apply \(\text{(rewrite in FOREACHc - - \(\square\) - vcg-intro-frame)}\)
apply \(\text{(rewrite in if excess - - \(\not=\) 0 then \(\square\) else - vcg-intro-frame)}\)
apply \(\text{(refine-vcg FOREACHc-rule[where}}\)
\(l=\lambda it \,(f',l',Q')\).
\(\text{Height-Bounded-Labeling c s t f' l'}\)
\(\land\) \(\text{QD-invar u f' Q'}\)
\(\land\) \(\{u',v'\in\text{cfE-of f}' \}\)
\(\land\) \(\text{it} \subseteq \{v. (u,v) \in \text{cfE-of f}' \}\)
\(\land\) \(\text{excess f'} u \not= 0 \implies (\forall v \in \{v. (u,v) \in \text{cfE-of f}' \}\)\)
\(\land\) \(\text{it} \subseteq \{v. (u,v) \in \text{cfE-of f}' \}\)

apply \(\text{(vc-solve simp: DINV QINV it-step-insert-iff split del: if-split)}\)

subgoal for \(v\) \(\text{it f' l'}\)
proof
\(\text{assume HBL: Height-Bounded-Labeling c s t f' l'}\)
then interpret \(l':\text{Height-Bounded-Labeling c s t f' l'}\).

assume \(X: \text{excess f'} u \not= 0\) \(\text{and UI: u \in V} \quad u \not= s \quad u \not= t\)
and \(\text{QDI: QD-invar u f' Q'}\)

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

assume \(\text{REL: ((f',l'), f, l)} \in \text{gap-algo-rel}^*\)
assume \(\text{SAT-EDGES: } \forall v \in \{v. (u,v) \in \text{cfE-of f}' \}\) \(\land\) \(\text{it} \subseteq \{v. (u,v) \in \text{cfE-of f}' \}\)
\(\land\) \(\text{it} \subseteq \{v. (u,v) \in l'.\text{cf.E}\}\)

from \(X\) \(UI\) \(l'.\text{excess-non-negative}\) have \(X': \text{excess f'} u > 0\) \(\text{by force}\)

have \(\text{PP: push-precond f' l' (u, v)}\) \(\text{if l' u = l' v + 1}\)
unfolding \(\text{push-precond-def}\) using that \(\text{UVE X'}\) \(\text{by auto}\)

show \(?\text{thesis}\)
apply \(\text{(rule vcg-rem-frame)}\)
apply \(\text{(rewrite in if - then (assert - \(\Rightarrow\) \(\square\)) else - vcg-intro-frame)}\)
apply \(\text{refine-vcg}\)
apply \(\text{(vc-solve simp: REL solve: PP l'.push-pres-height-bound HBL QDI split del: if-split)}\)
subgoal proof –
assume [simp]: \( l' \ u = \text{Suc} (l' \ v) \)
assume \( \text{PRE} \): push-precond \( f' \ l' (u, v) \)
then interpret \( \text{pe} \): push-effect-locale \( c \ s t f' \ l' \ u \ v \) by unfold-locales

have \( \text{UVNE}' : l'.cf (u, v) \neq 0 \)
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: \( \text{pe}'.\text{cf}'-alt \) augment-edge-cf-def)
subgoal using \( l'.\text{push-activate-pres-QD-invar}[OF \text{QDI PRE}] \)
subgoal using \( l'.\text{push-no-activate-pres-QD-invar}[OF \text{QDI PRE}] \)
by auto
subgoal
by (meson gap-algo-rel.\( \text{push REL PRE converse-rtrancl-into-rtrancl HBL} \))

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 \( \text{pe}.f'-alt \)
apply (simp add: augment-edge-cf')
unfolding \( \text{Graph.E-def} \)
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 \( \text{pe.}\Delta\text{-def} \) by auto
hence \( \text{E'}\text{SS} : \text{pe.}\text{e'.cf}.E \subseteq (l'.\text{cf}.E \cup \{(v,u)\}) - \{(u,v)\} \)
unfolding \( \text{pe.}\text{f'}\text{-alt} \)
apply (simp add: augment-edge-cf')
unfolding \( \text{Graph.E-def} \)
by auto

assume \( v' \in \text{it} \implies v' = v \) and \( \text{UV'}E : (u, v') \in \text{pe.}\text{l'.cf}.E \) and \( \text{LUSLV'} : l' \ v = l' \ v' \)
with \( \text{E'}\text{SS} \) have \( v' \in \text{it} \) by auto
moreover from \( \text{UV'}E \text{E'}\text{SS pe.uv-not-eq(2)} \) 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

49
done
qed
subgoal using ITSS by auto
subgoal using SAT-EDGES by auto
done
qed
subgoal premises prems for f' l' Q' proof –
from prems interpret l': Height-Bounded-Labeling c s t f' l' by simp
from prems have UI: u∈V u≠s u≠t
and X: excess f' u ≠ 0
and QDI: QD-invar u f' Q'
and REL: ((f', l'), f, l) ∈ gap-algo-rel
and NO-ADM: ∀v. (u, v) ∈ l'.cf.E → l' u ≠ Suc (l' v)
by simp-all
from X have X': excess f' u > 0 using l'.excess-non-negative UI by force
from X' UI NO-ADM have PRE: relabel-precond f' l' u
unfolding relabel-precond-def by auto
from l'.height-bound (u∈V) card-V-ge2 have [simp]: l' u < 2*card V by auto
from l'.relabel-pres-height-bound[of PRE]
interpret l'': Height-Bounded-Labeling c s t f' relabel-effect f' l' u.
from l''.height-bound (u∈V) card-V-ge2 have [simp]: relabel-effect f' l' u u < 2*card V by auto
show ?thesis
apply (rule vcg-rem-frame)
apply refine-vcg
apply (vc-solve
  simp: UI PRE
  simp: l'.gap-relabel-pres-hb-labeling[of PRE]
  simp: Q-invar-when-discharged2[of QDI X])
subgoal by unfold-locales
subgoal
by (meson PRE REL gap-algo-rel.relabel l'.Height-Bounded-Labeling-axioms rtrancl-into-trancl2)
done
qed
subgoal by (auto simp: Q-invar-when-discharged1 Q-invar-when-discharged2)
subgoal using XU-orig by (metis Pair-inject rtranclD)
subgoal by (auto simp: Q-invar-when-discharged1)
subgoal using XU-orig by (metis Pair-inject rtranclD)
done
qed
end — Network

3.2 Main Algorithm

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

  Q ← spec l. distinct l ∧ set l = {v∈V − {s,t}. excess f v ≠ 0}; — TODO: This is exactly E''{s} − {t}!

  (f,l,−) ← 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
    I=λ(f,l,Q). Height-Bounded-Labeling c s t f l ∧ Q-invar f Q
    and R=inv-image (gap-algo-rel) (λ(f,l,Q). ((f,l)))
  ]
)
apply (vc-solve solve: pp-init-height-bound)
subgoal by (blast intro: wf-lex-prod wf-trancl)
subgoal unfolding Q-invar-def by auto
subgoal for initQ f l proof —
  assume Height-Bounded-Labeling c s t f l
  then interpret Height-Bounded-Labeling c s t f l.
  assume Q-invar f []
  hence ∀u∈V − {s,t}. excess f u = 0 unfolding Q-invar-def by auto
thus isMaxFlow f by (rule no-excess-imp-maxflow)
qed
done

end — Network

end

4 Topological Ordering of Graphs

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

4.1 List-Before Relation

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

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

list-before only relates elements of the list

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

list-before is a strict ordering, i.e., it is transitive and asymmetric.
lemma list-before-trans[trans]: distinct l ⟷ trans (list-before-rel l)
by (clarsimp simp: trans-def list-before-rel-alt) (metis index-nth-id less-trans)

lemma list-before-asym: distinct l ⟷ asym (list-before-rel l)
by (meson asym.intros irrefl-def list-before-irrefl-eq-distinct list-before-trans transE)

Structural properties on the list
lemma list-before-rel-empty[simp]: list-before-rel [] = {}
unfolding list-before-rel-def by auto

lemma list-before-rel-cons: list-before-rel (x#l) = ({x}×set l) ∪ list-before-rel l
apply (intro equalityI subsetI; simp add: split-paired-all)
subgoal for a b proof –
  assume (a,b) ∈ list-before-rel (x # l)
  then obtain i j where IDX-BOUND: i < j j < Suc (length l) and [simp]:
a=(x#l)!i   b=(x#l)!j
  unfolding list-before-rel-alt by auto
  
  { assume i=0
    hence x=a   b∈set l using IDX-BOUND
        by (auto simp: nth-Cons split: nat.splits)
  }
  moreover {
    assume i≠0
    with IDX-BOUND have a=l!(i−1)   b=l!(j−1)   i−1 < j−1   j−1 < length l
      by auto
    hence (a, b) ∈ list-before-rel l unfolding list-before-rel-alt by blast
  }
ultimately show ?thesis by blast
qed

subgoal premises prems for a b
proof –
  { assume [simp]: a=x and b∈set l
    then obtain j where b = l!j   j<length l by (auto simp: in-set-conv-nth)
    hence a=(x#l)!0   b = (x#l)!Suc j   0 < Suc j   Suc j < length (x#l)
      by auto
    hence ?thesis unfolding list-before-rel-alt by blast
  }
  moreover {
    assume (a, b) ∈ list-before-rel l
    hence ?thesis unfolding list-before-rel-alt
      by clarsimp (metis Suc-mono nth-Cons-Suc)
  }
ultimately show ?thesis using prems by blast
qed
done
4.2 Topological Ordering

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

We define the predicate \textit{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.

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

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

\textbf{unfolding} \textit{is-top-sorted-def} \textbf{by} \texttt{auto}

\textbf{lemma} \textit{is-top-sorted-empty-rel}[\texttt{simp}]: \textit{is-top-sorted} \{\} \ l \iff \text{distinct} \ l

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

\textbf{lemma} \textit{is-top-sorted-empty-list}[\texttt{simp}]: \textit{is-top-sorted} \(R \ [\]

\textbf{by} \texttt{(auto simp: is-top-sorted-def)}

A topological sorted list must be distinct

\textbf{lemma} \textit{is-top-sorted-distinct}:
\begin{itemize}
  \item \textbf{assumes} \textit{is-top-sorted} \(R \ l\)
  \item \textbf{shows} \text{distinct} \(l\)
\end{itemize}

\textbf{proof} \texttt{(rule ccontr)}
\begin{itemize}
  \item \textbf{assume} \(\neg \text{distinct} \ l\)
  \item \textbf{with list-before-irrefl-eq-distinct[of \(l\)]} \textbf{obtain} \(x\) \textbf{where} \((x,x) \in (\text{list-before-rel} \ l)\)
  \item \textbf{by} \texttt{(auto simp: irrefl-def)}
  \item \textbf{with \textit{assms} show} \text{False} \textbf{unfolding} \textit{is-top-sorted-def} \textbf{by} \texttt{auto}
\end{itemize}
\textbf{qed}

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

\textbf{by} \texttt{(auto simp: list-before-rel-cons)}

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

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

\textbf{lemma} \textit{is-top-sorted-remove-elem} \(\textit{is-top-sorted} \ (l1 @ x \# l2) \implies \textit{is-top-sorted} \ (l1 @ l2)\)

\textbf{by} \texttt{(auto simp: is-top-sorted-cons is-top-sorted-append)}

Removing edges from the graph preserves topological sorting

\textbf{lemma} \textit{is-top-sorted-antimono}:
\begin{itemize}
  \item \textbf{assumes} \(R \subseteq R'\)
  \item \textbf{assumes} \textit{is-top-sorted} \(R' \ l\)
\end{itemize}
shows is-top-sorted $R \ l$
using assumptions 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
```

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 (push-effect f e) l (is ?G1)
  and 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 -
    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 \text{augment-edge-def residualGraph-def Graph.E-def}
by (auto split!: if-splits prod.splits)
hence $l'.cf.E = \text{insert (prod.swap } e \text{) cf.E} - \{ e \} \lor l'.cf.E = \text{cf.E} - \{ e \}$
by auto
thus $\text{adm-edges } ?f' l = \text{adm-edges } f l - \{ e \}$
proof (cases rule: disjE[consumes 1])
case 1
from \text{sat} have $e \in \text{adm-edges } f l$
unfolding \text{sat-push-precond-def adm-edges-def}
by auto
thus $\text{adm-edges } ?f' l = \text{adm-edges } f l - \{ e \}$
next
case 2
thus $\text{adm-edges } ?f' l = \text{adm-edges } f l - \{ e \}$
unfolding \text{adm-edges-def 2}
by auto
qed
qed
thus \text{?thesis} by auto
next
case \text{nonsat}

hence $\text{adm-edges } (\text{push-effect } f e) l = \text{adm-edges } f l$
proof (cases $e$; simp add: \text{nonsat-push-alt})
fix $u \ v$ assume [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 \text{push-pres-Labeling[OF assms] nonsat-push-alt nonsat}
by auto

from \text{nonsat} have $e \in \text{adm-edges } f l$
unfolding \text{nonsat-push-precond-def adm-edges-def} by auto
with \text{adm-edges-inv-disj} have \text{AUX: prod.swap } e \notin \text{adm-edges } f l$
by (auto simp: \text{swap-in-iff-inv})

from \text{nonsat} have
$\text{excess } f u < \text{cf } (u, v) \quad 0 < \text{excess } f u$
and [simp]: $l u = l v + 1$
unfolding \text{nonsat-push-precond-def} by auto

hence $l'.cf.E \subseteq \text{insert (prod.swap } e \text{) cf.E} \quad l'.cf.E \supseteq \text{cf.E}$
unfolding $l'.cf-def cf-def$
unfolding \text{augment-edge-def residualGraph-def Graph.E-def}
apply (safe)
apply (simp split: if-splits)
apply (simp split: if-splits)
subgoal

57
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 \( l'.cf.E = \) insert (prod.swap e) cf.E \( \lor l'.cf.E = cf.E \)
  by auto

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

qed

hence \( l' \).

cf.\(E = insert (prod.swap e) cf.\(E \lor cf.\(E = cf.\(E \)
  by auto

thus \( ?thesis \) by auto

qed

thus \(?G1 \ ?G2 \) by auto

qed

After a relabel operation, there is at least one admissible edge leaving the
relabeled node, but no admissible edges do enter the relabeled node [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 PRE: relabel-precond \( f \ l \ u \)
  defines \( l' \equiv relabel-effect \( f \ l \ u \)
  shows adm-edges \( f \ l' \cap cf.outgoing \ u \neq \{\} \) (is \( ?G1 \)
    and adm-edges \( f \ l' \cap cf.incoming \ u = \{\} \) (is \( ?G2 \)
    and adm-edges \( f \ l' = cf.adjacent \ u = adm-edges \ l - cf.adjacent \ u \) (is \( ?G3 \)
    proof

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

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

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

58
hence \((\text{simp}): l' \; uh = l \; uh \; \text{unfolding} \; l'\text{-def relabel-effect-def} \; \text{by simp}\)

from \(1 \; \text{relabel-increase-u[OF PRE, folded} \; l'\text{-def]} \; \text{have} \; l \; uh > l \; u + 1\)

by simp

with \(\text{valid[OF UHUE]} \; \text{have} \; \text{False} \; \text{by auto}\)

\)

thus \(?G2 \; \text{by (auto simp: cf.incoming-def)}\)

show \(?G3\)

\(\text{unfolding} \; \text{adm-edges-def}\)

by \(\text{(auto simp: l'\text{-def relabel-effect-def cf.adjacent-def\)}\}

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: \([v \in \text{adjacent-nodes} \; u - n \; u] \Rightarrow (u, v) \notin \text{adm-edges} f\)

assumes neighbors-adj: \(n \; u \subseteq \text{adjacent-nodes} \; u\)

assumes neighbors-finite: \(\text{finite} (n \; u)\)

begin

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

lemma push-pres-nbr-invar:

assumes PRE: push-precond f l e

shows neighbor-invar c s t (push-effect f e) l n

proof \(\text{cases e}\)

case \(\text{(simp): (Pair u v)}\)

show \(?thesis proof simp\)

from PRE interpret push-effect-locale c s t f l u v

by unfold-locales simp

from push-pres-height-bound[OF PRE]

interpret l': Height-Bounded-Labeling c s t f' l .

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
    apply (auto)
    by (smt DiffD2 Diff-triv adm-edges-def cf.incoming-def mem-Collect-eq prod.simps(2) relabel-preserve-other)
qed
qed

lemma excess-nz-iff-gz: [ u\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 cfE-ss-invE
unfolding relabel-precond-def adm-edges-def
by (auto simp: adjacent-nodes-def excess-nz-iff-gz)

lemma remove-neighbor-pres-nbr-invar: (u,v)\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

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 \begin{cases} 
\text{assert } (u \in V - \{s,t\}); \\
\text{while } \not(T(\lambda(f, l, n). \text{excess } f \neq 0)) \begin{cases} 
\text{v} \leftarrow \text{select } v. v \in n u; \\
\text{case v of} \\
\text{None } \Rightarrow \begin{cases} 
\text{l} \leftarrow \text{relabel } f, l, n (u \leftarrow \text{adjacent-nodes } u) \\
\text{return } (f, l, n(u := \text{adjacent-nodes } u)) \\
\text{| Some } v \Rightarrow \begin{cases} 
\text{assert } (v \in V \land (u, v) \in E \cup E^{-1}); \\
\text{if } ((u, v) \in cfE-of f \land l u = l v + 1) \text{ then do} \\
\text{f } \leftarrow \text{push } f, l (u, v); \\
\text{return } (f, l, n) \\
\text{| else do} \\
\text{assert } ((u, v) \notin \text{adm-edges } f, l); \\
\text{return } (f, l, n(u := n u - \{v\})) \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \

Invariant for the discharge loop

locale discharge-invar =
  neighbor-invar c s t f l n
  + lo: neighbor-invar c s t f l n
for c s t and u :: node and fo lo no f l n +
assumes lu-incr: lo u ≤ l u
assumes u-node: u ∈ V − {s,t}
assumes no-relab-adm-edges: lo u = l u ⇒ adm-edges f l ⊆ adm-edges fo lo
assumes no-relab-excess:
  [lo u = l u; u≠v; excess fo v ≠ excess f v] ⇒ (u,v)∈adm-edges fo lo
assumes adm-edges-leaving-u: (u',v)∈adm-edges f l − adm-edges fo lo ⇒ u'=u
assumes relab-u-no-incoming-adm: lo u ≠ l u ⇒ (v,u)∉adm-edges f l
assumes algo-rel: ((f,l),(fo,lo)) ∈ pr-algo-rel*
begin

lemma u-node-simp1[simp]: u≠s u≠t s≠u t≠u using u-node by auto
lemma u-node-simp2[simp, intro!]: u∈V using u-node by auto
lemma dis-is-lbl: Labeling c s t f l by unfold-locales

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

lemma dis-is-nbr: neighbor-invar c s t f l n by unfold-locales

lemma new-adm-imp-relabel:
\[(u',v) \in \text{adm-edges} f l \rightarrow \text{adm-edges} f o \quad \text{lo} \quad u \neq l u\]
using no-relabel-adm-edges adm-edges-leaving-u by auto

lemma push-pres-dis-invar:
assumes \(\text{PRE}: \text{push-precond} f l (u,v)\)
shows discharge-invar c s t u f o no \((\text{push-effect} f (u,v))\) l n
proof –
from \(\text{PRE}\) interpret push-effect-locale by unfold-locales
from push-pres-nbr-invar[\(\text{OF PRE}\)] interpret neighbor-invar c s t f' l n.

show discharge-invar c s t u f o lo no f' l n
apply unfold-locales
subgoal using lu-incr by auto
subgoal by auto
subgoal using no-relabel-adm-edges push-adm-edges(2)[\(\text{OF PRE}\)] by auto
subgoal for \(v'\) proof –
assume \(\text{LOU}: \text{lo} \quad u = l u\)
assume \(\text{EXNE}: \text{excess} f o \quad v' \neq \text{excess} f' v'\)
assume \(\text{UNV}': u \neq v'\)

\begin{align*}
&\text{assume} \quad \text{excess} f o \quad v' \neq \text{excess} f v' \\
&\text{from no-relabel-excess}[\(\text{OF LOU UNV'}\) \text{this}] \text{have} \quad ?\text{thesis} .
\end{align*}

moreover \{}
assume \(\text{excess} f o \quad v' = \text{excess} f v'\)
\begin{align*}
&\text{with EXNE have} \quad \text{excess} f v' \neq \text{excess} f' v' \quad \text{by simp} \\
&\text{hence} \quad v' = v \text{ using UNV'} by \ (\text{auto simp: excess'\-if split: if\-splits}) \\
&\text{hence} \quad ?\text{thesis using no-relabel-adm-edges}[\(\text{OF LOU}\) uv-adm by auto]
\end{align*}
\}
ultimately show \quad ?\text{thesis by blast}
qed

subgoal
by (meson Diff-iff push-adm-edges(2)[\(\text{OF PRE}\)] adm-edges-leaving-u subsetCE)
subgoal
using push-adm-edges(2)[\(\text{OF PRE}\)] relabel-u-no-incoming-adm by blast
subgoal
using converse-rtrancl-into-rtrancl[
\begin{align*}
&\text{OF pr-algo-rel.push}[\(\text{OF dis-is-hbl PRE}\) algo-rel]
\end{align*}
].
done
qed

lemma relabel-pres-dis-invar:

62
assumes \( \text{PRE}: \text{relabel-precond } f \ u \)
shows \( \text{discharge-invar } c \ s \ t \ u \ f o \ l o \ no \ f \)
\((\text{relabel-effect } f \ u) \ (n(u := \text{adjacent-nodes } u))\)

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

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

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

subgoal
proof
using relabel-adm-edges(2)[\(\text{OF PRE}\)]
unfolding \( \text{adm-edges-def } \text{cf.incoming-def } \)
by fastforce

subgoal
using converse-rtrancl-into-rtrancl[\(\text{OF pr-algo-rel relabel }[\text{OF dis-is-hbl PRE } \text{algo-rel}]\)]

. done
qed

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

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

shows discharge-invar \( c s t u f o l o no f l n ' \)

proof –

from remove-neighbor-pres-nbr-invar[OF PRE]
interpret neighbor-invar \( c s t f l n ' \) unfolding \( n' \)-def

show \(?thesis
  apply unfold-locales
  by auto
qed

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

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

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

end

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

context Network begin

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

lemma discharge-correct[THEN order-trans, refine-vcg]:
assumes DINV: neighbor-invar \( c s t f l n \)
assumes NOT-ST: \( u \neq t \quad u \neq s \) and UIV: \( u \in V \)
shows discharge \( f l n u \)
  \( \leq \) SPEC \( (\lambda(f',l',n'). \) discharge-invar \( c s t u f l n f' l' n' \)
  \( \wedge \) excess \( f' u = 0 \)
unfolding discharge-def push-def relabel-def
apply (refine-vcg WHILET-rule|where
  I=\( \lambda(f',l',n'). \) discharge-invar \( c s t u f l n f' l' n' \)
  and R=inv-image (pr-algo-rel \( \preceq \)lex\( \) finite-psubset
  \( (\lambda(f',l',n'). (\langle f',l'\rangle,n' u)) \))

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

ccontext Network
begin

Initially, all edges are unprocessed.

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

lemma rtf-init-n-finite[simp, intro!]: finite (rtf-init-n u)
    unfolding rtf-init-n-def
    by auto

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

lemma rtf-init-neighbor-invar:
    neighbor-invar c s t pp-init-f pp-init-l rtf-init-n

proof —
from pp-init-height-bound
interpret Height-Bounded-Labeling c s t pp-init-f pp-init-l .

have [simp]: rtf-init-n u ⊆ adjacent-nodes u for u
by (auto simp: rtf-init-n-def)

show \?thesis by unfold-locales auto
qed

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

let L-left=[];
L-right \leftarrow spec l. distinct l \land set l = V - \{s,t\};

(f,l,n,L-left,L-right) \leftarrow while T 
(\lambda(f,l,n,L-left,L-right). L-right \not= [])
(\lambda(f,l,n,L-left,L-right). do {
  let u = hd L-right;
  assert (u \in V);
  let old-lu = l u;
  (f,l,n) \leftarrow discharge f l n u;
  if (l u \not= old-lu) then do 
    — Move u to front of l, and restart scanning L
    let (L-left,L-right) = ([u],L-left @ tl L-right);
    return (f,l,n,L-left,L-right)
  } else do {
    — Goto next node in l
    let (L-left,L-right) = (L-left@[u], tl L-right);
    return (f,l,n,L-left,L-right)
  }
}) (f,l,n,L-left,L-right);

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

end — Network

Invariant for the main algorithm:

1. Nodes in the queue left of the current node are not active
2. The queue is a topological sort of the admissible network
3. All nodes except source and sink are on the queue
locale rtf-invar = neighbor-invar +
  fixes L-left L-right :: node list
  assumes left-no-excess: ∀ u ∈ set (L-left). excess f u = 0
  assumes L-sorted: is-top-sorted (adm-edges f l) (L-left @ L-right)
  assumes L-set: set L-left ∪ set L-right = V − {s, t}
begin
  lemma rtf-is-nbr: neighbor-invar c s t f l n by unfold-locales

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

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

  end

context Network begin

lemma rtf-init-invar:
  assumes DIS: distinct L-left and L-set: set L-left = V − {s, t}
  shows rtf-invar c s t pp-init-f pp-init-l rtf-init-n [] L-left
  proof
    from rtf-init-neighbor-invar interpret neighbor-invar c s t pp-init-f pp-init-l rtf-init-n .
    show ?thesis using DIS L-set by unfold-locales auto
  qed

theorem relabel-to-front-correct:
  relabel-to-front ≤ SPEC isMaxFlow
  unfolding relabel-to-front-def
  apply (rewrite in whileT - □ vcg-intro-frame)
  apply (refine-vcg
    WHILELET-rule[where
      l=λ(f,l,n,L-left,L-right). rtf-invar c s t f l n L-left L-right
      and R=inv-image
        (pr-algo-rel+ <*lex*> less-than)
        (λ(f,l,n,L-left,L-right). ((f,l),length L-right))
    ]
  )
  apply (vc-solve simp: rtf-init-invar rtf-invar rtf-is-nbr)
  subgoal by (blast intro: wf-lex-prod wf-trancl)
  subgoal for - f l n L-left L-right proof
    assume rtf-invar c s t f l n L-left L-right
    then interpret rtf-invar c s t f l n L-left L-right .
assume $L\text{-}right \neq []$ then obtain $u \ L\text{-}right'$

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

from $L\text{-}set$ have [simp]: $u \in V \ u \neq s \ u \neq t \ s \neq u \ t \neq u$ by auto

from $L\text{-}distinct$ 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 \[\(\text{cases }L\text{-}right\)\] auto)

from $L\text{-}set$ have [simp]: $\exists x \ s \ t \ u \ f \ l \ n \ f' \ l' \ n'$ by auto

show $rtf\text{-}invar c \ s \ t \ f' \ l' \ n'$ using $L\text{-}sorted$ 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 $L\text{-}right \neq []$ then obtain $u L\text{-}right'$

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

from $L\text{-}set$ have [simp]: $u \in V \ u \neq s \ u \neq t \ s \neq u \ t \neq u$ by auto

from $L\text{-}distinct$ 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 \[\(\text{cases }L\text{-}right\)\] auto)

from $L\text{-}set$ have [simp]: $\exists x \ s \ t \ u \ f \ l \ n \ f' \ l' \ n'$ by auto

show $rtf\text{-}invar c \ s \ t \ f' \ l' \ n'$ using $L\text{-}sorted$ 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 $L\text{-}set$ have [simp]: $\exists x \ s \ t \ u \ f \ l \ n \ f' \ l' \ n'$ by auto

show $rtf\text{-}invar c \ s \ t \ f' \ l' \ n'$ using $L\text{-}sorted$ 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 $L\text{-}set$ have [simp]: $\exists x \ s \ t \ u \ f \ l \ n \ f' \ l' \ n'$ by auto

show $rtf\text{-}invar c \ s \ t \ f' \ l' \ n'$ using $L\text{-}sorted$ 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 $L\text{-}set$ have [simp]: $\exists x \ s \ t \ u \ f \ l \ n \ f' \ l' \ n'$ by auto

show $rtf\text{-}invar c \ s \ t \ f' \ l' \ n'$ using $L\text{-}sorted$ 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.
subgoal using \textit{\textit{L-set}} by \textit{auto} 
done

qed

subgoal using \textit{l'.algo-rel} by \textit{(auto dest: rtranclD)}
subgoal proof —
  assume \textit{NO-RELABEL[simp]}: \textit{l' u = l u}
  — Intuition: non-zero excess would imply an admissible edge contrary to \textit{top-sorted}.
  have \textit{AUX: excess f' v = 0 if v \in set L-left for v}
  proof (rule ccontr)
      from that \textit{\langle u \notin set L-left \rangle} have \textit{u \neq v} by blast
  moreover assume \textit{excess f' v \neq 0}
  moreover from \textit{that left-no-excess have excess f v = 0 by auto}
  ultimately have \textit{(u,v) \in adm-edges f l}
      using \textit{l'.no-relabel-excess[OF NO-RELABEL[symmetric]]}
      by \textit{auto}
  by \textit{auto}

  with \textit{L-sorted} that show \textit{False}
      by \textit{(auto simp: is-top-sorted-append is-top-sorted-cons)}
  qed

show \textit{rtf-invar c s t f' l' n' (L-left @ [u]) L-right'}
  apply \textit{unfold-locales}
subgoal by \textit{(auto simp: AUX)}
subgoal
  apply \textit{(rule is-top-sorted-antimono)}
      \textit{OF l'.no-relabel-adm-edges[OF NO-RELABEL[symmetric]]}
  using \textit{L-sorted by simp}
subgoal using \textit{L-set by auto}
done

qed
subgoal using \textit{l'.algo-rel} by \textit{(auto dest: rtranclD)}
done

qed
done
subgoal by \textit{(auto intro: rtf-invar.terminated-imp-maxflow)}
done

end — Network

end

6 Tools for Implementing Push-Relabel Algorithms

theory \textit{Prpu-Common-Impl}
imports
  \textit{Prpu-Common-Inst}
  \textit{Flow-Networks.NetCheck}
  \textit{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 Δ ≡ do
assert (u ∈ V);
return (x(u := x u + Δ))

6.1.2 Labeling

Obtain the initial labeling: All nodes are zero, except the source which is labeled by |V|. The exact cardinality of V is passed as a parameter.
definition l-init :: nat ⇒ (node ⇒ nat) nres
where l-init C ≡ return ((λ_. 0)(s := C))

Get the label of a node.
definition l-get :: (node ⇒ nat) ⇒ node ⇒ nat nres
where l-get l u ≡ do
assert (u ∈ V);
return (l u)

Set the label of a node.
definition l-set :: (node ⇒ nat) ⇒ node ⇒ nat ⇒ (node ⇒ nat) nres
where l-set l u a ≡ do
assert (u ∈ V);
assert (a < 2*card V);
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.

\[
\text{return } (l(u := a))
\]

Get the count for a label value.

\[
\text{definition } \text{cnt-get} :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat \text{ nres}
\]

\[
\text{where } \text{cnt-get } \text{cnt lv } \equiv \text{ do }
\]

\[
\begin{align*}
\text{assert } (lv < 2\times N); \\
\text{return } (\text{cnt lv})
\end{align*}
\]

Increment the count for a label value by one.

\[
\text{definition } \text{cnt-incr} :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow (nat \Rightarrow nat) \text{ nres}
\]

\[
\text{where } \text{cnt-incr cnt lv } \equiv \text{ do }
\]

\[
\begin{align*}
\text{assert } (lv < 2\times N); \\
\text{return } (\text{cnt } lv := \text{cnt lv + 1})
\end{align*}
\]

Decrement the count for a label value by one.

\[
\text{definition } \text{cnt-decr} :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow (nat \Rightarrow nat) \text{ nres}
\]

\[
\text{where } \text{cnt-decr cnt lv } \equiv \text{ do }
\]

\[
\begin{align*}
\text{assert } (lv < 2\times N \wedge \text{cnt lv} > 0); \\
\text{return } (\text{cnt } lv := \text{cnt lv - 1})
\end{align*}
\]

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

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

\[
\text{lemma } \text{xf-rel-RELATES[refine-dref-RELATES]}: \text{ RELATES xf-rel}
\]
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 z = 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 (λ(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 adjacent-nodes s for u
  using that unfolding adjacent-nodes-def by auto
show ?thesis
unfolding pp-init-xcf2-aux-def
apply (refine-vcg FOREACH-rule[where I=λit (x,cf).
  x s = (\sum v \in adjacent-nodes s - it, - c(s,v))
  \land (\forall v \in 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 adjacent-nodes s. if v\notin it then cf (s,v) = 0 \land cf (v,s) = c (s,v)
  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 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=snd] intro: inj-onI)
  applyS (metis (mono-tags, lifting) Compl-iff Graph.zero-cap-simp insertE
  mem-Collect-eq)
  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-reg
  using E-ss-VxV
  by auto

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

6.2.3 Computing the Minimal Adjacent Label

definition (in Network) min-adj-label-aux cf l u ≡ do 
  assert (u∈V);
  x ← foreach (adjacent-nodes u) (λv x. do {
    assert (((a,v)∈E∪E−1));
    assert (v∈V);
  }) (x,cf)

if (cf (u, v) ≠ 0) then
  case x of
    None ⇒ return (Some (l v))
  | Some xx ⇒ return (Some (min (l v) (xx)))
else
  return x
}} None;
assert (x≠None);
return (the x)
}

lemma (in −) set-filter-xform-aux:
{ f x | x = a ∨ x∈S ∧ x≠it ) ∧ P x }
= (if P a then {a} else {}) ∪ {f x | x∈S−it ∧ P x}
by auto

lemma (in Labeling) min-adj-label-aux-spec:
assumes PRE: relabel-precond f l u
shows min-adj-label-aux cf l u ≤ SPEC (λx. x = Min { l v | v. (u,v)∈cf.E })
proof −
have AUX: cf (u,v) ≠ 0 ←→ (u,v)∈cf.E for v unfolding cf.E-def by auto

have EQ: { l v | v. (u,v)∈cf.E }
= { l v | v. v∈adjacent-nodes u ∧ cf (u,v)≠0 }

unfolding AUX
using cfE-ss-invE
by (auto simp: adjacent-nodes-def)

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

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

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

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 using (auto simp: Min.insert-remove) by auto
subgoal using AUX2 by auto
done

qed

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

lemma min-adj-label-refine[THEN order-trans, refine-vcg]:
assumes Height-Bounded-Labeling c s t f l
assumes AM: (am,adjacent-nodes)∈nat-rel→⟨nat-rel⟩list-set-rel
assumes PRE: relabel-precond f l u
assumes [simp]: cf = cf-of f
shows min-adj-label am cf l u ≤ SPEC (λx. x
= Min { l v | v. (u,v)∈cfE-of f })
proof –
interpret Height-Bounded-Labeling c s t f l by fact
have min-adj-label am (cf-of f) l u ≤ ⊥Id (min-adj-label-aux (cf-of f) l u)
unfolding min-adj-label-def min-adj-label-aux-def Let-def
unfolding am-get-def cf-get-def l-get-def
apply (simp only: nres-monad-laws)
supply LFO-refine[OF fun-relD[OF AM 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 | v. cf-of f (u, v) ≠ 0 \}
using cfE-of-ss-VxV
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:
\[
\begin{aligned}
&\left[ (\langle x, cf \rangle, f) \in xf-rel, (ei, e) \in Id \times Id \right] \\
\Rightarrow &\\text{push2-aux } x \ cf \ ei \ \leq \ \Downarrow xf-rel \ (\text{push } f \ l \ e) \\
\text{unfolding} &\\text{push-def push2-aux-def} \\
\text{apply} &\text{refine-vcg} \\
\text{apply} &\ (\text{vc-solve simp: xf-rel-def no-self-loop}) \\
\text{subgoal for} &\ u \ v \\
\text{unfolding} &\\text{push-precond-def using cfE-of-ss-invE by auto} \\
\text{subgoal for} &\ u \ v \\
\text{proof} = &\ \\
\text{assume} &\ [\text{simp}]: \text{Labeling c s f l} \\
\text{then interpret} &\ \text{Labeling c s f l} \\
\text{assume} &\\text{push-precond } f \ l \ (u, v) \\
\text{then interpret} &\ l': \text{push-effect-locale c s f l u v by unfold-locales} \\
\text{show} &\ \text{thesis} \\
\text{apply} &\ (\text{safe intro!}: \text{ext}) \\
\text{using} &\ l'.excess'-if l'.\Delta-def l'.cf'-alt l'.uv-not-eq(1) \\
\text{by} &\ (\text{auto}) \\
\text{qed} \\
\text{done} \\
\end{aligned}
\]

definition push2 x cf ≡ \lambda (u,v). do 
\{
  assert \ (\langle u,v \rangle \in E \cup E^{-1}) \\
  xu ← x-get x u; \\
  cfuv ← cf-get cf (u,v); \\
  cfvu ← cf-get cf (v,u); \\
  let Δ = \min xu cfuv; \\
  x ← x-add x u (-Δ); \\
  x ← x-add x v Δ; \\
  cf ← cf-set cf (u,v) (cfuv - Δ); \\
  cf ← cf-set cf (v,u) (cfvu + Δ); \\

  return (x,cf)
\}

lemma push2-refine[refine]:
\begin{aligned}
&\text{assumes } \ (\langle x,cf,f \rangle, f) \in xf-rel \ (ei, e) \in Id \times Id \\
&\text{shows } \text{push2 } x \ cf \ ei \ \leq \ \Downarrow xf-rel \ (\text{push } f \ l \ e) \\
\text{proof} = &\ \\
\text{have } &\text{push2 } x \ cf \ ei \ \leq \ (\text{push2-aux } x \ cf \ ei) \\
\text{unfolding} &\text{push2-def push2-aux-def} \\
\text{unfolding} &\text{x-get-def x-add-def cf-get-def cf-set-def} \\
\text{unfolding} &\text{augment-edge-cf-def} \\
\text{apply} &\ (\text{simp only: nres-monad-laws}) \\
\text{apply} &\\text{refine-vcg}
\end{aligned}
6.2.6 Adding frequency counters to labeling

definition l-invar \( l \equiv \forall v. l\,v \neq 0 \rightarrow v \in V \)

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

definition clc-rel \( \equiv \text{br}\,\text{snd}\,\text{clc\text{-}invar} \)

definition clc-init C \( \equiv \text{do}\{l \leftarrow l\text{-init}\,C;\ cnt \leftarrow cnt\text{-init}\,C;\ \text{return}\,(cnt,l)\} \)

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

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

definition clc-has-gap \( \equiv \lambda (cnt,l)\ lu. \text{do}\{\text{nlu} \leftarrow cnt\text{-get}\ cnt\,lu;\ \text{return}\,(\text{nlu} = 0)\} \)

lemma cardV-le-N: \text{card}\,V \leq N \text{using}\ \text{card-mono}\,[\text{OF } V\text{-ss}] \text{by auto}

lemma N-not-Z: N \neq 0 \text{using}\ \text{card-V-ge2}\ \text{cardV-le-N}\ \text{by auto}

lemma N-ge-2: 2 \leq N \text{using}\ \text{card-V-ge2}\ \text{cardV-le-N}\ \text{by auto}

lemma clc-init-refine\{refine\}: 
assumes [simplified,simp]: (Ci,C)\in\text{nat-rel}
assumes [simp]: C = \text{card}\,V
shows clc-init\ Ci \leq \text{clc-rel} (l\text{-init}\,C)
proof 
have AUX: \{u. u \neq s \land u \in V\} = V\setminus\{s\} \text{by auto}

show \text{thesis}
unfolding $\text{clc-init-def} \ l\text{-init-def} \ \text{cnt-init-def}$
apply $\text{refine-vcg}$
unfolding $\text{clc-rel-def} \ \text{clc-invar-def}$
using $\text{cardV-le-N} \ N\text{-not-Z}$
by (auto simp: $\text{in-br-conv} \ V\text{-not-empty AUX} \ \text{l-invar-def}$)
qed

lemma $\text{clc-get-refine}$[refine]:
$$[(\text{clc}, l) \in \text{clc-rel}; (ui, u) \in \text{nat-rel}] \implies \text{clc-get} \ l \text{ ui} \leq Id (l\text{-get} \ l \ u)$$
unfolding $\text{clc-get-def} \ \text{clc-rel-def}$
by (auto simp: $\text{in-br-conv}$ split: prod.split)
definition $l\text{-get-rlx} :: (\text{node} \Rightarrow \text{nat}) \Rightarrow \text{node} \Rightarrow \text{nat res}$
where $l\text{-get-rlx} \ l \ u \equiv \{ 
\text{assert} (u < N); 
\text{return} (l \ u)
\}$
definition $\text{clc-get-rlx} \equiv \lambda (\text{cnt}, l) \ u. l\text{-get-rlx} \ l \ u$

lemma $\text{clc-get-rlx-refine}$[refine]:
$$[(\text{clc}, l) \in \text{clc-rel}; (ui, u) \in \text{nat-rel}] \implies \text{clc-get-rlx} \ l \ \text{clc ui} \leq Id (l\text{-get-rlx} \ l \ u)$$
unfolding $\text{clc-get-rlx-def} \ \text{clc-rel-def}$
by (auto simp: $\text{in-br-conv}$ split: prod.split)

lemma $\text{card-insert-disjointI}$:
$$[\text{finite} \ Y; X = \text{insert} \ x \ Y; x \notin Y] \implies \text{card} \ X = \text{Suc} (\text{card} \ Y)$$
by auto

lemma $\text{clc-set-refine}$[refine]:
$$[(\text{clc}, l) \in \text{clc-rel}; (ui, u) \in \text{nat-rel}; (ai, a) \in \text{nat-rel}] \implies \text{clc-set} \ l \text{ ui ai} \leq Id (l\text{-set} \ l \ u \ a)$$
unfolding $\text{clc-set-def} \ \text{l-set-def} \ \text{l-get-def} \ \text{cnt-decr-def} \ \text{cnt-incr-def}$
apply $\text{refine-vcg}$
apply $\text{vc-solve}$
unfolding $\text{clc-rel-def} \ \text{in-br-conv} \ \text{clc-invar-def} \ \text{l-invar-def}$
subgoal using $\text{cardV-le-N}$ by auto
applyS auto
applyS (auto simp: simp: $\text{card-gt-0-iff}$)

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

subgoal
  by (auto intro!: arg-cong[where \(f=\text{card}\)])
done

done

\textbf{lemma} \textit{clc-has-gap-correct}[\textsc{THEN} order-trans, \textit{{refine-vcg}}]:

\[
[(\text{clc},l)\in\text{clc-rel}; k<2\ast N] \implies \text{clc-has-gap clc} k \leq (\text{spec} r. r \leftrightarrow \text{gap-precond} l k)
\]

unfolding \textit{clc-has-gap-def} \textit{cnt-get-def} \textit{gap-precond-def}
apply \textit{{refine-vcg}}
unfolding \textit{clc-rel-def} \textit{clc-invar-def} \textit{in-br-conv}
by auto

\textbf{6.2.7 Refinement of Gap-Heuristics}

Utilities to Implement Gap-Heuristics

definition \textit{gap-aux} \(C\ l\ k \equiv\) do 
  nfoldli \([0..<N]\) (\(\lambda\). True) (\(\lambda\nu\ l.\) do 
    lv \(\leftarrow\) \text{\textit{l-get-rlx}} l v;
    if \((k < lv \land lv < C)\) then do 
      assert \((C+1 < 2\ast N)\);
      l \(\leftarrow\) \text{\textit{l-set}} l v \((C+1)\);
    return l
  } else return l
}) l

\textbf{lemma} \textit{gap-effect-invar}[\textsc{simp}]: \textit{l-invar} \(l\implies l\text{-invar} (\text{gap-effect} l k)\)
unfolding \textit{gap-effect-def} \textit{l-invar-def}
by auto

\textbf{lemma} \textit{relabel-effect-invar}[\textsc{simp}]: \([\textit{l-invar} l; u\in V]\implies l\text{-invar} (\text{relabel-effect} f l u)\)
unfolding \textit{relabel-effect-def} \textit{l-invar-def}
by auto

\textbf{lemma} \textit{gap-aux-correct}[\textsc{THEN} order-trans, \textit{{refine-vcg}}]:

\[
[l\text{-invar} l; C=\text{card} V] \implies \text{gap-aux} C l k \leq \text{SPEC} (\lambda r. r=\text{gap-effect} l k)
\]

unfolding \textit{gap-aux-def} \textit{l-get-rlx-def} \textit{l-set-def}
apply \textit{(simp only; nres-mad-laws)}
apply \textit{(refine-vcg \textit{nfoldli-rule}[where \(I = \lambda it1 it2 l'.\forall u.\) if \(u\in set\) \(it2\) then \(l'\ u =\) \(l u\) else \(l'\ u = \text{gap-effect} l k u\)])}
apply \textit{(vc-solve simp: upt-eq-lel-conv)}

subgoal
  apply \textit{(frule gap-effect-invar[where \(k=k\)])}
  unfolding \textit{l-invar-def} using \textit{V-ss} by \textit{force}
subgoal using \textit{N-not-Z cardV-le-N} by auto
subgoal unfolding \textit{l-invar-def} by auto
subgoal unfolding \textit{gap-effect-def} by auto

81
subgoal for \( v' \)

apply (drule spec[where \( x=u \)])

by (auto split: if-splits simp: gap-effect-def)

subgoal by auto
done

definition gap2 \( C \) clc \( k \) ≡ do {
  nfoldli \([0..<N]\) (λ- True) (λv clc. do {
    lv ← clc-get-rlx clc v;
    if (k < lv ∧ lv < C) then do {
      clc ← clc-set clc v (C+1);
      return clc
    } else return clc
  }) clc
}

lemma gap2-refine[refine]:

assumes \([\text{simplified.simp}]\): (\( C_i,C \))\( \in \) nat-rel \( (k,i)\)\( \in \) nat-rel

assumes CLC: (clc,\( l \))\( \in \) clc-rel

shows gap2 \( C_i \) clc ki ≤⇓ clc-rel (gap-aux \( C \) \( l \) \( k \))

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 \( C \) \( l \) \( u \)
  else return l
}

lemma gap-relabel-aux-refine:

assumes \([\text{simp}]\): \( C = \text{card} \) \( V \) \( l\)-invar \( l \)

shows gap-relabel-aux \( C \) \( f \) \( l \) \( u \) ≤ gap-relabel \( f \) \( l \) \( u \)

unfolding gap-relabel-aux-def gap-relabel-def relabel-def

 gap-relabel-effect-def l-get-def

apply (simp only: Let-def nres-monad-laws)

apply refine-vcg

by auto

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

definition clc-relabel2 am of clc \( u \) ≡ do {
assert \( u \in V - \{s,t\} \);
\( nl \leftarrow \text{min-adj-label-clc} \ am \ cf \ clc \ u \);
\( clc \leftarrow \text{clc-set} \ clc \ u \ (nl+1) \);
return \( clc \)
}

\textbf{lemma} clc-relabel2-refine[refine]:
\textbf{assumes} \( XF: ((x,cf),f) \in xf-rel \)
\textbf{assumes} \( CLC: (clc,l) \in clc-rel \)
\textbf{assumes} \( AM: (am,\text{adjacent-nodes}) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{list-set-rel} \)
\textbf{assumes} \[ \text{[simplified,simp]}: (ui,u) \in Id \]
\textbf{shows} \( \text{clc-relabel2} \ am \ cf \ clc \ ui \leq \ll clc-rel \ (\text{relabel} f l u) \)

\textbf{proof} –
\textbf{have} \( \text{clc-relabel2} \ am \ cf \ clc \ ui \leq \ll clc-rel \ (\text{relabel2} am \ cf \ l \ ui) \)
\textbf{unfolding} \( \text{clc-relabel2-def} \ \text{relabel2-def} \)
\textbf{apply} \( \text{refine-rcg} \)
\textbf{apply} \( \text{refine-dref-type} \)
\textbf{apply} \( \text{vc-solve simp: CLC} \)
\textbf{subgoal}
\textbf{using} \( CLC \)
\textbf{unfolding} \( \text{clc-rel-def} \ \text{in-br-conv} \ \text{min-adj-label-clc-def} \)
\textbf{by} \( \text{(auto split: prod.split)} \)
\textbf{done}
\textbf{also note} \( \text{relabel2-refine[OF XF AM, of l l ui u]} \)
\textbf{finally show} \( \text{\textit{thesis by simp}} \)
\textbf{qed}

\textbf{definition} gap-relabel2 \ C \ am \ cf \ clc \ u \equiv \{ \)
\( lu \leftarrow \text{clc-get} \ clc \ u \);
\( clc \leftarrow \text{clc-relabel2} \ am \ cf \ clc \ u \);
\( \text{has-gap} \leftarrow \text{clc-has-gap} \ clc \ lu \);
\textbf{if} \( \text{has-gap} \ \text{then} \text{gap2} \ C \ clc \ lu \)
\textbf{else}
\( \text{RETURN} \ clc \)
\}

\textbf{lemma} gap-relabel2-refine-aux:
\textbf{assumes} \( XCF: ((x,cf),f) \in xf-rel \)
\textbf{assumes} \( CLC: (clc,l) \in clc-rel \)
\textbf{assumes} \( AM: (am,\text{adjacent-nodes}) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{list-set-rel} \)
\textbf{assumes} \[ \text{[simplified,simp]}: (Ci,C) \in Id \quad (ui,u) \in Id \]
\textbf{shows} \( \text{gap-relabel2} \ Ci \ am \ cf \ clc \ ui \leq \ll clc-rel \ (\text{gap-relabel-aux} \ C \ f \ l \ u) \)
\textbf{unfolding} \( \text{gap-relabel2-def} \ \text{gap-relabel-aux-def} \)
\textbf{apply} \( \text{refine-ecg} XCF AM CLC \ \text{if-bind-cond-refine bind-refine'} \)
\textbf{apply} \( \text{vc-solve solve: refl} \)
\textbf{subgoal for} \( - \ lu \)
\textbf{using} \( CLC \)
\textbf{unfolding} \( \text{clc-get-def} \ \text{l-get-def} \ \text{clc-rel-def} \ \text{in-br-conv} \ \text{clc-invar-def} \)

83
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: \((clc, l) \in clc-rel\)
assumes AM: \((am, adjacent-nodes) \in nat-rel \rightarrow \langle nat-rel \rangle \in \text{list-rel}\)
assumes [simplified, simp]: \((ui, u) \in Id\)
assumes CC: \(C = \text{card } V\)
shows gap-relabel2 \(C am cf clc ui \leq \text{clc-rel} (gap-relabel f l u)\)
proof –
from CLC have LINV: \(l-invar l\) unfolding clc-rel-def in-br-conv clc-invar-def
by auto

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

6.3 Refinement to Efficient Data Structures

6.3.1 Registration of Abstract Operations

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

context includes Network-Impl-Sepref-Register

begin
sepref-register x-get x-add

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

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

sepref-register cnt-init cnt-get cnt-incr cnt-decr

sepref-register gap2 min-adj-label min-adj-label-clc

sepref-register push2 relabel2 clc-relabel2 gap-relabel2

sepref-register pp-init-xcf2

end — Anonymous Context

6.3.2 Excess by Array

definition x-assn \(\equiv \text{is-nf } N (0::\text{capacity-impl})\)

lemma x-init-hnr[sepref-fr-rules]:
\((uncurry0 \ (Array.new N 0), \text{unit-assn} \rightarrow x-assn) \in x-assn\)

apply sepref-to-hoare unfolding x-assn-def x-init-def
by (sep-auto heap: nf-init-rule)

**Lemma** \( x\text{-get-hnr}[\text{sepref-fr-rules}]: \)

\[
(\text{uncurry Array.nth, un curry (PR-CONST x\text{-get})}) \\
\in x\text{-assn}^k * a \text{ node-assn}^k \rightarrow_a \text{ cap-assn}
\]

**Apply** sepref-to-hoare

**Unfolding** \( x\text{-assn-def} x\text{-get-def} \) by (sep-auto simp: refine-pw-simps)

**Definition** (in −) \( x\text{-add-impl} xu \Delta \equiv \{ \)

\[
xu \leftarrow \text{Array.nth} x u; \\
xu \leftarrow \text{Array.upd} u (xu + \Delta); \\
\text{return} x
\]

**Lemma** \( x\text{-add-hnr}[\text{sepref-fr-rules}]: \)

\[
(\text{uncurry2 x\text{-add-impl, un curry2 (PR-CONST x\text{-add})})} \\
\in x\text{-assn}^d * a \text{ node-assn}^k * a \text{ cap-assn}^k \rightarrow_a x\text{-assn}
\]

**Apply** sepref-to-hoare

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

**Definition** (in −) \( l\text{-init-impl} N s \text{ cardV} \equiv \{ \)

\[
l \leftarrow \text{Array.new} N (0::\text{nat}); \\
l \leftarrow \text{Array.upd} s \text{ cardV} l; \\
\text{return} l
\]

**Lemma** \( l\text{-init-hnr}[\text{sepref-fr-rules}]: \)

\[
(l\text{-init-impl} N s, (\text{PR-CONST l\text{-init}})) \in \text{nat-assn}^k \rightarrow_a l\text{-assn}
\]

**Apply** sepref-to-hoare

**Unfolding** \( l\text{-assn-def} l\text{-init-def} l\text{-init-impl-def} \)

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

**Lemma** \( l\text{-get-hnr}[\text{sepref-fr-rules}]: \)

\[
(\text{uncurry Array.nth, un curry (PR-CONST l\text{-get})}) \\
\in l\text{-assn}^k * a \text{ node-assn}^k \rightarrow_a \text{ nat-assn}
\]

**Apply** sepref-to-hoare

**Unfolding** \( l\text{-assn-def} l\text{-get-def} \) by (sep-auto simp: refine-pw-simps)

**Lemma** \( l\text{-get-rlx-hnr}[\text{sepref-fr-rules}]: \)

\[
(\text{uncurry Array.nth, un curry (PR-CONST l\text{-get-rlx})}) \\
\in l\text{-assn}^k * a \text{ node-assn}^k \rightarrow_a \text{ nat-assn}
\]

**Apply** sepref-to-hoare

**Unfolding** \( l\text{-assn-def} l\text{-get-rlx-def} \) by (sep-auto simp: refine-pw-simps)

**Lemma** \( l\text{-set-hnr}[\text{sepref-fr-rules}]: \)

\[
(\text{uncurry2 (\lambda i x. Array.upd i x a), un curry2 (PR-CONST l\text{-set})})
\]

85
\[ \in l\text{-assn}^d \ast_a \node\text{-assn}^k \ast_a \text{nat-assn}^k \rightarrow_a l\text{-assn} \]

apply sepref-to-hoare

unfolding l-assn-def l-set-def

by (sep-auto simp: refine-pw-simps split: prod.split)

\subsection*{6.3.4 Label Frequency by Array}

definition cnt-assn \((f::\node\Rightarrow \text{nat})\) \(a\)
\[ \equiv \exists \_A \_l. \_a \rightarrow \_l \ast \left( (\text{length} \_l = 2 \ast \_N \wedge (\forall \_i < 2 \ast \_N. \_l i = f \_i) \wedge (\forall \_i \geq 2 \ast \_N. f \_i = 0) \right) \]

definition (in −) cnt-init-impl \(N C \equiv \{\)
\(a \leftarrow \text{Array}.\text{new} \ (2 \ast \_N \ (0::\text{nat});
\(a \leftarrow \text{Array}.\text{upd} \ 0 \ (C-1) \ a;
\(a \leftarrow \text{Array}.\text{upd} \ C \ 1 \ a;
\) return \(a\) \}

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

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

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

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

apply sepref \text{-} to \text{-} hoare

unfolding cnt \text{-} decr \text{-} def cnt \text{-} decr \text{-} impl \text{-} def cnt \text{-} assn \text{-} def

by ( sep \text{-} auto simp: refine \text{-} pw \text{-} simps )

6.3.5 Combined Frequency Count and Labeling

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

sepref \text{-} thm clc \text{-} init \text{-} impl is PR \text{-} CONST clc \text{-} init :: nat \text{-} assn_k \rightarrow_a clc \text{-} assn

unfolding clc \text{-} init \text{-} def PR \text{-} CONST \text{-} def clc \text{-} assn \text{-} def

by sepref

concrete \text{-} definition ( in \rightarrow ) clc \text{-} init \text{-} impl

uses Network \text{-} Impl.clc \text{-} init \text{-} impl.refine \text{-} raw

lemmas [ sepref \text{-} fr \text{-} rules ] = clc \text{-} init \text{-} impl.refine[ OF Network \text{-} Impl.axioms ]

sepref \text{-} thm clc \text{-} get \text{-} impl is uncurry ( PR \text{-} CONST clc \text{-} get )

:: clc \text{-} assn_k \ast_a node \text{-} assn_k \rightarrow_a nat \text{-} assn

unfolding clc \text{-} get \text{-} def PR \text{-} CONST \text{-} def clc \text{-} assn \text{-} def

by sepref

concrete \text{-} definition ( in \rightarrow ) clc \text{-} get \text{-} impl

uses Network \text{-} Impl.clc \text{-} get \text{-} impl.refine \text{-} raw is ( uncurry \ ?f,\cdot\rightarrow ) \in -

lemmas [ sepref \text{-} fr \text{-} rules ] = clc \text{-} get \text{-} impl.refine[ OF Network \text{-} Impl.axioms ]

sepref \text{-} thm clc \text{-} get \text{-} rlx \text{-} impl is uncurry ( PR \text{-} CONST clc \text{-} get \text{-} rlx )

:: clc \text{-} assn_k \ast_a node \text{-} assn_k \rightarrow_a nat \text{-} assn

unfolding clc \text{-} get \text{-} rlx \text{-} def PR \text{-} CONST \text{-} def clc \text{-} assn \text{-} def

by sepref

concrete \text{-} definition ( in \rightarrow ) clc \text{-} get \text{-} rlx \text{-} impl

uses Network \text{-} Impl.clc \text{-} get \text{-} rlx \text{-} impl.refine \text{-} raw is ( uncurry \ ?f,\cdot\rightarrow ) \in -

lemmas [ sepref \text{-} fr \text{-} rules ] = clc \text{-} get \text{-} rlx \text{-} impl.refine[ OF Network \text{-} Impl.axioms ]

sepref \text{-} thm clc \text{-} set \text{-} impl is uncurry2 ( PR \text{-} CONST clc \text{-} set )

:: clc \text{-} assn \ast_a node \text{-} assn_k \ast_a nat \text{-} assn_k \rightarrow_a clc \text{-} assn

unfolding clc \text{-} set \text{-} def PR \text{-} CONST \text{-} def clc \text{-} assn \text{-} def

by sepref

concrete \text{-} definition ( in \rightarrow ) clc \text{-} set \text{-} impl

uses Network \text{-} Impl.clc \text{-} set \text{-} impl.refine \text{-} raw is ( uncurry2 \ ?f,\cdot\rightarrow ) \in -

lemmas [ sepref \text{-} fr \text{-} rules ] = clc \text{-} set \text{-} impl.refine[ OF Network \text{-} Impl.axioms ]

sepref \text{-} thm clc \text{-} has \text{-} gap \text{-} impl is uncurry ( PR \text{-} CONST clc \text{-} has \text{-} gap )

:: clc \text{-} assn_k \ast_a nat \text{-} assn_k \rightarrow_a bool \text{-} assn

unfolding clc \text{-} has \text{-} gap \text{-} def PR \text{-} CONST \text{-} def clc \text{-} assn \text{-} def

by sepref

concrete \text{-} definition ( in \rightarrow ) clc \text{-} has \text{-} gap \text{-} impl

uses Network \text{-} Impl.clc \text{-} has \text{-} gap \text{-} impl.refine \text{-} raw is ( uncurry \ ?f,\cdot\rightarrow ) \in -

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

**sepref-thm** *push-impl* is uncurry2 (PR-CONST push2)
:: x-assn\(_d\) *\(_a\) cf-assn\(_d\) *\(_a\) edge-assn\(_k\) \(\to\) \(x\)-assn\(_x\) *\(_a\) cf-assn

**unfolding** *push2-def* PR-CONST-def

**by** sepref

**concrete-definition** (in \(\neg\)) *push-impl*

**uses** Network-Impl.push-impl.refine-raw is (uncurry2 \(?f, ?.\)\(\in\)-

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

6.3.7 Relabel

**sepref-thm** *min-adj-label-impl* is uncurry3 (PR-CONST min-adj-label)
:: am-assn\(_k\) *\(_a\) cf-assn\(_k\) *\(_a\) l-assn\(_k\) *\(_a\) node-assn\(_k\) \(\to\) nat-assn

**unfolding** *min-adj-label-def* PR-CONST-def

**by** sepref

**concrete-definition** (in \(\neg\)) *min-adj-label-impl*

**uses** Network-Impl.min-adj-label-impl.refine-raw is (uncurry3 \(?f, ?.\)\(\in\)-

**lemmas** [sepref-fr-rules] = min-adj-label-impl.refine[OF Network-Impl-axioms]

**sepref-thm** *relabel-impl* is uncurry3 (PR-CONST relabel2)
:: am-assn\(_k\) *\(_a\) cf-assn\(_k\) *\(_a\) l-assn\(_d\) *\(_a\) node-assn\(_k\) \(\to\) l-assn

**unfolding** *relabel2-def* PR-CONST-def

**by** sepref

**concrete-definition** (in \(\neg\)) *relabel-impl*

**uses** Network-Impl.relabel-impl.refine-raw is (uncurry3 \(?f, ?.\)\(\in\)-

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

6.3.8 Gap-Relabel

**sepref-thm** *gap-impl* is uncurry2 (PR-CONST gap2)
:: nat-assn\(_k\) *\(_a\) cf-assn\(_k\) *\(_a\) l-assn\(_d\) *\(_a\) node-assn\(_k\) \(\to\) nat-assn

**unfolding** *gap2-def* PR-CONST-def

**by** sepref

**concrete-definition** (in \(\neg\)) *gap-impl*

**uses** Network-Impl.gap-impl.refine-raw is (uncurry2 \(?f, ?.\)\(\in\)-

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

**sepref-thm** *min-adj-label-clc-impl* is uncurry3 (PR-CONST min-adj-label-clc)
:: am-assn\(_k\) *\(_a\) cf-assn\(_k\) *\(_a\) clc-assn\(_k\) *\(_a\) node-assn\(_k\) \(\to\) nat-assn

**unfolding** *min-adj-label-clc-def* PR-CONST-def clc-assn-def

**by** sepref

**concrete-definition** (in \(\neg\)) *min-adj-label-clc-impl*

**uses** Network-Impl.min-adj-label-clc-impl.refine-raw is (uncurry3 \(?f, ?.\)\(\in\)-

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

**sepref-thm** *clc-relabel-impl* is uncurry3 (PR-CONST clc-relabel2)
:: am-assn\(_k\) *\(_a\) cf-assn\(_k\) *\(_a\) clc-assn\(_d\) *\(_a\) node-assn\(_k\) \(\to\) 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-raw is (uncurry3 f,−)∈-

sepref-thm gap-relabel-impl is uncurry4 (PR-CONST gap-relabel2)
:: nat-assn+k ∗a am-assn+k ∗a cf-assn+k ∗a clc-assn+d ∗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-raw 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 ∗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 (?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 × r; (Qi,Q) ∈ Id]
⇒ fifo-push2-aux x cf Qi ei ≤ ≡ (xf-rel × r, Id) (fifo-push f l Q v)
unfolding fifo-push-def fifo-push2-aux-def
apply refine-vcg
apply (vc-solve simp: xf-rel-def no-self-loop)
subgoal for \( u \) \( v \)
  unfolding push-precond-def using cfE-of-ss-invE by auto
subgoal for \( u \) \( v \)
proof
proof
Proof

lemma fifo-push2-refine[refine]:
assumes \((x,cf,f)\)\(\in\)xf-rel \( (ei,e)\)\(\in\)Id \(x,Id \ (Qi,Q)\)\(\in\)Id
shows fifo-push2 \( x \) \( cf \) \( Qi \) \( ei \) \( \leq \) \( (xf-rel \times, \text{Id}) \) \( (fifo-push f l Q \ c) \)
proof
have fifo-push2 \( x \) \( cf \) \( Qi \) \( ei \) \( \leq \) \( (fifo-push2-aux \ x \ cf \ Qi \ ei) \)
  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

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

  \( cf \) \( \leftarrow \) \( cf\text{-set} \ (u,v) \) \( (cfuw - \Delta) \);
  \( cf \) \( \leftarrow \) \( cf\text{-set} \ (v,u) \) \( (cfvu + \Delta) \);

  if \((v\neq s \land v\neq t \land xv = 0)\) then do \{ 
    \( Q \) \( \leftarrow \) q-enqueue \( v \) \( Q \);
    return \((x,cf),Q\)
  \} else 
  return \((x,cf),Q\)
\}

qed

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

7.2.2 Refinement of Gap-Relabel

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

lemma fifo-gap-relabel-aux-refine:
assumes [simp]: C = card V  l-invar l
shows fifo-gap-relabel-aux C f l Q u ≤ fifo-gap-relabel f l Q u
unfolding fifo-gap-relabel-aux-def fifo-gap-relabel-def relabel-def
gap-relabel-effect-def l-get-def q-enqueue-def
apply (simp only: Let-def nres-monad-laws)
apply refine-vcg
by auto

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

lemma fifo-gap-relabel2-refine-aux:
assumes XCF: ((x, cf), f) ∈ xf-rel
assumes CLC: (clc,l)∈clc-rel
assumes AM: (am, adjacent-nodes)∈nat-rel→(nat-rel)list-set-rel
assumes [simplified,simp]: (Ci,C)∈Id   (Qi,Q)∈Id   (ui,u)∈Id
shows fifo-gap-relabel2 Ci am cf clc Qi ui ≤ ⟨Id×Id, Id⟩ (fifo-gap-relabel-aux C f l Q u)
unfolding fifo-gap-relabel2-def fifo-gap-relabel-aux-def
apply (refine-vcg XCF AM CLC if-bind-cond-refine bind-refine')

92
apply refine-d-refine-type
apply \(\text{vc-solve solve: refl}\)
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: \((\text{clc,l})\in\text{clc-rel}\)
assumes AM: \((\text{am,adjacent-nodes})\in\text{nat-rel}\rightarrow\text{nat-rel}\rightarrow\text{list-set-rel}\)
assumes [simplified,simp]: \((Q_i,Q)\in Id (ui,u)\in Id\)
assumes CC: \(C = \text{card } V\)
shows \(\text{fifo-gap-relabel2 } C \text{ am cf clc } Q_i u_i \leq \Downarrow (\text{clc-rel } \times_r \text{ Id}) (\text{fifo-gap-relabel } f l Q u)\)
proof
  from CLC have LINV: \(l\text{-invar } l\)
  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 \(\{v . (u,v)\in cfE-of f\} = \text{set (filter } (\lambda v. cf-of f (u,v) \neq 0) (am u))\)
  using assms cfE-of-ss-invE
  unfolding is-adj-map-def Graph.E-def
  by fastforce

private lemma adjacent-nodes-iterate-refine1:
fixes \(ff u f\)
assumes AMR: \((am,\text{adjacent-nodes})\in Id \rightarrow \{Id\}\text{-list-set-rel}\)
assumes CR: \(\Lambda s si. (si,s)\in Id \implies cci si \iff cc s\)
assumes FR: \(\forall vi si s si [\{vi,v\}\in Id; v\in V; (u,v)\in E\cup E^{-1}; (si,s)\in Id] \implies fff vi si \leq \Downarrow Id\ (\text{do } \{\}
  \text{ if } (cf-of f (u,v) \neq 0) \text{ then } ff v s \text{ else return } s
  \}) (\text{is } \Lambda v vi s si. [\vdots;\vdots;\vdots;\vdots] \implies - \leq \Downarrow (\text{?ff' v s})\)
assumes S0R: \((s0i,s0)\in Id\)
assumes UR: \((ui,u)\in Id\)
shows nfoldli (am ui) cci ffi s0i
      ≤⇓Id (FOREACHc {v . (u,v)∈cfE-of f} cc ff s0)

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

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

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

private definition dis-loop-aux am f_0 l Q u ≡ do {
  assert (u∈V −{s,t});
  assert (distinct (am u));
nfoldli (am u) (λ(f,l,Q). excess f u ≠ 0) (λv (f,l,Q). do {
      assert ((u,v)∈E∪E−1 ∧ v∈V);
      if (cf-of f_0 (u,v) ≠ 0) then do {
        if (l u = l v + 1) then do {
          (f',l,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)
    }) else return (f,l,Q)
  }
}

94
private definition fifo-discharge-aux am \( f_0 \) \( l \) \( Q \) \( \equiv \) do {
  \((u, Q) \leftarrow q\text{-dequeue } Q;\)
  \assert (u \in V \land u \neq s \land u \neq t);\)

  \((f, l, Q) \leftarrow \text{dis-loop-aux am } f_0 \) \( l \) \( Q \);\)
  
  if excess \( f u \neq 0 \) then do {
    \((l, Q) \leftarrow \text{fifo-gap-relabel } f l Q u;\)
    \return \((f, l, Q)\);\)
  } else do {
    \return \((f, l, Q)\);\)

  \}
}

private lemma fifo-discharge-aux-refine:
\asserts AM: \((am, adjacent-nodes) \in Id \rightarrow (Id)\text{-list-set-rel}\)
\asserts \[(simplified, simp): (f, f) \in Id \quad (li, l) \in Id \quad (Qi, Q) \in Id\]
\shows fifo-discharge-aux am \( f_0 \) \( l \) \( Q \) \( u \) \( \leq \Downarrow Id \) (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 \( f_0 \) \( l \) \( Q \) \( u \) \( \equiv \) do {
  \assert (u \in V \setminus \{s, t\});
  \assert (distinct (am u));
  nfoldli (am u) (λ(f,l, Q). excess f u \( \neq \) 0) (λv (f,l, Q). do {
    \assert ((u,v) \in E \cup E^{-1} \land v \in V);\)
    \if (cf-of f (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 (\( \forall v'. v' \neq v \rightarrow cf\text{-of } f' (u,v') = cf\text{-of } f (u,v')\));
        \return \((f',l, Q)\);\)
      } else return \((f,l, Q)\);\)
    } else \return \((f,l, Q)\);\)
  }) \((f_0,l, Q)\)
}

private lemma dis-loop-aux2-refine:
\shows dis-loop-aux2 am \( f_0 \) \( l \) \( Q \) \( u \) \( \leq \Downarrow Id \) (dis-loop-aux am \( f_0 \) \( l \) \( Q \) \( u \))
unfolding dis-loop-aux2-def dis-loop-aux-def
apply (intro ASSERT-refine-right ASSERT-refine-left; assumption?)
apply (rule nfoldli-invar-refine[where
I=λit1 it2 (f, -, -). ∀ v ∈ set it2. cf-of f (u, v) = cf-of f0 (u, v)])
apply refine-dref-type
apply vc-solve
apply (auto simp: pw-leof-iff refine-pw-simps fifo-push-def; metis)
done

private definition dis-loop-aux3 am x cf l Q u ≡ do {
  assert (u ∈ V ∧ distinct (am u));
  monadic-nfoldli (am u)
  \(\lambda (x, cf), l, Q). \{ xu ← x-get x u; return (xu ≠ 0) \}'
  \(\lambda v ((x, cf), l, Q).\ do \{ cfuv ← cf-get cf (u, v);
    if (cfuv ≠ 0) then do {
      lu ← l-get l u;
      lv ← l-get l v;
      if (lu = lv + 1) then do {
        \((x, cf), l, Q) ← fifo-push2 x cf Q (u, v);
        return ((x, cf), l, Q)
      } else return ((x, cf), l, Q)
    } else return ((x, cf), l, Q)
\}\ ((x, cf), l, Q)
}

private lemma dis-loop-aux3-refine:
assumes [simplified, simp]: (ami, am) ∈ Id (li, l) ∈ Id (Qi, Q) ∈ Id (ui, u) ∈ Id
assumes XF: \((x, cf), f) ∈ xf-rel
shows \(\land (xf-rel × r Id × r Id) \land dis-loop-aux2 am f l Q u\)
unfolding dis-loop-aux3-def dis-loop-aux2-def
unfolding x-get-def cf-get-def l-get-def
apply (simp only: nres-monad-laws nfoldli-to-monadic)
apply (refine-rcg)
apply refine-dref-type
using XF
by (vc-solve simp: xf-rel-def in-br-conv)

definition dis-loop2 am x cf clc Q u ≡ do {
  assert (distinct (am u));
amu ← am-get am u;
  monadic-nfoldliamu
  \(\lambda ((x, cf), clc, Q). \{ xu ← x-get x u; return (xu ≠ 0) \}'
  \(\lambda v ((x, cf), clc, Q).\ do \{ cfuv ← cf-get cf (u, v);
    if (cfuv ≠ 0) then do {
      lu ← clc-get clc u;
      lv ← clc-get clc v;
\}\ ((x, cf), clc, Q)

if \((lu = lv + 1)\) then do 
\[
((x, cf), Q) \leftarrow \text{fifo-push2} x cf Q (u, v);
\]
return \(((x, cf), clc, Q)\)
} else return \(((x, cf), clc, Q)\)
} else return \(((x, cf), clc, Q)\)

private lemma dis-loop2-refine-aux:
assumes \([\text{simplified}, \text{simp}]: (xi, x) \in Id \ (cfi, cf) \in Id \ (ami, am) \in Id\)
assumes \([\text{simplified}, \text{simp}]: (li, l) \in Id \ (Qi, Q) \in Id \ (ui, u) \in Id\)
assumes CLC: \((clc, l) \in clc-rel\)
shows dis-loop2 ami xi cfi clc Qi ui
\[
\leq \downarrow (Id \times_r clc-rel \times_r Id) \ (\text{dis-loop-aux3} \ am \ x \ cf \ l \ Q \ u)
\]
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: \(((x, cf), f) \in zf-rel\)
assumes CLC: \((clc, l) \in clc-rel\)
assumes \([\text{simplified}, \text{simp}]: (ami, am) \in Id \ (Qi, Q) \in Id \ (ui, u) \in Id\)
shows dis-loop2 ami x cf clc Qi ui
\[
\leq \downarrow (zf-rel \times_r clc-rel \times_r Id) \ (\text{dis-loop-aux} \ am \ f \ l \ Q \ u)
\]
proof –
have [simp]:
\[
((Id \times_r clc-rel \times_r Id) \ O (zf-rel \times_r Id)) = zf-rel \times_r clc-rel \times_r Id
\]
by (auto simp: prod-rel-comp)

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

definition fifo-discharge2 C am x cf clc Q \equiv do 
\[
(u, Q) \leftarrow \text{q-dequeue} Q;
\]
assert \((u \in V \land u \neq s \land u \neq t)\);
\[
((x, cf), clc, Q) \leftarrow \text{dis-loop2} \ am \ x \ cf \ clc \ Q \ u;
\]
xu \leftarrow \text{x-get} x \ u;
if xu \neq 0 then do 
\[
(clc, Q) \leftarrow \text{fifo-gap-relabel2} \ C \ am \ cf \ clc \ Q \ u;
\]
return ((x, cf), clc, Q)
} else do {
  return ((x, cf), clc, Q)
}

lemma fifo-discharge2-refine[refine]:
assumes AM: (am, adjacent-nodes) ∈ nat-rel → ⟨nat-rel⟩ list-set-rel
assumes XCF: ((x, cf), f) ∈ xf-rel
assumes CLC: (clc, l) ∈ clc-rel
assumes simplified, simp: (Qi, Q) ∈ Id
assumes CC: C = card V
shows fifo-discharge2 C am x cf clc Qi
  ≤⇓ (xf-rel × 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: vres-monad-laws)
    apply (refine-vcg XCF CLC AM IdI)
    apply (vc-solve simp: CC)
    subgoal unfolding xf-rel-def in-br-conv by auto
    applyS assumption
    done
  also note fifo-discharge-aux-refine[OF AM IdI IdI IdI]
  finally show thesis by simp
qed

end — Anonymous Context

7.2.4 Computing the Initial Queue

definition q-init am ≡ do {
  Q ← q-empty;
  ams ← am-get am s;
  nfoldli ams (λ-. True) (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 \( q\text{-init \ am} \leq \text{RETURN} \ (\text{filter} \ (\neq) \ t) \ (\text{am s}) \)

unfolding \( q\text{-init-def} \ q\text{-empty-def} \ q\text{-enqueue-def} \ am\text{-get-def} \)
apply (refine-vcg nfoldli-rule[where \( I = l \ 1 \ l = \text{filter} \ (\neq) \ t \ 1 \)])
by auto
also have \dots
\[
\leq (\text{spec} \ l \ \text{distinct} \ l \ \land \ \text{set} \ l = \{v \in V - \{s, t\}. \ \text{excess} \ pp\text{-init-f} \ v \neq 0})
\]
proof –
from \( am\text{-to-adj-nodes-refine}[OF \ AM] \)
have [simp]: \( \text{distinct} \ (\text{am s}) \ \land \ \text{set} \ (\text{am s}) = \text{adjacent-nodes} \ s \)
unfolding list-set-rel-def
by (auto simp: in-br-conv)
show \(?thesis"
using E-ss-VxV
apply (auto simp: pp-init-x-def adjacent-nodes-def)
unfolding Graph.E-def by auto
qed
finally show \(?thesis .
qed

7.2.5 Refining the Main Algorithm

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

lemma \( \text{fifo-push-relabel-aux-refine}: \)
assumes \( \text{AM: is-adj-map \ am} \)
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 l-init-def pp-init-l-def q-is-empty-def bind-to-let-conv
apply (rule specify-left[OF init-C-correct[OF AM]])
apply (refine-rg 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-reg)
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
7.3 Separating out the Initialization of the Adjacency Matrix

context Network-Impl

begin

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

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

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

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

definition fifo-push-relabel-run2 am cf ≡ do {
  cardV ← init-C am;
  (x,cf) ← pp-init-xcf2' am cf;
  clc ← clc-init cardV;
  Q ← q-init am;

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

  return cf
}

lemma fifo-push-relabel2-alt:
  fifo-push-relabel2 am = do {
    cf ← fifo-push-relabel-init2;
    fifo-push-relabel-run2 am cf
  }

unfolding fifo-push-relabel-init2-def fifo-push-relabel-run2-def
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 ≡ \( \lambda (L,R). L@\text{rev } R \)
definition (in −) q-empty-impl ≡ ([],[])
definition (in −) q-is-empty-impl ≡ \( \lambda (L,R). \text{is-Nil } L \land \text{is-Nil } R \)
definition (in −) q-enqueue-impl ≡ \( \lambda x \ (L,R). (L,x\#R) \)
definition (in −) q-dequeue-impl
≡ \( \lambda (x\#L,R) \Rightarrow (x,(L,R)) \| ([],R) \Rightarrow \text{case rev } R \text{ of } (x\#L) \Rightarrow (x,(L,[])) \)

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 \( q\text{-is-empty-impl-correct} \): \( q\text{-is-empty-impl} \ Q \iff q\alpha \ Q = [] \)
unfolding \( q\alpha\text{-def} q\text{-is-empty-impl-def} \)
by (cases \( Q \)) (auto split: list.splits)

lemma \( q\text{-dequeue-impl-correct-aux} \):
\[
\begin{align*}
[q\alpha \ Q = x#xs] \Rightarrow \text{apsnd } q\alpha (q\text{-dequeue-impl} \ Q) = (x,xs)
\end{align*}
\]
unfolding \( q\alpha\text{-def} q\text{-dequeue-impl-def} \)
by (cases \( Q \)) (auto split: list.splits)

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

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

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

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

lemma \( q\text{-enqueue-impl-hnr} \) [sepref-fr-rules]:
(uncurry (return oo \( q\text{-enqueue-impl} \)), uncurry \( \text{PR-CONST } q\text{-enqueue} \))
\( \in \) nat-assn \( b \ast_a q\text{-assn} \rightarrow_a q\text{-assn} \)
apply (sepref-to-hoare)
unfolding \( q\text{-assn-def} q\text{-enqueue-def} \text{ pure-def} \)
by (sepauto simp: in-br-cone refine-pw-simps)

lemma \( q\text{-dequeue-impl-hnr} \) [sepref-fr-rules]:
(return o \( q\text{-dequeue-impl} \), \( q\text{-dequeue} \)) \( \in \) q-assn \( d \rightarrow_a \text{nat-assn} \times_a q\text{-assn} \)
apply (sepref-to-hoare)
unfolding \( q\text{-assn-def} q\text{-dequeue-def} \text{ pure-def prod-assn-def} \)
by (sepauto simp: in-br-cone refine-pw-simps split: prod.split)
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
\rightarrow a ((x-assn \times a cf-assn) \times a q-assn)
unfolding fifo-push2-def PR-CONST-def
by sepref
concrete-definition (in –) fifo-push-impl
uses Network-Impl.fifo-push-impl.refine-raw is (uncurry3 ?f,\_)\in-
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
\rightarrow a clc-assn \times a q-assn
unfolding fifo-gap-relabel2-def PR-CONST-def
by sepref
concrete-definition (in –) fifo-gap-relabel-impl
uses Network-Impl.fifo-gap-relabel-impl.refine-raw is (uncurry5 ?f,\_)\in-

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
\rightarrow 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,\_)\in-

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
\rightarrow a (x-assn \times a cf-assn) \times a clc-assn \times a q-assn
unfolding fifo-discharge2-def PR-CONST-def
by sepref
concrete-definition (in –) fifo fifo-discharge-impl
uses Network-Impl.fifo fifo-discharge-impl.refine-raw is (uncurry5 ?f,\_)\in-
lemmas [sepref-fr-rules] = fifo fifo-discharge-impl.refine[OF Network-Impl-axioms]

7.4.6 Computing the Initial State

sepref-thm fifo-init-C-impl is (PR-CONST init-C)
:: am-assn \rightarrow a nat-assn
unfolding init-C-def PR-CONST-def
by sepref
concrete-definition (in –) fifo-init-C-impl
uses Network-Impl.fifo-init-C.impl.refine-raw is (?f,-)∈-

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

sepref-thm fifo-q-init-impl is (PR-CONST q-init)
:: am-assn\(^k\)→\(_a\) 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-run2)
:: 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-init2)
:: 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-alt PR-CONST-def
by sepref

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

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 \(AM: \text{is-adj-map am}\)
shows \(<\text{am-assn am ami}>\)
\(\text{fifo-push-relabel-impl c s t N ami}\)
\(<\lambda cf. \exists A cf. \text{am-assn am ami} * cf-assn cf cf}\)
\(* ↑(\text{isMaxFlow (flow-of-cf cf)} ∧ \text{RGraph-Impl c s t N cf}) >\)

proof —

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

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

note \(R2 = \text{fifo-push-relabel-impl-refine}[\)
\(\text{OF Network-Impl-axioms, to-hnr, unfolded autoref-tag-defs}\)
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 \(RGII = \text{rgraph-and-network-impl-imp-rgraph-impl}[\text{OF}\)
\(\text{RPreGraph, maxflow-impl-rgraph}\)
\(\text{Network-Impl-axioms}\)

show \(?\text{thesis}\)
by (sep-auto
heap: \(R4\)
simp: \(RGII\)
simp: \text{pw-le-iff refine-pw-simps in-br-conv})

qed

7.6 Combination with Network Checker and Main Correctness Theorem

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

\textbf{theorem} fifo-push-relabel-impl-tab-am-correct[sep-heap-rules]:
\begin{itemize}
\item \textbf{assumes} NW: $\text{Network } c s t$
\item \textbf{assumes} VN: $\text{Graph. } V c \subseteq \{0..<N\}$
\item \textbf{assumes} ABS-PS: $\text{Graph. is-adj-map } c \text{ am}$
\end{itemize}
\textbf{shows}
\begin{itemize}
\item 
\begin{align*}
\lambda(ami, cfi) . \exists \text{ cf }. & \quad \text{ am-assn } N \text{ am } ami \ast \text{ cf-assn } N \text{ cf } cfi \\
\ast \uparrow(\text{Network. isMaxFlow } c s t (\text{Network. flow-of-cf } c cf) \wedge \text{ RGraph-Impl } c s t N \text{ cf}) \}
\end{align*}
\end{itemize}
\textbf{proof} –
\begin{itemize}
\item \textbf{interpret} $\text{Network } c s t$ \text{ by fact}
\item \textbf{interpret} $\text{Network-Impl } c s t N$ \text{ using VN by unfold-locales}
\end{itemize}
\textbf{from} ABS-PS \textbf{have} \begin{itemize}
\item [simp]: $am \ u = [] \text{ if } u \geq N$ \text{ for } u
\end{itemize}
\textbf{unfolding} \text{ is-adj-map-def}
\textbf{using} E-ss-VxV VN \text{ that}
\textbf{apply} (subgoal-tac w\# V)
\textbf{by} \begin{itemize}
\item (auto simp del: inV-less-N)
\end{itemize}
\textbf{show} ?\textbf{thesis}
\textbf{unfolding} fifo-push-relabel-impl-tab-am-def
\textbf{apply} vcg
\textbf{apply} \begin{itemize}
\item (rule Hoare-Triple.cons-rule[OF - fifo-push-relabel-impl-correct[OF ABS-PS]])
\end{itemize}
\textbf{subgoal unfolding} am-assn-def is-nf-def \textbf{by} sep-auto
\textbf{apply} \begin{itemize}
\item (rule ent-refl)
\end{itemize}
\textbf{subgoal} \textbf{by} sep-auto
\textbf{done}
\textbf{qed}

\textbf{definition} fifo-push-relabel el s t \equiv do \{
\begin{itemize}
\item case prepareNet el s t of
\item None \Rightarrow return None
\item Some \((c, \text{am}, \text{N})\) \Rightarrow do \\
\item \((ami, cf) \leftarrow \text{fifo-push-relabel-impl-tab-am } c s t \text{ N am};
\item return \((\text{Some } (c, ami, N, cf))\)
\end{itemize}
\}
\textbf{export-code} fifo-push-relabel \textbf{checking} SML-imp

Main correctness statement:
• If `fifo-push-relabel` returns `None`, the edge list was invalid or described an invalid network.

• If it returns `Some (c, am, N, cfi)`, then the edge list is valid and describes a valid network. Moreover, `cfi` is an integer square matrix of dimension `N`, which describes a valid residual graph in the network, whose corresponding flow is maximal. Finally, `am` is a valid adjacency map of the graph, and the nodes of the graph are integers less than `N`.

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

\[
\begin{align*}
\lambda & (\text{emp}) \\
\text{fifo-push-relabel } e & \ s & \ t \\
< & \lambda \\
\text{None } \Rightarrow & \uparrow (\neg \text{ln-invar } e & \lor \neg \text{Network (ln-\alpha } e) \ s \ t) \\
\mid & \text{Some (c, am, N, cfi)} \Rightarrow \\
& \uparrow (c = \text{ln-\alpha } e & \land \text{ln-invar } e & \land \text{Network } c & s \ t) \\
& \exists \text{am cf. am-assn } N & \text{am am} & \text{cf-assn } N & \text{cf cfi} \\
& \land \text{Network.isMaxFlow } c & s & t (\text{Network.} \text{flow-of-cf } c & \text{cf}) \\
\end{align*}
\]

> \text{t}

**Unfolding** `fifo-push-relabel-def`

**Using** `prepareNet-correct[of el s t]`

**By** `(sep-auto simp: ln-rel-def in-br-cone)`

### 7.6.1 Justification of Splitting into Prepare and Run Phase

**Definition** `fifo-push-relabel-prepare-impl el s t`:

\[
\text{case prepareNet } el & \ s & \ t \text{ of} \\
\text{None } \Rightarrow & \text{return None} \\
\mid & \text{Some (c, am, N)} \Rightarrow \text{do} \\
& \text{ami } \leftarrow \text{Array.make } N & \text{am;} \\
& \text{cfi } \leftarrow \text{fifo-push-relabel-init-impl } c & N; \\
& \text{return (Some (N, am, N, cfi))} \\
\}
\]

**Theorem** `justify-fifo-push-relabel-prep-run-split`:

\[
\text{fifo-push-relabel } e & \ s & \ t = \\
\text{do} \\
& \text{pr } \leftarrow \text{fifo-push-relabel-prepare-impl } e & \ s & \ t; \\
\text{case pr of} \\
\text{None } \Rightarrow & \text{return None} \\
\mid & \text{Some (N, ami, c) } \Rightarrow \text{do} \\
& \text{cf } \leftarrow \text{fifo-push-relabel-run-impl } s & t & N & \text{ami cf}; \\
& \text{return (Some (c, ami, N, cf))} \\
\}
\]
7.7 Usage Example: Computing Maxflow Value

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

definition 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

definition fifo-push-relabel-compute-flow-val-def
by sep-auto simp: Network.is-max-flow-val-def aux

qed

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

end
8 Implementation of Relabel-to-Front

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

8.1 Basic Operations

context Network-Impl
begin

8.1.1 Neighbor Lists

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

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

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

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

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

lemma n-init-refine[refine2]:
  assumes AM: is-adj-map am
  shows n-init am
    ≤ (spec c. (c, rtf-init-n) ∈ (nat-rel ⇒ (nat-rel)list-set-rel))
proof −
have[simp]: am v = [] if \( v \not\in V \) for v

proof -
  from that have adjacent-nodes v = {}
  unfolding adjacent-nodes-def using E-ss-VxV by auto
  thus ?thesis using am-to-adj-nodes-refine[OF AM]
  by (auto simp: list-set-rel-def in-br-conv)
qed

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

8.2 Refinement to Basic Operations

8.2.1 Discharge

definition discharge2 am x cf l n u ≡ do
assert (u ∈ V);
monadic-WHILEIT (λ-. True)
  (λ((x,cf),l,n). do { xu ← x-get x u; return (xu ≠ 0) } )
  (λ((x,cf),l,n). do {
    at-end ← n-at-end n u;
    if at-end then do {
      l ← relabel2 am cf l u;
      n ← n-reset am n u;
      return ((x,cf),l,n)
    } else do {
      v ← n-get-hd n u;
      cfuv ← cf-get cf (u,v);
      lu ← l-get l u;
      lv ← l-get l v;
      if (cfuv ≠ 0 ∧ lu = lv + 1) then do {
        (x,cf) ← push2 x cf (u,v);
        return ((x,cf),l,n)
      } else do {
        n ← n-move-next n u;
        return ((x,cf),l,n)
      }
    }
  }) ((x,cf),l,n)
)

lemma discharge-structure-refine-aux:
  assumes SR: (ni,n)∈nat-rel → ⟨nat-rel⟩ list-set-rel
  assumes SU: (ui,u)∈Id
  assumes fNR: fNi ≤ ⇓R fN
  assumes UIV: u∈V − {s,t}
  assumes fSR: \( v \not\in V \) for v
\[(vi,v) \in \text{Id}; \ v \in n \ u; \ ni \ u = v \# vs; \ (v \# vs, n \ u) \in \langle \text{nat-rel} \rangle \text{list-set-rel} \]

\[\Rightarrow fSi \ vi \leq \downarrow R (fS v)\]

shows

\[\begin{aligned}
\text{do}\left\{\right.
\begin{array}{l}
\text{at-end} \leftarrow n\text{-at-end} ni ui;
\text{if at-end then } fNi
\text{else do }\left\{\right.
\begin{array}{l}
v \leftarrow n\text{-get-hd} ni ui;
\text{fSi } v
\end{array}
\right.\right.
\end{array}
\begin{array}{l}
\} \}
\end{array}
\end{aligned}\]

\[\downarrow R (\text{do }\left\{\right.
\begin{array}{l}
\text{v} \leftarrow \text{select } v. \ v \in n \ u;
\text{case } v \text{ of }
\begin{array}{l}
\text{None } \Rightarrow fN
\mid \text{Some } v \Rightarrow fS v
\end{array}
\right.\right.
\begin{array}{l}
\} \}
\end{array})(\text{is } ?\text{lhs} \leq \downarrow R ?\text{rhs})\]

unfolding n-at-end-def n-get-hd-def

apply (simp only: nres-monad-laws)

apply (cases ni u)

subgoal

using fun-relD[OF SR SU] SU UIV fNR

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

subgoal for v vs

using fun-relD[OF SR SU] SU UIV

using fSR[OF IdI[of v], of vs]

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

by fastforce

done

lemma xf-rel-RELATES[refine-dref-RELATES]: RELATES xf-rel

by (auto simp: RELATES-def)

lemma discharge2-refine[refine]:

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

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

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

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

shows discharge2 am x cf li ni ui

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

unfolding discharge2-def discharge-def

apply (rewrite in monadic-WHILEIT - - - vseg-intro-frame)

apply refine-reg

apply (vc-solve simp: A NR)

subgoal by (simp add: xf-rel-def x-get-def)
subgoal for \( f l n x c f n i \)
apply \( (\text{subst vcg-rem-frame}) \)
unfolding \( n\text{-reset-def} cf\text{-def} l\text{-get-def} n\text{-move-next-def} \)
apply \( (\text{simp only: nres-monad-laws}) \)
apply \( (\text{rule discharge-structure-refine-aux}; (\text{refine-vcg AM})?) \)
apply \( (\text{assumption})? \)

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

done

done

8.2.2 Initialization of Queue

lemma \( V\text{-is-adj-nodes} \): \( V = \{ v .\ adjacent-nodes v \neq \{\} \} \)
unfolding \( V\text{-def adjacent-nodes-def} \)

definition \( \text{init-CQ am} \equiv \text{do} \{ \)
let \( \text{cardV} = 0 \);
let \( Q = [] \);
nfoldli \( [0..<N] \) \( (\lambda v. \text{True}) \) \( (\lambda v (\text{cardV},Q). \) \( \{ \)
assert \( (v < N) \);
in\( V \leftarrow \text{am-is-in-V am v} \);
if in\( V \) then \( \{ \)
let \( \text{cardV} = \text{cardV} + 1 \);
if \( v \neq s \land v \neq t \) then 
return \( (\text{cardV},v \# Q) \)
else 
return \( (\text{cardV},Q) \)
\} \) \( (\text{cardV},Q) \)
\}

lemma \( \text{init-CQ-correct}[\text{THEN order-trans, refine-vcg}]: \)
assumes \( \text{AM}: is\text{-adj-map am} \)
shows \( \text{init-CQ am} \leq \text{SPEC} \ (\lambda (C,Q). C = \text{card V} \land \text{distinct Q} \land \text{set Q} = \text{V} - \{s,t\}) \)
unfolding \( \text{init-CQ-def} \)
apply \( (\text{refine-vcg}) \)
nfoldli-rule[where \( I = \text{MT - (C,Q)} \),
\( C = \text{card} (V \cap \text{set} \ I) \land \text{distinct Q} \land \text{set Q} = (V \cap \text{set} \ I) - \{s,t\} \)]
apply \( (\text{clarsimp-all simp: am-to-adj-nodes-refine[OF AM]} \)
using \( \text{V-ss} \) by \( (\text{auto simp: upt-eq-lel-conv Int-absorb2}) \)

8.2.3 Main Algorithm

definition \( \text{relabel-to-front2 am} \equiv \text{do} \{ \)
(cardV, L-right) ← init-CQ am;

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

let L-left = [];

((x, cf), l, n, L-left, L-right) ← while_T
(λ((x, cf), l, n, L-left, L-right). L-right ≠ [])
(λ((x, cf), l, n, L-left, L-right). do {
  assert (L-right ≠ []);
  let u = hd L-right;
  old-lu ← l-get l u;
  (x, cf), l, n) ← discharge2 am x cf l n u;
  lu ← l-get l u;
  if (lu ≠ old-lu) then do {
    — Move u to front of l, and restart scanning L. The cost for
    — rev-append is amortized by going to next node in L
    let (L-left, L-right) = ([u], rev-append L-left (tl L-right));
    return ((x, cf), l, n, L-left, L-right)
  } else do {
    — Goto next node in L
    let (L-left, L-right) = (u#L-left, tl L-right);
    return ((x, cf), l, n, L-left, L-right)
  }
}) (xcf, l, n, L-left, L-right);

return cf

lemma relabel-to-front2-refine[refine]:
assumes AM: is-adj-map am
shows relabel-to-front2 am
≤ ↓ (br (flow-of-cf) (RPreGraph c s t)) relabel-to-front
proof
 define s-rel
:: ( - × (capacity-impl flow
  × (nat⇒nat)
  × (node⇒node set)
  × node list
  × node list)) set
where s-rel ≡
  xf-rel
\times_r \text{Id}
\times_r (\text{nat-rel} \to (\text{nat-rel})\text{-list-set-rel})
\times_r \text{br rev} (\lambda\cdot \text{True})
\times_r \text{Id}

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

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

\text{have AUXR: do \{ 
(\text{cardV}, \text{L-right}) \leftarrow \text{init-CQ am}; 
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} \{ 
\text{L-right} \leftarrow \text{spec l. distinct l} \land \text{set l} = V - \{s, t\}; 
F \text{ L-right} 
\})

\text{if} \ \bigwedge \text{L-right xcf n.} \\
\ [ (xcf,pp-init-f) \in xcf-rel; (n,rtf-init-n) \in \text{nat-rel} \to (\text{nat-rel})\text{-list-set-rel} ] \\
\implies \text{Fi L-right xcf pp-init-l n} \leq \downarrow R (F \text{ L-right})

\text{for Fi F R} \\
\text{unfolding l-init-def} \\
\text{apply (rule refine2specI)} \\
\text{supply pp-init-xcf2-refine} \\
\ [ \text{OF AM, unfolded conc-fun-RETURN, THEN order-trans, refine-vcg} ] \\
\text{supply n-init-refine[OF AM,THEN order-trans, refine-vcg]} \\
\text{apply (refine-vcg AM V-ss)} \\
\text{apply clarsimp} \\
\text{subgoal for L-right x cf n} \\
\text{using (that[of (x,cf) n L-right])} \\
\text{unfolding pp-init-l-def} \\
\text{by (clarsimp simp: pw-le-iff refine-pw-simps; meson)} \\
\text{done} \\
\text{show ?thesis} \\
\text{unfolding relabel-to-front2-def relabel-to-front-def Let-def l-get-def} \\
\text{apply (simp only: ares-monad-laws)} \\
\text{apply (rule AUXR)} \\
\text{apply (refine-vcg)}

115
apply refine-dref-type
unfolding s-rel-def
apply (rc-solve
    simp: in-br-conv rev-append-eq zv-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^k → 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^k * a node-assn^k → 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 \( \text{(in \(-\)) \ n\text{-get-hd-impl \ n \ u \equiv do} \) \{ 
    \text{nu} \leftarrow \text{Array.nth \ n \ u;}
    \text{return} \ (\text{hd \ nu})
\}

lemma \([\text{sepref-fr-rules}]\):
\((\text{uncurry \ n\text{-get-hd-impl, uncurry \ (PR-CONST \ n\text{-get-hd})}})\) \in \text{n-assn}^k \ast_a \text{node-assn}^k \rightarrow_a \text{node-assn}
apply \text{sepref-to-hoare} unfolding \text{n\text{-get-hd-impl-def \ n\text{-get-hd-def \ n\text{-assn-def}}}
by (\text{sep-auto simp: refine-pw-simps})

definition \( \text{(in \(-\)) \ n\text{-move-next-impl \ n \ u \equiv do} \) \{ 
    \text{nu} \leftarrow \text{Array.nth \ n \ u;}
    \text{n} \leftarrow \text{Array.update \ u \ (tl \ nu);}
    \text{return} \ n
\}

lemma \([\text{sepref-fr-rules}]\):
\((\text{uncurry \ n\text{-move-next-impl, uncurry \ (PR-CONST \ n\text{-move-next})}})\) \in \text{n-assn}^d \ast_a \text{node-assn}^k \rightarrow_a \text{n-assn}
apply \text{sepref-to-hoare} unfolding \text{n\text{-move-next-impl-def \ n\text{-move-next-def \ n\text{-assn-def}}}
by (\text{sep-auto simp: refine-pw-simps})

definition \( \text{(in \(-\)) \ n\text{-reset-impl \ am \ n \ u \equiv do} \) \{ 
    \text{nu} \leftarrow \text{Array.nth \ am \ u;}
    \text{n} \leftarrow \text{Array.update \ u \ nu \ n;}
    \text{return} \ n
\}

lemma \([\text{sepref-fr-rules}]\):
\((\text{uncurry2 \ n\text{-reset-impl, uncurry2 \ (PR-CONST \ n\text{-reset})}})\) \in \text{am-assn}^k \ast_a \text{n-assn}^d \ast_a \text{node-assn}^k \rightarrow_a \text{n-assn}
apply \text{sepref-to-hoare} unfolding \text{n\text{-reset-impl-def \ n\text{-assn-def \ am\text{-assn-def}}}
by (\text{sep-auto simp: refine-pw-simps})

8.3.2 Discharge

\text{sepref-thm discharge-impl is uncurry5 \ (PR-CONST discharge2)}
\:: \text{am-assn}^k \ast_a \text{x-assn}^d \ast_a \text{cf-assn}^d \ast_a \text{l-assn}^d \ast_a \text{n-assn}^d \ast_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 discharge2-def \text{PR-CONST-def}
by sepref

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

8.3.3 Initialization of Queue

\text{sepref-thm init-CQ-impl is \(\text{(PR-CONST init-CQ)}\)
\:: \text{am-assn}^k \rightarrow_a \text{nat-assn} \times_a \text{list-assn \ nat-assn}
unfolding 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 ∈ 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
<cfifi. ≥cfifi. cf-assn cf efi
    * (isMaxFlow (flow-of-cf cf) ∧ RGraph-Impl c s t N cf)≥t

proof −
note relabel-to-front2-refine[OF AM]
also note relabel-to-front-correct
finally have R1:
relabel-to-front2 am
≤ ↓ (br flow-of-cf (RPreGraph c s t)) (SPEC isMaxFlow) .

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

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

note RGII = rgraph-and-network-impl-imp-rgraph-impl[OF
RPreGraph.maxflow-imp-rgraph

118
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;
  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
<emp>
  relabel-to-front-impl-tab-am c s t N am
<λcf. ∃A cf.
  asmtx-assn N id-assn cf A
  * ↑(Network.isMaxFlow c s t (Network.flow-of-cf c cf)
    ∧ RGraph-Impl c s t N cf
  )>_{t}
proof
  interpret Network c s t by fact
  interpret Network-Impl c s t N using VN by unfold-locales

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

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

definition relabel-to-front el s t =
  do
  case prepareNet el s t of
  None ⇒ return None

Some \( (c, am, N) \Rightarrow \text{do} \{
    cf \leftarrow \text{relabel-to-front-impl-tab-am } c \ s \ t \ N \ am;
    \text{return } \left( \text{Some } (c, am, N, cf) \right)
\}\}

\text{export-code relabel-to-front checking SML-imp}

Main correctness statement:

- If \text{relabel-to-front} returns \text{None}, the edge list was invalid or described an invalid network.

- If it returns \text{Some } \( (c, am, N, cfi) \), then the edge list is valid and describes a valid network. Moreover, \text{cfi} is an integer square matrix of dimension \( N \), which describes a valid residual graph in the network, whose corresponding flow is maximal. Finally, \text{am} is a valid adjacency map of the graph, and the nodes of the graph are integers less than \( N \).

\text{theorem relabel-to-front-correct}:
\begin{align*}
\text{<emp> relabel-to-front el s t}<
\text{\lambda}
\text{None } \Rightarrow \uparrow (\neg \text{ln-invar el } \lor \neg \text{Network (ln-\alpha el) s t}) \\
| \text{Some } (c, am, N, cfi) \Rightarrow \\
\uparrow (c = \text{ln-\alpha el } \land \text{ln-invar el}) \\
\ast (\exists A cf. \text{asmtx-assn N int-assn cf cfi}) \\
\ast \uparrow (\text{RGraph-Impl c s t N cf} \\
\land \text{Network.isMaxFlow c s t (Network.flow-of-cf c cf) }) \\
\ast \uparrow (\text{Graph.is-adj-map c am}) \\
>_{t}
\end{align*}

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

\text{end}

\textbf{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


