Using Orlin's Algorithm

Thomas Ammer

October 15, 2025

Contents

0.1 Orlin's Algorithm Executable

theory *Instantiation*
imports *Graph-Algorithms-Dev.RBT-Map-Extension*
Directed-Set-Graphs.Pair-Graph-RBT
Graph-Algorithms-Dev.Bellman-Ford-Example
Graph-Algorithms-Dev.DFS-Example
Mincost-Flow-Algorithms.Orlins
begin

0.1.1 Definitions

hide-const *RBT-Set.empty Set.empty not-blocked-update*
notation *vset-empty* ($\emptyset_N$)

fun *list-to-rbt* :: ('*a*::*linorder*) *list* ⇒ '*a rbt* **where**
list-to-rbt [] = *Leaf*
| *list-to-rbt* (*x*#*xs*) = *vset-insert x* (*list-to-rbt xs*)

value *vset-diff* (*list-to-rbt* [*1*::*nat*, *2*, *3*, *4*,*6*]) (*list-to-rbt* [*0*..*20*])

set of edges

definition *get-from-set* = *List.find*
fun *are-all* **where** *are-all P* (*Nil*) = *True*|
are-all P (*x*#*xs*) = (*P x* ∧ *are-all P xs*)
definition *set-invar* = *distinct*
definition *to-set* = *set*
definition *to-list* = *id*

notation *map-empty* ($\emptyset_G$)

definition *flow-empty* = *vset-empty*
definition *flow-update* = *update*
definition *flow-delete* = *RBT-Map.delete*
definition *flow-lookup* = *lookup*
definition *flow-invar* = $(\lambda t.\ M.invar\ t \wedge\ \ rbt\text{-}red\ t)$

definition *bal-empty* = *vset-empty*
definition *bal-update* = *update*
definition *bal-delete* = *RBT-Map.delete*
definition *bal-lookup* = *lookup*
definition *bal-invar* = $(\lambda t.\ M.invar\ t \wedge\ \ rbt\text{-}red\ t)$

definition *rep-comp-empty* = *vset-empty*
definition *rep-comp-update* = *update*
definition *rep-comp-delete* = *RBT-Map.delete*
definition *rep-comp-lookup* = *lookup*
definition *rep-comp-invar* = $(\lambda t.\ M.invar\ t \wedge\ \ rbt\text{-}red\ t)$

definition *conv-empty* = *vset-empty*
definition *conv-update* = *update*
definition *conv-delete* = *RBT-Map.delete*
definition *conv-lookup* = *lookup*
definition *conv-invar* = $(\lambda t.\ M.invar\ t \wedge\ \ rbt\text{-}red\ t)$

definition *not-blocked-empty* = *vset-empty*
definition *not-blocked-update* = *update*
definition *not-blocked-delete* = *RBT-Map.delete*
definition *not-blocked-lookup* = *lookup*
definition *not-blocked-invar* = $(\lambda t.\ M.invar\ t \wedge\ \ rbt\text{-}red\ t)$

definition *rep-comp-upd-all* = $(update\text{-}all :: ('a \Rightarrow 'a \times nat \Rightarrow 'a \times nat)$
$\Rightarrow (('a \times 'a \times nat) \times color)\ tree$
$\Rightarrow (('a \times 'a \times nat) \times color)\ tree)$
definition *not-blocked-upd-all* = $(update\text{-}all :: ('edge\text{-}type \Rightarrow bool \Rightarrow bool)$
$\Rightarrow (('edge\text{-}type \times bool) \times color)\ tree$
$\Rightarrow (('edge\text{-}type \times bool) \times color)\ tree)$
definition *flow-update-all* = $(update\text{-}all :: ('edge\text{-}type \Rightarrow real \Rightarrow real)$
$\Rightarrow (('edge\text{-}type \times real) \times color)\ tree$
$\Rightarrow (('edge\text{-}type \times real) \times color)\ tree)$

lemma *rep-comp-upd-all*:
$\bigwedge rep\ f.\ rep\text{-}comp\text{-}invar\ rep \Longrightarrow (\bigwedge x.\ x \in dom\ (rep\text{-}comp\text{-}lookup\ rep)$
$\Longrightarrow rep\text{-}comp\text{-}lookup\ (rep\text{-}comp\text{-}upd\text{-}all\ f\ rep)\ x =$
$Some\ (f\ x\ (the\ (rep\text{-}comp\text{-}lookup\ rep\ x))))$
$\bigwedge rep\ f\ g.\ rep\text{-}comp\text{-}invar\ rep \Longrightarrow (\bigwedge x.\ x \in dom\ (rep\text{-}comp\text{-}lookup\ rep) \Longrightarrow$
$f\ x\ (the\ (rep\text{-}comp\text{-}lookup\ rep\ x)) = g\ x\ (the\ (rep\text{-}comp\text{-}lookup\ rep$

x))) $\Longrightarrow$
rep-comp-upd-all f rep = *rep-comp-upd-all g rep*
$\bigwedge$ *rep f*. *rep-comp-invar rep* $\Longrightarrow$ *rep-comp-invar* (*rep-comp-upd-all f rep*)
$\bigwedge$ *rep f*. *rep-comp-invar rep* $\Longrightarrow$ *dom* (*rep-comp-lookup* (*rep-comp-upd-all f rep*))
= *dom* (*rep-comp-lookup rep*)
and *not-blocked-upd-all*:
$\bigwedge$ *nblckd f*. *not-blocked-invar nblckd* $\Longrightarrow$ ($\bigwedge$ *x*. *x* $\in$ *dom* (*not-blocked-lookup nblckd*)
$\Longrightarrow$ *not-blocked-lookup* (*not-blocked-upd-all f nblckd*) *x* =
Some (*f x* (*the* (*not-blocked-lookup nblckd x*))))
$\bigwedge$ *nblckd f g*. *not-blocked-invar nblckd* $\Longrightarrow$ ($\bigwedge$ *x*. *x* $\in$ *dom* (*not-blocked-lookup nblckd*) $\Longrightarrow$
f x (*the* (*not-blocked-lookup nblckd x*)) = *g x* (*the* (*not-blocked-lookup nblckd x*))) $\Longrightarrow$
not-blocked-upd-all f nblckd = *not-blocked-upd-all g nblckd*
$\bigwedge$ *nblckd f*. *not-blocked-invar nblckd* $\Longrightarrow$ *not-blocked-invar* (*not-blocked-upd-all f nblckd*)
$\bigwedge$ *nblckd f*. *not-blocked-invar nblckd* $\Longrightarrow$ *dom* (*not-blocked-lookup* (*not-blocked-upd-all f nblckd*))
= *dom* (*not-blocked-lookup nblckd*)
and *flow-update-all*:
$\bigwedge$ *fl f*. *flow-invar fl* $\Longrightarrow$ ($\bigwedge$ *x*. *x* $\in$ *dom* (*flow-lookup fl*)
$\Longrightarrow$ *flow-lookup* (*flow-update-all f fl*) *x* =
Some (*f x* (*the* (*flow-lookup fl x*))))
$\bigwedge$ *fl f g*. *flow-invar fl* $\Longrightarrow$ ($\bigwedge$ *x*. *x* $\in$ *dom* (*flow-lookup fl*) $\Longrightarrow$
f x (*the* (*flow-lookup fl x*)) = *g x* (*the* (*flow-lookup fl x*))) $\Longrightarrow$
flow-update-all f fl = *flow-update-all g fl*
$\bigwedge$ *fl f*. *flow-invar fl* $\Longrightarrow$ *flow-invar* (*flow-update-all f fl*)
$\bigwedge$ *fl f*. *flow-invar fl* $\Longrightarrow$ *dom* (*flow-lookup* (*flow-update-all f fl*))
= *dom* (*flow-lookup fl*)
and *get-max*: $\bigwedge$ *b f*. *bal-invar b* $\Longrightarrow$ *dom* (*bal-lookup b*) $\neq$ {}
$\Longrightarrow$ *get-max f b* = *Max* {*f y* (*the* (*bal-lookup b y*)) | *y*. *y* $\in$ *dom* (*bal-lookup b*)}
and *to-list*: $\bigwedge$ *E*. *set-invar E* $\Longrightarrow$ *to-set E* = *set* (*to-list E*)
$\bigwedge$ *E*. *set-invar E* $\Longrightarrow$ *distinct* (*to-list E*)
using *update-all*(*3*)
by (*auto simp add*: *rep-comp-lookup-def rep-comp-upd-all-def rep-comp-invar-def*

M.invar-def update-all(*1*) *color-no-change rbt-red-def rbt-def*
not-blocked-invar-def not-blocked-lookup-def not-blocked-upd-all-def
flow-invar-def flow-lookup-def flow-update-all-def bal-invar-def
bal-update-def bal-lookup-def to-list-def to-set-def set-invar-def
intro!: *update-all*(*2*,*3*,*4*) *get-max-correct*)

interpretation *adj*: *Map*
where *empty* = *vset-empty* **and** *update*=*edge-map-update* **and**
delete=*delete* **and** *lookup*= *lookup* **and** *invar*=*adj-inv*
using *RBT-Map.M.Map-axioms*
by(*auto simp add*: *Map-def rbt-red-def rbt-def M.invar-def edge-map-update-def adj-inv-def RBT-Set.empty-def*)

lemmas *Map-satisfied* = *adj.Map-axioms*

lemmas *Set-satisified* = *dfs.Graph.vset.set.Set-axioms*

lemma *Set-Choose-axioms*: *Set-Choose-axioms vset-empty isin sel*
  apply(*rule Set-Choose-axioms.intro*)
  unfolding *RBT-Set.empty-def*
  subgoal for *s*
    by(*induction rule*: *sel.induct*) *auto*
  done

lemmas *Set-Choose-satisfied* = *dfs.Graph.vset.Set-Choose-axioms*

interpretation *Pair-Graph-Specs-satisfied*:
    Pair-Graph-Specs map-empty RBT-Map.delete lookup vset-insert isin t-set sel
    edge-map-update adj-inv vset-empty vset-delete vset-inv
  using *Set-Choose-satisfied Map-satisfied*
  by(*auto simp add*: *Pair-Graph-Specs-def map-empty-def RBT-Set.empty-def*)

lemmas *Pair-Graph-Specs-satisfied* = *Pair-Graph-Specs-satisfied.Pair-Graph-Specs-axioms*

lemmas *Set2-satisfied* = *dfs.set-ops.Set2-axioms*

definition *realising-edges-empty* = (*vset-empty*::((('*a* ::*linorder*× '*a*) × ('*edge-type list*)) × *color*) *tree*)
definition *realising-edges-update* = *update*
definition *realising-edges-delete* = *RBT-Map.delete*
definition *realising-edges-lookup* = *lookup*
definition *realising-edges-invar* = *M.invar*

interpretation *Map-realising-edges*:
 Map realising-edges-empty realising-edges-update realising-edges-delete
    realising-edges-lookup realising-edges-invar
  using *RBT-Map.M.Map-axioms*
   by(*auto simp add*: *realising-edges-update-def realising-edges-empty-def realising-edges-delete-def*
           realising-edges-lookup-def realising-edges-invar-def)

lemmas *Map-realising-edges* = *Map-realising-edges.Map-axioms*

lemmas *Map-connection* = *Map-connection.Map-axioms*

lemmas *bellman-ford-spec* = *bellford.bellman-ford-spec-axioms*

locale *function-generation* =

 Map-realising: *Map realising-edges-empty realising-edges-update*::(('*a*)× '*a*) ⇒ '*edge-type list* ⇒ '*realising-type* ⇒ '*realising-type*

$realising\text{-}edges\text{-}delete\ realising\text{-}edges\text{-}lookup\ realising\text{-}edges\text{-}invar\ +$

$Map\text{-}bal$: $Map\ bal\text{-}empty\ bal\text{-}update::'a \Rightarrow real \Rightarrow 'bal\text{-}impl \Rightarrow 'bal\text{-}impl$
$bal\text{-}delete\ bal\text{-}lookup\ bal\text{-}invar\ +$

$Map\text{-}flow$: $Map\ flow\text{-}empty::'flow\text{-}impl\ flow\text{-}update::'edge\text{-}type \Rightarrow real \Rightarrow 'flow\text{-}impl \Rightarrow 'flow\text{-}impl$
$flow\text{-}delete\ flow\text{-}lookup\ flow\text{-}invar\ +$

$Map\text{-}not\text{-}blocked$: $Map\ not\text{-}blocked\text{-}empty\ not\text{-}blocked\text{-}update::'edge\text{-}type \Rightarrow bool \Rightarrow 'nb\text{-}impl \Rightarrow 'nb\text{-}impl$
$not\text{-}blocked\text{-}delete\ not\text{-}blocked\text{-}lookup\ not\text{-}blocked\text{-}invar\ +$

$Map\ rep\text{-}comp\text{-}empty\ rep\text{-}comp\text{-}update::'a \Rightarrow ('a \times nat) \Rightarrow 'rcomp\text{-}impl \Rightarrow 'rcomp\text{-}impl\ rep\text{-}comp\text{-}delete$
$rep\text{-}comp\text{-}lookup\ rep\text{-}comp\text{-}invar$

for

realising-edges-empty
realising-edges-update
realising-edges-delete
realising-edges-lookup
realising-edges-invar

bal-empty
bal-update
bal-delete
bal-lookup
bal-invar

flow-empty
flow-update
flow-delete
flow-lookup
flow-invar

not-blocked-empty
not-blocked-update
not-blocked-delete
not-blocked-lookup
not-blocked-invar

rep-comp-empty
rep-comp-update
rep-comp-delete
rep-comp-lookup
rep-comp-invar+

fixes $\mathcal{E}$*-impl*::*'edge-type-set-impl*
and *to-list*:: *'edge-type-set-impl* ⇒ *'edge-type list*
and *fst*::*'edge-type* ⇒ (*'a*::*linorder*)
and *snd*::*'edge-type* ⇒ *'a*
and *create-edge*::*'a* ⇒ *'a* ⇒ *'edge-type*
and c*-impl*::*'c-impl*
and b*-impl*::*'bal-impl*
and *to-set*::*'edge-type-set-impl* ⇒ *'edge-type set*
and *c-lookup*::*'c-impl* ⇒ *'edge-type* ⇒ *real option*
begin

definition *make-pair e* ≡ (*fst e*, *snd e*)

definition u = (λ *e*::*'edge-type*. *PInfty*)
definition c *e* = (*case* (*c-lookup* c*-impl e*) *of Some c* ⇒ *c* | *None* ⇒ *0*)
definition $\mathcal{E}$ **where** $\mathcal{E}$ = *to-set* $\mathcal{E}$*-impl*
definition *N* = *length* (*remdups* (*map fst* (*to-list* $\mathcal{E}$*-impl*) @ *map snd* (*to-list* $\mathcal{E}$*-impl*)))
definition ε = *1* / (*max 3* (*real N*))

definition b = (λ *v*. *if bal-lookup* b*-impl v* ≠ *None then the* (*bal-lookup* b*-impl v*) *else 0*)
abbreviation *EEE* ≡ *flow-network-spec*.$\mathfrak{E}$ $\mathcal{E}$
abbreviation *fstv* == *flow-network-spec.fstv fst snd*
abbreviation *sndv* == *flow-network-spec.sndv fst snd*

abbreviation *VV* ≡ *dVs* (*make-pair* ' $\mathcal{E}$)

definition *es* = *remdups*(*map make-pair* (*to-list* $\mathcal{E}$*-impl*)@(*map prod.swap* (*map make-pair* (*to-list* $\mathcal{E}$*-impl*))))
definition *vs* = *remdups* (*map prod.fst es*)

definition *dfs F t* = (*dfs.DFS-impl F t*) **for** *F*
definition *dfs-initial s* = (*dfs.initial-state s*)

definition *get-path u v E* = *rev* (*stack* (*dfs E v* (*dfs-initial u*)))

fun *get-source-aux-aux* **where**
get-source-aux-aux b γ [] = *None*|
get-source-aux-aux b γ (*v*#*xs*) =
(*if b v* > (*1* − ε) ∗ γ *then Some v else*
get-source-aux-aux b γ xs)

definition *get-source-aux b γ xs* = (*get-source-aux-aux b γ xs*)

fun *get-target-aux-aux* **where**
get-target-aux-aux b γ [] = *None*|
get-target-aux-aux b γ (*v*#*xs*) =
(*if b v* < − (*1* − ε) ∗ γ *then Some v else*

```
            get-target-aux-aux b γ xs)

definition get-target-aux b γ xs = (get-target-aux-aux b γ xs)

definition ℰ-list = to-list ℰ-impl

definition realising-edges-general list =
         (foldr (λ e found-edges. let ee = make-pair e in
                  (case realising-edges-lookup found-edges ee of
                   None ⇒ realising-edges-update ee [e] found-edges
                  |Some ds ⇒ realising-edges-update ee (e#ds) found-edges))
          list realising-edges-empty)

definition realising-edges = realising-edges-general ℰ-list

definition find-cheapest-forward f nb list e c=
      foldr (λ e (beste, bestc). if nb e ∧ ereal (f e) < u e ∧
                                        ereal (c e) < bestc
                                    then (e, ereal (c e))
                                    else (beste, bestc)) list (e, c)


definition find-cheapest-backward f nb list e c=
      foldr (λ e (beste, bestc). if nb e ∧ ereal (f e) > 0 ∧
                                       ereal (− c e) < bestc
                                    then (e, ereal (− c e))
                                    else (beste, bestc)) list (e, c)


definition get-edge-and-costs-forward nb (f::'edge-type ⇒ real) =
                         (λ u v. (let ingoing-edges = (case (realising-edges-lookup
realising-edges (u, v)) of
                                 None ⇒ []|
                                 Some list ⇒ list);
                outgoing-edges = (case (realising-edges-lookup realising-edges (v, u))
of
                               None ⇒ [] |
                               Some list ⇒ list);
                (ef, cf) = find-cheapest-forward f nb ingoing-edges
                           (create-edge u v) PInfty;
                (eb, cb) = find-cheapest-backward f nb outgoing-edges
                           (create-edge v u) PInfty
                 in (if cf ≤ cb then (F ef, cf) else (B eb, cb))))


definition get-edge-and-costs-backward nb (f::'edge-type ⇒ real) =
                         (λ v u. (let ingoing-edges = (case (realising-edges-lookup
realising-edges (u, v)) of
                                 None ⇒ []|
```

$Some\ list \Rightarrow list);$
$outgoing\text{-}edges = (case\ (realising\text{-}edges\text{-}lookup\ realising\text{-}edges\ (v,\ u))$
of
$None \Rightarrow [] \mid$
$Some\ list \Rightarrow list);$
$(ef,\ cf) = find\text{-}cheapest\text{-}forward\ f\ nb\ ingoing\text{-}edges$
$(create\text{-}edge\ u\ v)\ PInfty;$
$(eb,\ cb) = find\text{-}cheapest\text{-}backward\ f\ nb\ outgoing\text{-}edges$
$(create\text{-}edge\ v\ u)\ PInfty$
$in\ (if\ cf \le cb\ then\ (F\ ef,\ cf)\ else\ (B\ eb,\ cb))))$

definition $bellman\text{-}ford\text{-}forward\ nb\ (f::'edge\text{-}type \Rightarrow real)\ s =$
$bellman\text{-}ford\text{-}algo\ (\lambda\ u\ v.\ prod.snd\ (get\text{-}edge\text{-}and\text{-}costs\text{-}forward\ nb\ f\ u\ v))\ es$
$(length\ vs - 1)$
$(bellman\text{-}ford\text{-}init\text{-}algo\ vs\ s)$

definition $bellman\text{-}ford\text{-}backward\ nb\ (f::'edge\text{-}type \Rightarrow real)\ s =$
$bellman\text{-}ford\text{-}algo\ (\lambda\ u\ v.\ prod.snd\ (get\text{-}edge\text{-}and\text{-}costs\text{-}backward\ nb\ f\ u\ v))$
$es\ (length\ vs - 1)$
$(bellman\text{-}ford\text{-}init\text{-}algo\ vs\ s)$

fun $get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux$ **where**
$get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux\ connections\ b\ \gamma\ \ [] = None|$
$get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux\ connections\ b\ \gamma\ (v\#xs) =$
$(if\ b\ v < \ - \varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ v)) < PInfty$
$then\ Some\ v\ else$
$get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs)$

definition $get\text{-}target\text{-}for\text{-}source\text{-}aux\ connections\ b\ \gamma\ xs = the(get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux$
$connections\ b\ \gamma\ xs)$

fun $get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux$ **where**
$get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ \ [] = None|$
$get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ (v\#xs) =$
$(if\ b\ v > \ \ \varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ v)) < PInfty\ then$
$Some\ v\ else$
$get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs)$

definition $get\text{-}source\text{-}for\text{-}target\text{-}aux\ connections\ b\ \gamma\ xs =$
$the\ (get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs)$

definition $get\text{-}source\ state = get\text{-}source\text{-}aux\text{-}aux$
$(\lambda\ v.\ abstract\text{-}real\text{-}map\ (bal\text{-}lookup\ (balance\ state))\ v)\ (current\text{-}\gamma\ state)\ vs$

definition $get\text{-}target\ state = get\text{-}target\text{-}aux\text{-}aux$
$(\lambda\ v.\ abstract\text{-}real\text{-}map\ (bal\text{-}lookup\ (balance\ state))\ v)\ (current\text{-}\gamma\ state)\ vs$

definition $pair\text{-}to\text{-}realising\text{-}redge\text{-}forward\ state=$
$(\lambda e.\ prod.fst\ (get\text{-}edge\text{-}and\text{-}costs\text{-}forward$

(*abstract-bool-map* (*not-blocked-lookup* (*not-blocked state*)))
(*abstract-real-map* (*flow-lookup* (*current-flow state*))) (*prod.fst e*) (*prod.snd e*)))

definition *get-source-target-path-a state s* =
(*let bf* = *bellman-ford-forward* (*abstract-bool-map* (*not-blocked-lookup* (*not-blocked state*)))
(*abstract-real-map* (*flow-lookup* (*current-flow state*))) *s*
in case (*get-target-for-source-aux-aux bf*
(λ *v. abstract-real-map* (*bal-lookup* (*balance state*)) *v*)
(*current-*γ *state*) *vs*) *of*
Some t $\Rightarrow$ (*let Pbf* = *search-rev-path-exec s bf t Nil*;
P = (*map* (*pair-to-realising-redge-forward state*)
(*edges-of-vwalk Pbf*))
in Some (*t*, *P*))|
None $\Rightarrow$ *None*)

definition *pair-to-realising-redge-backward state* =
(λ*e. prod.fst* (*get-edge-and-costs-backward*
(*abstract-bool-map* (*not-blocked-lookup* (*not-blocked state*)))
(*abstract-real-map* (*flow-lookup* (*current-flow state*))) (*prod.snd e*)
(*prod.fst e*)))

definition *get-source-target-path-b state t* =
(*let bf* = *bellman-ford-backward* (*abstract-bool-map* (*not-blocked-lookup* (*not-blocked state*)))
(*abstract-real-map* (*flow-lookup* (*current-flow state*))) *t*
in case (*get-source-for-target-aux-aux bf*
(λ *v. abstract-real-map* (*bal-lookup* (*balance state*)) *v*)
(*current-*γ *state*) *vs*) *of*
Some s $\Rightarrow$ *let Pbf* =*itrev* (*search-rev-path-exec t bf s Nil*);
P = (*map* (*pair-to-realising-redge-backward state*)
(*edges-of-vwalk Pbf*))
in Some (*s*, *P*) |
None $\Rightarrow$ *None*)

fun *test-all-vertices-zero-balance-aux* **where**
test-all-vertices-zero-balance-aux b Nil = *True*|
test-all-vertices-zero-balance-aux b (*x#xs*) = (*b x* = *0* $\wedge$ *test-all-vertices-zero-balance-aux b xs*)

definition *test-all-vertices-zero-balance state-impl* =
test-all-vertices-zero-balance-aux (λ *x. abstract-real-map* (*bal-lookup* (*balance state-impl*)) *x*) *vs*

definition *ees* = *to-list* $\mathcal{E}$*-impl*
definition *init-flow* = *foldr* (λ *x fl. flow-update x 0 fl*) *ees flow-empty*
definition *init-bal* = b-*impl*
definition *init-rep-card* = *foldr* (λ *x fl. rep-comp-update x* (*x*,*1*) *fl*) *vs rep-comp-empty*
definition *init-not-blocked* = *foldr* (λ *x fl. not-blocked-update x False fl*) *ees*

not-blocked-empty
end

lemmas *Set-Choose* = *Set-Choose-satisfied*

global-interpretation
Adj-Map-Specs2: *Adj-Map-Specs2 lookup t-set sel edge-map-update adj-inv vset-empty vset-delete vset-insert vset-inv isin*
defines *neighbourhood*= *Adj-Map-Specs2.neighbourhood*
using *Set-Choose*
by(*auto intro*: *Adj-Map-Specs2.intro*
simp add: *RBT-Set.empty-def* *Adj-Map-Specs2-def* *Map′-def*
Pair-Graph-Specs-satisfied.adjmap.map-update
Pair-Graph-Specs-satisfied.adjmap.invar-update)

lemmas *Adj-Map-Specs2* = *Adj-Map-Specs2.Adj-Map-Specs2-axioms*

lemma *invar-filter*: $\llbracket$ *set-invar s1*$\rrbracket \Longrightarrow$ *set-invar*(*filter P s1*)
by (*simp add*: *set-invar-def*)

lemma *set-get*:
$\llbracket$*set-invar s1*; $\exists$ *x*. *x* $\in$ *to-set s1* $\wedge$ *P x* $\rrbracket \Longrightarrow \exists$ *y*. *get-from-set P s1* = *Some y*
$\llbracket$ *set-invar s1*; *get-from-set P s1* = *Some x*$\rrbracket \Longrightarrow$ *x* $\in$ *to-set s1*
$\llbracket$ *set-invar s1*; *get-from-set P s1* = *Some x*$\rrbracket \Longrightarrow$ *P x*
$\llbracket$ *set-invar s1*; $\bigwedge$ *x*. *x* $\in$ *to-set s1* $\Longrightarrow$ *P x* = *Q x*$\rrbracket$
$\Longrightarrow$ *get-from-set P s1* = *get-from-set Q s1*
using *find-Some-iff*[*of P s1 x*] *find-cong*[*OF refl*, *of s1 P Q*] *find-None-iff*[*of P s1*]
by (*auto simp add*: *get-from-set-def set-invar-def to-set-def*)

lemma *are-all*: $\llbracket$*set-invar S*$\rrbracket \Longrightarrow$ *are-all P S* $\longleftrightarrow$ ($\forall$ *x* $\in$ *to-set S*. *P x*)
unfolding *to-set-def set-invar-def*
by(*induction S*) *auto*

interpretation *Set-with-predicate*: *Set-with-predicate get-from-set filter are-all set-invar to-set*
using *set-filter invar-filter* *set-get*(*1*,*2*)
by (*auto intro*!: *filter-cong Set-with-predicate.intro intro*: *set-get*(*3*−) *set-filter simp add*: *are-all to-set-def*)
fastforce+

lemmas *Set-with-predicate* = *Set-with-predicate.Set-with-predicate-axioms*

interpretation *bal-map*: *Map*
where *empty* = *bal-empty* **and** *update*=*bal-update* **and** *lookup*= *bal-lookup* **and**
delete= *bal-delete* **and** *invar* = *bal-invar*
using *RBT-Map.M.Map-axioms*
by(*auto simp add*: *bal-update-def bal-empty-def* *bal-delete-def*
bal-lookup-def bal-invar-def *M.invar-def Map-def rbt-red-def rbt-def*)

lemmas *Map-bal* = *bal-map.Map-axioms*

interpretation *Map-conv*: *Map conv-empty conv-update conv-delete conv-lookup conv-invar*
  using *RBT-Map.M.Map-axioms*
  by(*auto simp add*: *conv-update-def conv-empty-def conv-delete-def*
      conv-lookup-def conv-invar-def M.invar-def Map-def rbt-red-def rbt-def)

lemmas *Map-conv* = *Map-conv.Map-axioms*

interpretation *flow-map*: *Map*
  where *empty* = *flow-empty* **and** *update*=*flow-update* **and** *lookup*= *flow-lookup* **and**
    delete= *flow-delete* **and** *invar* = *flow-invar*
  using *RBT-Map.M.Map-axioms*
  by(*auto simp add*: *flow-update-def flow-empty-def flow-delete-def*
      flow-lookup-def flow-invar-def M.invar-def Map-def rbt-red-def rbt-def)

lemmas *Map-flow* = *flow-map.Map-axioms*

interpretation *Map-not-blocked*:
  Map not-blocked-empty not-blocked-update not-blocked-delete not-blocked-lookup not-blocked-invar
  using *RBT-Map.M.Map-axioms*
  by(*auto simp add*: *not-blocked-update-def not-blocked-empty-def not-blocked-delete-def*
      not-blocked-lookup-def not-blocked-invar-def M.invar-def Map-def
      rbt-red-def rbt-def)

lemmas *Map-not-blocked* = *Map-not-blocked.Map-axioms*

interpretation *Map-rep-comp*: *Map rep-comp-empty rep-comp-update rep-comp-delete rep-comp-lookup rep-comp-invar*
  using *RBT-Map.M.Map-axioms*
  by(*auto simp add*: *rep-comp-update-def rep-comp-empty-def rep-comp-delete-def*
      rep-comp-lookup-def rep-comp-invar-def M.invar-def Map-def rbt-red-def rbt-def)

lemmas *Map-rep-comp* = *Map-rep-comp.Map-axioms*

global-interpretation *selection-functions*: *function-generation*
  where *realising-edges-empty*= *realising-edges-empty*
    and *realising-edges-update*=*realising-edges-update*
    and *realising-edges-delete*=*realising-edges-delete*
    and *realising-edges-lookup*= *realising-edges-lookup*
    and *realising-edges-invar*= *realising-edges-invar*

and *bal-empty=bal-empty*
and *bal-update=bal-update*
and *bal-delete= bal-delete*
and *bal-lookup=bal-lookup*
and *bal-invar=bal-invar*

and *flow-empty=flow-empty*
and *flow-update=flow-update*
and *flow-delete=flow-delete*
and *flow-lookup=flow-lookup*
and *flow-invar=flow-invar*

and *not-blocked-empty=not-blocked-empty*
and *not-blocked-update=not-blocked-update*
and *not-blocked-delete=not-blocked-delete*
and *not-blocked-lookup=not-blocked-lookup*
and *not-blocked-invar= not-blocked-invar*

and *rep-comp-empty = rep-comp-empty*
and *rep-comp-update = rep-comp-update*
and *rep-comp-delete = rep-comp-delete*
and *rep-comp-lookup = rep-comp-lookup*
and *rep-comp-invar = rep-comp-invar*

and *to-list=to-list*
and *fst=fst*
and *snd=snd*
and *create-edge=create-edge*
and *to-set = to-set*
and $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl*
and b-*impl* = b-*impl*
and c-*impl* = c-*impl*
and *c-lookup = c-lookup*
for *fst snd create-edge* $\mathcal{E}$*-impl* b-*impl* c-*impl* **and**
c-lookup::*'c-impl* ⇒ *'edge-type*::*linorder* ⇒ *real option*
defines *get-source-target-path-a= selection-functions.get-source-target-path-a*
and *get-source-target-path-b = selection-functions.get-source-target-path-b*
and *get-source = selection-functions.get-source*
and *get-target = selection-functions.get-target*
and *test-all-vertices-zero-balance = selection-functions.test-all-vertices-zero-balance*

and *init-flow = selection-functions.init-flow*
and *init-bal = selection-functions.init-bal*
and *init-rep-card = selection-functions.init-rep-card*
and *init-not-blocked = selection-functions.init-not-blocked*
and *N = selection-functions.N*
and *get-path = selection-functions.get-path*
and *get-target-for-source-aux-aux=selection-functions.get-target-for-source-aux-aux*
and *get-source-for-target-aux-aux = selection-functions.get-source-for-target-aux-aux*

and *get-edge-and-costs-backward* = *selection-functions.get-edge-and-costs-backward*
and *get-edge-and-costs-forward* = *selection-functions.get-edge-and-costs-forward*
and *bellman-ford-backward* =*selection-functions.bellman-ford-backward*
and *bellman-ford-forward* = *selection-functions.bellman-ford-forward*
and *pair-to-realising-redge-forward*=*selection-functions.pair-to-realising-redge-forward*
and *pair-to-realising-redge-backward*=*selection-functions.pair-to-realising-redge-backward*
and *get-target-aux-aux* = *selection-functions.get-target-aux-aux*
and *get-source-aux-aux* = *selection-functions.get-source-aux-aux*
and *ees* =*selection-functions.ees*
and *vs* = *selection-functions.vs*
and *find-cheapest-backward*=*selection-functions.find-cheapest-backward*
and *find-cheapest-forward* = *selection-functions.find-cheapest-forward*
and *realising-edges* = *selection-functions.realising-edges*
and ε = *selection-functions.*ε
and *es* = *selection-functions.es*
and *realising-edges-general* = *selection-functions.realising-edges-general*
and $\mathcal{E}$*-list* = *selection-functions.*$\mathcal{E}$*-list*
and u = *selection-functions.*u
and c = *selection-functions.*c
and $\mathcal{E}$ = *selection-functions.*$\mathcal{E}$
and b = *selection-functions.*b
by(*auto intro*!: *function-generation.intro*
simp add: *Map-realising-edges Map-bal Map-flow Map-not-blocked Map-rep-comp*)

lemmas *function-generation* = *selection-functions.function-generation-axioms*

global-interpretation *orlins-spec*: *orlins-spec*
where *edge-map-update* =*edge-map-update*
and *vset-empty* = $\emptyset_N$
and *vset-delete* = *vset-delete*
and *vset-insert*=*vset-insert*
and *vset-inv* = *vset-inv*
and *isin* = *isin*
and *get-from-set* = *get-from-set*
and *filter* = *filter*
and *are-all* = *are-all*
and *set-invar* = *set-invar*
and *to-set* = *to-set*
and *lookup* = *lookup*
and *t-set* = *t-set*
and *sel*=*sel*
and *adjmap-inv* = *adj-inv*
and *empty-forest*= *map-empty*
and *get-path* = *get-path*

and *flow-empty* = *flow-empty*
and *flow-update* = *flow-update*
and *flow-delete* = *flow-delete*
and *flow-lookup*= *flow-lookup*

and *flow-invar* = *flow-invar*

and *bal-empty* = *bal-empty*
and *bal-update*=*bal-update*
and *bal-delete* = *bal-delete*
and *bal-lookup* =*bal-lookup*
and *bal-invar*=*bal-invar*

and *rep-comp-empty*=*rep-comp-empty*
and *rep-comp-update* =*rep-comp-update*
and *rep-comp-delete*=*rep-comp-delete*
and *rep-comp-lookup*=*rep-comp-lookup*
and *rep-comp-invar*=*rep-comp-invar*

and *conv-empty* =*conv-empty*
and *conv-update* = *conv-update*
and *conv-delete* = *conv-delete*
and *conv-lookup*=*conv-lookup*
and *conv-invar* = *conv-invar*

and *not-blocked-update*=*not-blocked-update*
and *not-blocked-empty*=*not-blocked-empty*
and *not-blocked-delete*=*not-blocked-delete*
and *not-blocked-lookup*=*not-blocked-lookup*
and *not-blocked-invar*= *not-blocked-invar*

and *rep-comp-upd-all* = *rep-comp-upd-all*
and *not-blocked-upd-all* = *not-blocked-upd-all*
and *flow-update-all* = *flow-update-all*
and *get-max* = *get-max*

and *get-source-target-path-a*=
get-source-target-path-a fst snd create-edge $\mathcal{E}$*-impl* c*-impl c-lookup*
and *get-source-target-path-b* =
get-source-target-path-b fst snd create-edge $\mathcal{E}$*-impl* c*-impl c-lookup*
and *get-source* = *get-source fst snd* $\mathcal{E}$*-impl*
and *get-target* = *get-target fst snd* $\mathcal{E}$*-impl*
and *test-all-vertices-zero-balance* = *test-all-vertices-zero-balance fst snd* $\mathcal{E}$*-impl*

and *init-flow* = *init-flow* $\mathcal{E}$*-impl*
and *init-bal* = *init-bal* b*-impl*
and *init-rep-card* = *init-rep-card fst snd* $\mathcal{E}$*-impl*
and *init-not-blocked* = *init-not-blocked* $\mathcal{E}$*-impl*

and *N* = *N fst snd* $\mathcal{E}$*-impl*
and *snd* = *snd*
and *fst* = *fst*
and *create-edge* = *create-edge*
and c = c c*-impl c-lookup*

and $\mathcal{E} = \mathcal{E}$ $\mathcal{E}$-*impl*
and $\mathcal{E}$-*impl* = $\mathcal{E}$-*impl*
and u = u
and b = b b-*impl*
and $\varepsilon = \varepsilon$
for *fst snd create-edge* $\mathcal{E}$-*impl* b-*impl* c-*impl* *c-lookup* ε

defines *initial* = *orlins-spec.initial*
and *orlins* = *orlins-spec.orlins*
and *send-flow* = *orlins-spec.send-flow*
and *maintain-forest* =*orlins-spec.maintain-forest*
and *augment-edge* = *orlins-spec.augment-edge*
and *augment-edges* = *orlins-spec.augment-edges*
and *orlins-impl* = *orlins-spec.orlins-impl*
and *send-flow-impl* = *orlins-spec.send-flow-impl*
and *maintain-forest-impl* =*orlins-spec.maintain-forest-impl*
and *augment-edge-impl* = *orlins-spec.augment-edge-impl*
and *augment-edges-impl* = *orlins-spec.augment-edges-impl*
and *to-redge-path* = *orlins-spec.to-redge-path*
and *add-direction* = *orlins-spec.add-direction*
and *move-balance* = *orlins-spec.move-balance*
and *move*= *orlins-spec.move*
and *insert-undirected-edge* = *orlins-spec.insert-undirected-edge*
and *abstract-conv-map-i* = *orlins-spec.abstract-conv-map*
and *abstract-not-blocked-map-i* = *orlins-spec.abstract-not-blocked-map*
and *abstract-rep-map* = *orlins-spec.abstract-rep-map*
and *abstract-comp-map* = *orlins-spec.abstract-comp-map*
and *abstract-flow-map-i* = *orlins-spec.abstract-flow-map*
and *abstract-bal-map-i* = *orlins-spec.abstract-bal-map*
and *new*-γ= *orlins-spec.new*-γ
and *make-pair* = *orlins-spec.make-pair*
and *neighbourhood'* = *orlins-spec.neighbourhood'*
using *Map-bal Map-conv Map-flow Map-not-blocked Map-rep-comp*
by(*auto intro*!: *orlins-spec.intro algo-spec.intro maintain-forest-spec.intro rep-comp-upd-all*
send-flow-spec.intro maintain-forest-spec.intro flow-update-all get-max
not-blocked-upd-all
map-update-all.intro map-update-all-axioms.intro
simp add: *Set-with-predicate Adj-Map-Specs2*)

lemmas [*code*] = *orlins-spec.orlins-impl.simps*[*folded orlins-impl-def*]

lemmas *orlins-spec* = *orlins-spec.orlins-spec-axioms*
lemmas *send-flow-spec* = *orlins-spec.send-flow-spec-axioms*
lemmas *algo-spec* = *orlins-spec.algo-spec-axioms*
lemmas *maintain-forest-spec* = *orlins-spec.maintain-forest-spec-axioms*

0.1.2 Proofs

lemma *set-filter*:

⟦ *set-invar s1* ⟧ $\Longrightarrow$ *to-set*(*filter P s1*) = *to-set s1* − {*x*. *x* ∈ *to-set s1* ∧ ¬ *P x*}
⟦ *set-invar s1*; ⋀ *x*. *x* ∈ *to-set s1* $\Longrightarrow$ *P x* = *Q x* ⟧ $\Longrightarrow$ *filter P s1* = *filter Q s1*
using *filter-cong*[*OF refl*, *of s1 P Q*]
by (*auto simp add*: *set-invar-def to-set-def*)

lemma *flow-invar-Leaf*: *flow-invar Leaf*
by (*metis RBT-Set.empty-def flow-empty-def flow-map.invar-empty*)

lemma *flow-invar-fold*:
⟦*flow-invar T*; (⋀ *T e*. *flow-invar T* $\Longrightarrow$ *flow-invar* (*f e T*))⟧
$\Longrightarrow$ *flow-invar* (*foldr* (λ *e tree*. *f e tree*) *list T*)
by(*induction list*) *auto*

lemma *dom-fold*:
flow-invar T $\Longrightarrow$
dom (*flow-lookup* (*foldr* (λ *e tree*. *flow-update* (*f e*) (*g e*) *tree*) *AS T*))
= *dom* (*flow-lookup T*) ∪ *f* ' *set AS*
by(*induction AS*)
(*auto simp add*: *flow-map.map-update flow-invar-fold flow-map.invar-update*)

lemma *fold-lookup*:
⟦*flow-invar T*; *bij f*⟧
$\Longrightarrow$ *flow-lookup* (*foldr* (λ *e tree*. *flow-update* (*f e*) (*g e*) *tree*) *AS T*) *x*
= (*if inv f x* ∈ *set AS then Some* (*g* (*inv f x*)) *else flow-lookup T x*)
apply(*induction AS*)
subgoal by *auto*
apply(*subst foldr-Cons*, *subst o-apply*)
apply(*subst flow-map.map-update*)
apply(*subst flow-invar-fold*)
apply *simp*
apply(*rule flow-map.invar-update*)
apply *simp*
apply *simp*
subgoal for *a AS*
using *bij-inv-eq-iff bij-betw-inv-into-left*
by (*fastforce intro*: *flow-map.invar-update simp add*: *bij-betw-def*)
done

interpretation *bal-map*: *Map* **where** *empty* = *bal-empty* **and** *update*=*bal-update* **and** *lookup*= *bal-lookup*
and *delete*= *bal-delete* **and** *invar* = *bal-invar*
using *Map-bal* **by** *auto*

lemma *bal-invar-fold*:
bal-invar bs $\Longrightarrow$ *bal-invar* (*foldr* (λ*xy tree*. *bal-update* (*g xy*) (*f xy tree*) *tree*) *ES bs*)
by(*induction ES*)(*auto simp add*: *bal-map.invar-update*)

lemma *bal-dom-fold*:

bal-invar bs $\Longrightarrow$
dom (*bal-lookup* (*foldr* (λ*xy tree*. *bal-update* (*g xy*) (*f xy tree*) *tree*) *ES bs*))
= *dom* (*bal-lookup bs*) $\cup$ *g* ‘ (*set ES*)
apply(*induction ES*)
subgoal
by *auto*
by(*simp add*: *dom-def*, *subst bal-map.map-update*) (*auto intro*: *bal-invar-fold*)

interpretation *rep-comp-map*:
Map **where** *empty* = *rep-comp-empty* **and** *update*=*rep-comp-update*
and *lookup*= *rep-comp-lookup* **and** *delete*= *rep-comp-delete* **and** *invar* =
rep-comp-invar
using *Map-rep-comp* **by** *auto*

interpretation *conv-map*:
Map **where** *empty* = *conv-empty* **and** *update*=*conv-update* **and** *lookup*= *conv-lookup*
and *delete*= *conv-delete* **and** *invar* = *conv-invar*
using *Map-conv* **by** *auto*

interpretation *not-blocked-map*:
Map **where** *empty* = *not-blocked-empty* **and** *update*=*not-blocked-update* **and**
lookup= *not-blocked-lookup*
and *delete*= *not-blocked-delete* **and** *invar* = *not-blocked-invar*
using *Map-not-blocked* **by** *auto*

lemma *bal-invar-b*:*bal-invar* (*foldr* (λ *xy tree*. *update* (*prod.fst xy*) (*prod.snd xy*)
tree) *xs Leaf*)
by(*induction xs*)
(*auto simp add*: *invc-upd*(*2*) *update-def invh-paint invh-upd*(*1*) *color-paint-Black*
bal-invar-def M.invar-def rbt-def rbt-red-def inorder-paint inorder-upd

sorted-upd-list)

lemma *dom-update-insert*:*M.invar T* $\Longrightarrow$ *dom* (*lookup* (*update x y T*)) = *Set.insert*
x (*dom* (*lookup T*))
by(*auto simp add*: *M.map-update*[*simplified update-def*] *update-def dom-def*)

lemma *M-invar-fold*:*M.invar* (*foldr* (λ *xy tree*. *update* (*prod.fst xy*) (*prod.snd xy*)
tree) *list Leaf*)
by(*induction list*) (*auto intro*: *M.invar-update M.invar-empty*[*simplified RBT-Set.empty-def*])

lemma *M-dom-fold*: *dom* (*lookup* (*foldr* (λ *xy tree*. *update* (*prod.fst xy*) (*prod.snd*
xy) *tree*) *list Leaf*))
= *set* (*map prod.fst list*)
by(*induction list*)(*auto simp add*: *dom-update-insert*[*OF M-invar-fold*])

hide-const *RBT.B*

locale *function-generation-proof* =

function-generation **where**
to-set = *to-set*:: $'$*edge-type-set-impl* $\Rightarrow$ $'$*edge-type set*
and *fst* = *fst* :: ($'$*edge-type*::*linorder*) $\Rightarrow$ ($'$*a*::*linorder*)
and *snd* = *snd* :: ($'$*edge-type*::*linorder*) $\Rightarrow$ $'$*a*
and *not-blocked-update* = *not-blocked-update* ::$'$*edge-type* $\Rightarrow$ *bool* $\Rightarrow$ $'$*not-blocked-impl*$\Rightarrow$ $'$*not-blocked-impl*
and *flow-update* = *flow-update*::$'$*edge-type* $\Rightarrow$ *real* $\Rightarrow$ $'$*f-impl* $\Rightarrow$ $'$*f-impl*
and *bal-update* = *bal-update*:: $'$*a* $\Rightarrow$ *real* $\Rightarrow$ $'$*b-impl* $\Rightarrow$ $'$*b-impl*
and *rep-comp-update*=*rep-comp-update*:: $'$*a* $\Rightarrow$ $'$*a* $\times$ *nat* $\Rightarrow$ $'$*r-comp-impl*$\Rightarrow$ $'$*r-comp-impl*+

Set-with-predicate **where**
get-from-set=*get-from-set*::($'$*edge-type* $\Rightarrow$ *bool*) $\Rightarrow$ $'$*edge-type-set-impl* $\Rightarrow$ $'$*edge-type option*
and *to-set* =*to-set* +

multigraph: *multigraph fst snd create-edge* $\mathcal{E}$+

Set-with-predicate: *Set-with-predicate get-from-set filter are-all set-invar to-set* +

rep-comp-maper: *Map rep-comp-empty rep-comp-update*::$'$*a* $\Rightarrow$ ($'$*a* $\times$ *nat*) $\Rightarrow$ $'$*r-comp-impl* $\Rightarrow$ $'$*r-comp-impl*
rep-comp-delete rep-comp-lookup rep-comp-invar +

conv-map: *Map conv-empty conv-update*::$'$*a* $\times$ $'$*a* $\Rightarrow$ $'$*edge-type Redge* $\Rightarrow$ $'$*conv-impl* $\Rightarrow$ $'$*conv-impl*
conv-delete conv-lookup conv-invar +

not-blocked-map: *Map not-blocked-empty not-blocked-update*::$'$*edge-type* $\Rightarrow$ *bool* $\Rightarrow$ $'$*not-blocked-impl*$\Rightarrow$ $'$*not-blocked-impl*
not-blocked-delete not-blocked-lookup not-blocked-invar +

rep-comp-iterator: *Map-iterator rep-comp-invar rep-comp-lookup rep-comp-upd-all*+

flow-iterator: *Map-iterator flow-invar flow-lookup flow-update-all*+

not-blocked-iterator: *Map-iterator not-blocked-invar not-blocked-lookup not-blocked-upd-all*
for *get-from-set*
to-set
fst snd
rep-comp-update
conv-empty
conv-delete
conv-lookup
conv-invar
conv-update
not-blocked-update

flow-update
bal-update
rep-comp-upd-all
flow-update-all
not-blocked-upd-all +

fixes *get-max*::($'a \Rightarrow real \Rightarrow real) \Rightarrow 'b\text{-}impl \Rightarrow real$
assumes $\mathcal{E}$*-impl-invar*: *set-invar* $\mathcal{E}$*-impl*
and *invar-b-impl*: *bal-invar* b-*impl*
and *b-impl-dom*: *dVs* (*make-pair* ' *to-set* $\mathcal{E}$*-impl*) = *dom* (*bal-lookup* b-*impl*)
and *N-gre-0*: $N > 0$
and *get-max*: $\bigwedge$ *b f*. ⟦*bal-invar b*; *dom* (*bal-lookup b*) $\neq$ {}⟧
$\Longrightarrow$ *get-max f b* = *Max* {*f y* (*the* (*bal-lookup b y*)) | *y*. *y* $\in$ *dom* (*bal-lookup b*)}
and *to-list*: $\bigwedge$ *E*. *set-invar E* $\Longrightarrow$ *to-set E* = *set* (*to-list E*)
$\bigwedge$ *E*. *set-invar E* $\Longrightarrow$ *distinct* (*to-list E*)
begin

lemmas *rep-comp-upd-all* = *rep-comp-iterator.update-all*
lemmas *flow-update-all* = *flow-iterator.update-all*
lemmas *not-blocked-upd-all* = *not-blocked-iterator.update-all*

notation *vset-empty* ($\emptyset_N$)

lemma *make-pairs-are*:*multigraph.make-pair* = *make-pair*
multigraph-spec.make-pair fst snd = *make-pair*
by(*auto intro*!: *ext*
simp add: *make-pair-def multigraph.make-pair-def multigraph-spec.make-pair-def*)

lemmas *create-edge* = *local.multigraph.create-edge*[*simplified make-pairs-are*(*1*)]

lemma *vs-are*: *dVs* (*make-pair* ' $\mathcal{E}$) = *set* (*map fst* $\mathcal{E}$*-list*) $\cup$ *set* (*map snd* $\mathcal{E}$*-list*)
using *multigraph.make-pair*[*OF refl refl*] *to-list* $\mathcal{E}$*-impl-invar*
by(*auto simp add*: $\mathcal{E}$*-def* $\mathcal{E}$*-list-def dVs-def make-pairs-are*)

lemma $\mathcal{E}$*-impl*: *set-invar* $\mathcal{E}$*-impl* $\exists$ *e*. *e* $\in$ (*to-set* $\mathcal{E}$*-impl*) *finite* $\mathcal{E}$
and *b-impl*: *bal-invar* b-*impl dVs* (*make-pair* ' (*to-set* $\mathcal{E}$*-impl*)) = *dom* (*bal-lookup* b-*impl*)
and ε*-axiom*: $0 < \varepsilon$ $\varepsilon \leq 1 / 2$ $\varepsilon \leq 1/$ (*real* (*card* (*multigraph*.$\mathcal{V}$))) $\varepsilon < 1/2$
proof(*goal-cases*)
case *1*
then show *?case*
by (*simp add*: $\mathcal{E}$*-impl-invar*)
next
case *2*
then show *?case*
using *local*.$\mathcal{E}$*-def local.multigraph.E-not-empty* **by** *auto*
next
case *3*

then show *?case*
by (*simp add*: *local.multigraph.finite-E*)
next
case *4*
then show *?case*
using *invar-b-impl* **by** *auto*
next
case *5*
then show *?case*
using *b-impl-dom* **by** *auto*
next
case *6*
then show *?case*
using *$\mathcal{E}$-impl-invar local.$\mathcal{E}$-def local.multigraph.E-not-empty local.to-list*(*1*)
by (*auto simp add*: *ε-def N-def*)
next
case *7*
then show *?case*
by(*auto simp add*: *ε-def*)
next
case *8*
then show *?case*
using *vs-are N-gre-0*
by(*auto simp add*: *ε-def N-def to-set-def vs-are insert-commute $\mathcal{E}$-list-def length-remdups-card-conv frac-le make-pairs-are*)
next
case *9*
then show *?case*
by(*auto simp add*: *ε-def*)
qed

lemma *N-def'*: *N* =*card VV*
using *$\mathcal{E}$-impl-invar local.to-list*(*1*)
by(*auto intro*!: *cong*[*of card*, *OF refl*]
simp add: *N-def dVs-def $\mathcal{E}$-def to-set-def length-remdups-card-conv make-pair-def*)

lemma *$\mathcal{E}$-impl-basic*: *set-invar $\mathcal{E}$-impl* $\exists$ *e*. *e* $\in$ (*to-set $\mathcal{E}$-impl*) *finite $\mathcal{E}$*
using *$\mathcal{E}$-impl* **by** *auto*

interpretation *cost-flow-network*:
cost-flow-network **where** $\mathcal{E}$ = $\mathcal{E}$ **and** c = c **and** u = u
and *fst* = *fst* **and** *snd* = *snd* **and** *create-edge* = *create-edge*
using *$\mathcal{E}$-def multigraph.multigraph-axioms*
by(*auto simp add*: u-*def cost-flow-network-def flow-network-axioms-def flow-network-def*)

lemmas *cost-flow-network*[*simp*] = *cost-flow-network.cost-flow-network-axioms*

abbreviation $\mathfrak{c}$ $\equiv$ *cost-flow-network*.$\mathfrak{c}$

```
abbreviation to-edge == cost-flow-network.to-vertex-pair
abbreviation oedge == flow-network-spec.oedge
abbreviation rcap == cost-flow-network.rcap

lemma make-pair-fst-snd: make-pair e = (fst e, snd e)
  using multigraph.make-pair′ make-pairs-are by simp

lemma es-is-E:set es = make-pair ‘ ℰ ∪ {(u, v) | u v.  (v, u) ∈ make-pair ‘ ℰ}
  using to-list(1)[OF ℰ-impl-basic(1)]
  by(auto simp add: es-def to-list-def ℰ-def  make-pair-fst-snd)

lemma vs-is-V: set vs = VV
proof–
 have 1:x ∈ prod.fst ‘ (make-pair ‘ local.ℰ ∪ {(u, v). (v, u) ∈ make-pair ‘ local.ℰ})
⟹
         x ∈ local.multigraph.𝒱 for x
  proof(goal-cases)
    case 1
  then obtain e where e-pr:x = prod.fst e
                e ∈ make-pair ‘ ℰ ∪ {(u, v). (v, u) ∈ make-pair ‘ ℰ} by auto
  hence aa:e ∈ make-pair ‘ ℰ ∨ prod.swap e ∈ make-pair ‘ ℰ by auto
  show ?case
  proof–
    obtain pp  where
      f1: ∀ X1. pp X1 = (fst X1, snd X1)
      by moura
    then have e ∈ pp ‘ ℰ ∨ prod.swap e ∈ pp ‘ ℰ
      using aa make-pair-fst-snd by auto
    then have prod.fst e ∈ dVs (pp ‘ ℰ)
      by(auto intro: dVsI′(1) dVsI(2) simp add: prod.swap-def)
    then have prod.fst e ∈ dVs ((λe. (fst e, snd e)) ‘ ℰ)
      using f1 by force
    thus x ∈ local.multigraph.𝒱
      by (auto simp add: e-pr(1) make-pair-fst-snd make-pairs-are)
  qed
qed
 have 2:x ∈ local.multigraph.𝒱 ⟹
         x ∈ prod.fst ‘ (make-pair ‘ local.ℰ ∪ {(u, v). (v, u) ∈ make-pair ‘ local.ℰ})
for x
  proof(goal-cases)
    case 1
    note 2 = this
  then obtain a b where x ∈{a,b} (a,b) ∈ make-pair ‘ ℰ
    by (metis in-dVsE(1) insert-iff make-pairs-are)
  then show ?case by force
qed
  show ?thesis
    by(fastforce intro: 1 2 simp add: vs-def  es-is-E  dVsI′ make-pairs-are)
qed
```

lemma *vs-and-es*: $vs \neq []$ *set vs* = *dVs* (*set es*) *distinct vs distinct es*
  using $\mathcal{E}$*-def es-is-E vs-def vs-is-V es-is-E* $\mathcal{E}$*-impl-basic*
  by (*auto simp add*: *vs-def es-def dVs-def*)

definition *no-cycle-cond* = ($\neg$ *has-neg-cycle make-pair* (*function-generation*.$\mathcal{E}$ $\mathcal{E}$*-impl to-set*) c)

context
  assumes *no-cycle-cond*: *no-cycle-cond*
begin

lemma *conservative-weights*:
$\nexists$ *C*. *closed-w* (*make-pair* ' $\mathcal{E}$) (*map make-pair C*) $\wedge$ (*set C* $\subseteq \mathcal{E}$) $\wedge$ *foldr* (λ *e acc*. *acc* + c *e*) *C 0* < *0*
  using *no-cycle-cond*
  by(*auto simp add*: *no-cycle-cond-def has-neg-cycle-def*
      ab-semigroup-add-class.add.commute[*of* - c -])

thm *algo-axioms-def*

lemma *algo-axioms*: *algo-axioms snd* u c $\mathcal{E}$ *set-invar to-set lookup adj-inv*
      ε $\mathcal{E}$*-impl map-empty N fst*
  using ε*-axiom* $\mathcal{E}$*-impl*(*1*) *no-cycle-cond*
  by(*auto intro*!: *algo-axioms.intro*
    simp add: u*-def* $\mathcal{E}$*-def N-def'* *Pair-Graph-Specs-satisfied.adjmap.map-empty*
        Pair-Graph-Specs-satisfied.adjmap.invar-empty make-pairs-are
no-cycle-cond-def)

lemmas *dfs-defs* = *dfs.initial-state-def*

lemma *same-digraph-abses*:
 Adj-Map-Specs2.digraph-abs = *Pair-Graph-Specs-satisfied.digraph-abs*
and *same-neighbourhoods*:
 Adj-Map-Specs2.neighbourhood = *Pair-Graph-Specs-satisfied.neighbourhood*
 by(*auto intro*!: *ext*
   simp add: *Adj-Map-Specs2.digraph-abs-def Pair-Graph-Specs-satisfied.digraph-abs-def*
     Adj-Map-Specs2.neighbourhood-def Pair-Graph-Specs-satisfied.neighbourhood-def)

lemma *maintain-forest-axioms-extended*:
 assumes *maintain-forest-spec.maintain-forest-get-path-cond* $\emptyset_N$ *vset-inv isin lookup*
     t-set adj-inv get-path u v (*E*::
       (($'a \times ('a \times$ *color*) *tree*) $\times$ *color*) *tree*) *p q*
  shows *vwalk-bet* (*Adj-Map-Specs2.digraph-abs E*) *u p v*
    distinct p
proof(*insert maintain-forest-spec.maintain-forest-get-path-cond-unfold-meta*[*OF*
       maintain-forest-spec assms], *goal-cases*)
  case *1*
  note *assms* = *this*

have *graph-invar*: *Pair-Graph-Specs-satisfied.graph-inv E*
and *finite-graph*:*Pair-Graph-Specs-satisfied.finite-graph E*
and *finite-neighbs*:*Pair-Graph-Specs-satisfied.finite-vsets E*
using *assms*(*4*) **by** (*auto elim*!: *Instantiation.Adj-Map-Specs2.good-graph-invarE*)
obtain *e* **where** *e-prop*:$e \in$ (*Adj-Map-Specs2.digraph-abs E*) u = *prod.fst e*
using *assms*(*1*) *assms*(*5*) *no-outgoing-last*
by(*unfold vwalk-bet-def Pair-Graph-Specs-satisfied.digraph-abs-def*) *fastforce*
hence *neighb-u*: *Adj-Map-Specs2.neighbourhood E u* $\neq$ *vset-empty*
using *assms*(*2*)
Pair-Graph-Specs-satisfied.are-connected-absI[*OF - graph-invar*,
of prod.fst e prod.snd e, *simplified*]
by(*auto simp add*: *same-neighbourhoods same-digraph-abses*)
have *q-non-empt*: $q \neq []$
using *assms*(*1*) **by** *auto*
obtain *d* **where** $d \in$ (*Adj-Map-Specs2.digraph-abs E*) v = *prod.snd d*
using *assms*(*1*) *assms*(*5*) *singleton-hd-last'*[*OF q-non-empt*]
vwalk-append-edge[*of - butlast q* [*last q*],*simplified append-butlast-last-id*[*OF q-non-empt*]]
by(*force simp add*: *vwalk-bet-def Adj-Map-Specs2.digraph-abs-def*)
have *u-in-Vs*:$u \in$ *dVs* (*Adj-Map-Specs2.digraph-abs E*)
using *assms*(*1*) *assms*(*2*) **by** *auto*
have *dfs-axioms*: *DFS.DFS-axioms isin t-set adj-inv* $\emptyset_N$ *vset-inv lookup*
E u
using *finite-graph finite-neighbs graph-invar u-in-Vs*
by(*simp only*: *dfs.DFS-axioms-def same-neighbourhoods same-digraph-abses*)
have *dfs-thms*: *DFS-thms map-empty delete vset-insert isin t-set sel update adj-inv vset-empty vset-delete*
vset-inv vset-union vset-inter vset-diff lookup E u
by(*auto intro*!: *DFS-thms.intro DFS-thms-axioms.intro simp add*: *dfs.DFS-axioms dfs-axioms*)
have *dfs-dom*:*DFS.DFS-dom vset-insert sel vset-empty vset-diff lookup*
E v (*dfs-initial u*)
using *DFS-thms.initial-state-props*(*6*)[*OF dfs-thms*]
by(*simp add*: *dfs-initial-def dfs-initial-state-def DFS-thms.initial-state-props*(*6*) *dfs-axioms*)
have *rectified-map-subset*:*dfs.Graph.digraph-abs E* $\subseteq$
(*Adj-Map-Specs2.digraph-abs E*)
by (*simp add*: *assms*(*2*) *same-neighbourhoods same-digraph-abses*)
have *rectified-map-subset-rev*:*Adj-Map-Specs2.digraph-abs E*
$\subseteq$ *dfs.Graph.digraph-abs E*
by (*simp add*: *assms*(*2*) *same-neighbourhoods same-digraph-abses*)
have *reachable*:*DFS-state.return* (*dfs E v* (*dfs-initial u*)) = *Reachable*
proof(*rule ccontr*,*rule DFS.return.exhaust*[*of DFS-state.return* (*dfs E v* (*dfs-initial u*))],*goal-cases*)
case *2*
hence $\nexists p$. *distinct p* $\wedge$ *vwalk-bet* (*dfs.Graph.digraph-abs E*) *u p v*
using *DFS-thms.DFS-correct-1*[*OF dfs-thms*, *of v*] *DFS-thms.DFS-to-DFS-impl*[*OF dfs-thms*, *of v*]
by (*auto simp add*: *dfs-def dfs-initial-def dfs-initial-state-def simp add*:

dfs-impl-def)
moreover obtain *q′* **where** *vwalk-bet* (*Adj-Map-Specs2.digraph-abs E*) *u q′ v distinct q′*
using *vwalk-bet-to-distinct-is-distinct-vwalk-bet*[*OF assms*(*1*)]
by(*auto simp add*: *distinct-vwalk-bet-def*)
moreover hence *vwalk-bet* (*dfs.Graph.digraph-abs E*) *u q′ v*
by (*meson rectified-map-subset-rev vwalk-bet-subset*)
ultimately show *False* **by** *auto*
next
qed *simp*
have *vwalk-bet* (*dfs.Graph.digraph-abs E*)
u (*rev* (*stack* (*dfs E v* (*dfs-initial u*)))) *v*
using *reachable sym*[*OF DFS-thms.DFS-to-DFS-impl*[*OF dfs-thms*, *of v*]]
by(*auto intro*!: *DFS-thms.DFS-correct-2*[*OF dfs-thms*, *of v*]
simp add: *dfs-initial-def dfs-def dfs-axioms dfs-impl-def dfs-initial-state-def*)

thus *vwalk-bet* (*Adj-Map-Specs2.digraph-abs E*) *u p v*
using *rectified-map-subset vwalk-bet-subset assms*(*2*)
by (*simp add*: *local.get-path-def*)
show *distinct p*
using *DFS-thms.DFS-correct-2*(*2*)[*OF dfs-thms*]
using *DFS-thms.initial-state-props*(*1*,*3*)[*OF dfs-thms*]
dfs-dom DFS-thms.DFS-to-DFS-impl[*OF dfs-thms*] *reachable*
by(*auto simp add*: *assms*(*2*) *get-path-def same-neighbourhoods same-digraph-abses*
dfs-def dfs-impl-def dfs-initial-def dfs-initial-state-def)
qed

lemma *flow-map-update-all*:
map-update-all flow-empty flow-update flow-delete flow-lookup flow-invar flow-update-all
using *local.flow-update-all*
by(*fastforce intro*!: *map-update-all.intro map-update-all-axioms.intro*
simp add: *Map-flow.Map-axioms domIff*)

lemma *rep-comp-map-update-all*:
map-update-all rep-comp-empty rep-comp-update rep-comp-delete
rep-comp-lookup rep-comp-invar rep-comp-upd-all
using *local.rep-comp-upd-all*
by(*fastforce intro*!: *map-update-all.intro map-update-all-axioms.intro*
simp add: *Map-rep-comp.Map-axioms domIff Map-axioms*)

lemma *not-blocked-upd-all-locale*:
map-update-all not-blocked-empty not-blocked-update not-blocked-delete
not-blocked-lookup not-blocked-invar not-blocked-upd-all
using *local.not-blocked-upd-all*
by(*fastforce intro*!: *map-update-all.intro map-update-all-axioms.intro*
simp add: *Map-not-blocked.Map-axioms domIff*)

interpretation *algo*: *algo*
where $\mathcal{E} = \mathcal{E}$

and c = c
and u = u
and *edge-map-update* = *edge-map-update*
and *vset-empty* = *vset-empty*
and *vset-delete*= *vset-delete*
and *vset-insert* = *vset-insert*
and *vset-inv* = *vset-inv*
and *isin* = *isin*
and *get-from-set*=*get-from-set*
and *filter*=*filter*
and *are-all*=*are-all*
and *set-invar*=*set-invar*
and *to-set*=*to-set*
and *lookup*=*lookup*
and *t-set*=*t-set*
and *sel*=*sel*
and *adjmap-inv*=*adj-inv*
and $\varepsilon = \varepsilon$
and $\mathcal{E}$*-impl*=$\mathcal{E}$*-impl*
and *empty-forest*=*map-empty*
and b = b **and** $N = N$
and *snd* = *snd*
and *fst* = *fst*
and *create-edge*=*create-edge*

and *flow-empty* = *flow-empty*
and *flow-lookup* = *flow-lookup*
and *flow-update* = *flow-update*
and *flow-delete*=*flow-delete*
and *flow-invar* = *flow-invar*

and *bal-empty* = *bal-empty*
and *bal-lookup* = *bal-lookup*
and *bal-update* = *bal-update*
and *bal-delete*=*bal-delete*
and *bal-invar* = *bal-invar*

and *conv-empty* = *conv-empty*
and *conv-lookup* = *conv-lookup*
and *conv-update* = *conv-update*
and *conv-delete*=*conv-delete*
and *conv-invar* = *conv-invar*

and *rep-comp-empty* = *rep-comp-empty*
and *rep-comp-lookup* = *rep-comp-lookup*
and *rep-comp-update* = *rep-comp-update*
and *rep-comp-delete*=*rep-comp-delete*
and *rep-comp-invar* = *rep-comp-invar*

and *not-blocked-empty* = *not-blocked-empty*
and *not-blocked-lookup* = *not-blocked-lookup*
and *not-blocked-update* = *not-blocked-update*
and *not-blocked-delete*=*not-blocked-delete*
and *not-blocked-invar* = *not-blocked-invar*
using *cost-flow-network*
by(*auto intro*!: *algo.intro algo-spec.intro*
simp add: *Adj-Map-Specs2 algo-axioms algo-def Set-with-predicate-axioms flow-map-update-all*
Map-bal.Map-axioms rep-comp-map-update-all conv-map.Map-axioms not-blocked-upd-all-locale)

lemmas *algo* = *algo.algo-axioms*

lemma *maintain-forest-axioms*:
maintain-forest-axioms $\emptyset_N$ *vset-inv* (*isin*:: ($'a \times$ *color*) *tree* $\Rightarrow$ $'a \Rightarrow$ *bool*) *lookup t-set*
adj-inv local.get-path
by(*auto intro*!: *maintain-forest-axioms.intro maintain-forest-axioms-extended*)

interpretation *maintain-forest*:
Maintain-Forest.maintain-forest snd create-edge u $\mathcal{E}$ c *edge-map-update vset-empty*
vset-delete vset-insert vset-inv isin filter are-all set-invar to-set lookup t-set sel
adj-inv flow-empty flow-update flow-delete flow-lookup flow-invar bal-empty bal-update
bal-delete bal-lookup bal-invar rep-comp-empty rep-comp-update rep-comp-delete rep-comp-lookup
rep-comp-invar conv-empty conv-update conv-delete conv-lookup conv-invar not-blocked-update
not-blocked-empty not-blocked-delete not-blocked-lookup not-blocked-invar rep-comp-upd-all
flow-update-all not-blocked-upd-all b *get-max* ε *N get-from-set map-empty* $\mathcal{E}$*-impl get-path fst*
by(*auto intro*!: *maintain-forest.intro maintain-forest-axioms*
simp add: *algo.algo-spec-axioms maintain-forest-spec-def algo*)

lemma *realising-edges-general-invar*:
realising-edges-invar (*realising-edges-general list*)
unfolding *realising-edges-general-def*
by(*induction list*)
(*auto intro*: *Map-realising.invar-update split*: *option.split*
simp add: *Map-realising.invar-empty realising-edges-empty-def*
realising-edges-invar-def realising-edges-update-def)

lemma *realising-edges-general-dom*:
(u, v) $\in$ *set* (*map make-pair list*)
$\longleftrightarrow$ *realising-edges-lookup* (*realising-edges-general list*) (u, v) $\neq$ *None*
unfolding *realising-edges-general-def*
proof(*induction list*)

```
  case Nil
  then show ?case
   by(simp add: realising-edges-lookup-def realising-edges-empty-def Map-realising.map-empty)
next
  case (Cons e list)
  show ?case
  proof(cases make-pair e = (u, v))
    case True
    show ?thesis
      apply(subst foldr.foldr-Cons, subst o-apply)
      apply(subst realising-edges-general-def[ symmetric])+
      using True
      by(auto intro: Map-realising.invar-update split: option.split
              simp add: Map-realising.map-update realising-edges-update-def realis-
ing-edges-lookup-def
                    realising-edges-general-invar[simplified realising-edges-invar-def])
  next
    case False
    hence in-list:(u, v) ∈ set (map make-pair list)
                 ⟷ (u, v) ∈ set (map make-pair (e#list))
      using make-pair-fst-snd[of e] by auto
    note ih = Cons(1)[simplified in-list]
      show ?thesis
      unfolding Let-def
      using realising-edges-general-invar False
      by(subst foldr.foldr-Cons, subst o-apply,subst ih[simplified Let-def])
        (auto split: option.split
              simp add: realising-edges-update-def realising-edges-lookup-def
                        Map-realising.map-update realising-edges-invar-def
                         realising-edges-general-def[simplified Let-def, symmetric] )
  qed
qed

lemma realising-edges-dom:
    ((u, v) ∈ set (map make-pair ℰ-list)) =
    (realising-edges-lookup realising-edges (u, v) ≠ None)
  using realising-edges-general-dom
  by(fastforce simp add: realising-edges-def)

lemma not-both-realising-edges-none:
(u, v) ∈ set es ⟹ realising-edges-lookup realising-edges (u, v) ≠ None ∨
                     realising-edges-lookup realising-edges (v, u) ≠ None
  using realising-edges-dom make-pair-fst-snd
  by(auto simp add:  es-def ℰ-list-def)

lemma find-cheapest-forward-props:
   assumes (beste, bestc) = find-cheapest-forward f nb list e c
          edges-and-costs = Set.insert (e, c)
              {(e, ereal (c e)) | e. e ∈ set list ∧ nb e ∧ ereal (f e) < u e}
```

shows (*beste*, *bestc*) ∈ *edges-and-costs* ∧
(∀ (*ee*, *cc*) ∈ *edges-and-costs*. *bestc* ≤ *cc*)
using *assms*
unfolding *find-cheapest-forward-def*
by(*induction list arbitrary*: *edges-and-costs beste bestc*)
(*auto split*: *if-split prod.split* ,
insert ereal-le-less nless-le order-less-le-trans, *fastforce*+)

lemma *find-cheapest-backward-props*:
assumes (*beste*, *bestc*) = *find-cheapest-backward f nb list e c*
edges-and-costs = *Set.insert* (*e*, *c*)
{(*e*, *ereal* (− c *e*)) | *e*. *e* ∈ *set list* ∧ *nb e* ∧ *ereal* (*f e*) > *0*}
shows (*beste*, *bestc*) ∈ *edges-and-costs* ∧
(∀ (*ee*, *cc*) ∈ *edges-and-costs*. *bestc* ≤ *cc*)
using *assms*
unfolding *find-cheapest-backward-def*
by(*induction list arbitrary*: *edges-and-costs beste bestc*)
(*auto split*: *if-split prod.split*,
insert ereal-le-less less-le-not-le nless-le order-less-le-trans, *fastforce*+)

lemma *get-edge-and-costs-forward-not-MInfty*:
prod.snd(*get-edge-and-costs-forward nb f u v*) ≠*MInfty*
unfolding *get-edge-and-costs-forward-def*
using *not-both-realising-edges-none*[*of u v*]
imageI[*OF conjunct1*[*OF*
find-cheapest-forward-props[*OF prod.collapse refl*,
of f nb - create-edge u v PInfty]],
of prod.snd , *simplified image-def*, *simplified*]
imageI[*OF conjunct1*[*OF*
find-cheapest-backward-props[*OF prod.collapse refl*,
of f nb - create-edge v u PInfty]],
of prod.snd, *simplified image-def*, *simplified*]
by(*auto split*: *if-split prod.split option.split*)
(*metis MInfty-neq-PInfty*(*1*) *MInfty-neq-ereal*(*1*) *snd-conv*)+

lemma *get-edge-and-costs-backward-not-MInfty*:
prod.snd(*get-edge-and-costs-backward nb f u v*) ≠*MInfty*
unfolding *get-edge-and-costs-backward-def*
using *not-both-realising-edges-none*[*of v u*]
using *imageI*[*OF conjunct1*[*OF*
find-cheapest-forward-props[*OF prod.collapse refl*,
of f nb - create-edge v u PInfty]],
of prod.snd, *simplified image-def*, *simplified*]
using *imageI*[*OF conjunct1*[*OF*
find-cheapest-backward-props[*OF prod.collapse refl*,
of f nb - create-edge u v PInfty]],
of prod.snd, *simplified image-def*, *simplified*]
by(*auto split*: *if-split prod.split option.split*)
(*metis MInfty-neq-PInfty*(*1*) *MInfty-neq-ereal*(*1*) *snd-conv*)+

lemma *realising-edges-general-result-None-and-Some*:
  assumes (*case realising-edges-lookup* (*realising-edges-general list*) (*u*, *v*)
      of Some ds ⇒ *ds*
        | *None* ⇒ [])= *ds*
  shows *set ds* = {*e* | *e*. *e* ∈ *set list* ∧ *make-pair e* = (*u*, *v*)}
  using *assms*
  apply(*induction list arbitrary*: *ds*)
  apply(*simp add*: *realising-edges-lookup-def realising-edges-general-def*
        realising-edges-empty-def Map-realising.map-empty)
  subgoal for *a list ds*
    unfolding *realising-edges-general-def*
    apply(*subst* (*asm*) *foldr.foldr-Cons*, *subst* (*asm*) *o-apply*)
    unfolding *realising-edges-general-def*[*symmetric*]
    unfolding *Let-def realising-edges-lookup-def realising-edges-update-def*
    apply(*subst* (*asm*) (*9*) *option.split*, *subst* (*asm*) *Map-realising.map-update*)
    using *realising-edges-general-invar*
    apply(*force simp add*: *realising-edges-invar-def*)
    apply(*subst* (*asm*) *Map-realising.map-update*)
    using *realising-edges-general-invar*
    apply(*force simp add*: *realising-edges-invar-def*)
  by(*cases make-pair a* = (*u*, *v*))
   (*auto intro*: *option.exhaust*[*of realising-edges-lookup* (*realising-edges-general list*)
(*fst a*, *snd a*)]
      simp add: *make-pair-fst-snd*)
  done

lemma *realising-edges-general-result*:
  assumes *realising-edges-lookup* (*realising-edges-general list*) (*u*, *v*) = *Some ds*
  shows *set ds* = {*e* | *e*. *e* ∈ *set list* ∧ *make-pair e* = (*u*, *v*)}
  using *realising-edges-general-result-None-and-Some*[*of list u v ds*] *assms*
  by *simp*

lemma *realising-edges-result*:
    realising-edges-lookup realising-edges (*u*, *v*) = *Some ds* ⟹
    set ds = {*e* |*e*. *e* ∈ *set* $\mathcal{E}$*-list* ∧ *make-pair e* = (*u*, *v*)}
  by (*simp add*: *realising-edges-def realising-edges-general-result*)

lemma *get-edge-and-costs-forward-result-props*:
  assumes *get-edge-and-costs-forward nb f u v* = (*e*, *c*) *c* ≠*PInfty oedge e* = *d*
    shows *nb d*∧ *rcap f e* > *0* ∧ *fstv e* = *u* ∧ *sndv e* = *v* ∧
        d ∈ $\mathcal{E}$ ∧ *c* = $\mathfrak{c}$ *e*
proof–
  define *ingoing-edges* **where** *ingoing-edges* =
      (*case realising-edges-lookup realising-edges*
        (*u*, *v*) *of*
      None ⇒ [] | *Some list* ⇒ *list*)
  define *outgoing-edges* **where** *outgoing-edges* =
      (*case realising-edges-lookup realising-edges*

$(v,\ u)$ *of*
None $\Rightarrow$ [] | *Some list* $\Rightarrow$ *list*)
define *ef* **where** *ef* = *prod.fst* (*find-cheapest-forward f nb ingoing-edges* (*create-edge u v*) *PInfty*)
define *cf* **where** *cf* = *prod.snd* (*find-cheapest-forward f nb ingoing-edges* (*create-edge u v*) *PInfty*)
define *eb* **where** *eb* = *prod.fst* (*find-cheapest-backward f nb outgoing-edges* (*create-edge v u*) *PInfty*)
define *cb* **where** *cb* = *prod.snd* (*find-cheapest-backward f nb outgoing-edges* (*create-edge v u*) *PInfty*)
have *result-simp*:$(e,\ c)$ = (*if* $cf \leq cb$ *then* (*F ef*, *cf*) *else* (*B eb*, *cb*))
by(*auto split*: *option.split prod.split*
simp add: *get-edge-and-costs-forward-def sym*[*OF assms*(*1*)] *cf-def cb-def ingoing-edges-def outgoing-edges-def ef-def eb-def*)
show *?thesis*
proof(*cases* $cf \leq cb$)
case *True*
hence *result-is*:*F ef* = *e cf* = *c ef* = *d*
using *result-simp* *assms*(*3*) **by** *auto*
define *edges-and-costs* **where** *edges-and-costs* =
Set.insert (*create-edge u v*, *PInfty*)
$\{(e,\ ereal\ (\mathrm{c}\ e))\ |e.\ e \in set\ ingoing\text{-}edges \wedge nb\ e \wedge ereal\ (f\ e) < \mathrm{u}\ e\}$
have *ef-cf-prop*:(*ef*, *cf*) $\in$ *edges-and-costs*
using *find-cheapest-forward-props*[*of ef cf f nb ingoing-edges*
create-edge u v PInfty edges-and-costs]
by (*auto simp add*: *cf-def edges-and-costs-def ef-def*)
hence *ef-in-a-Set*:(*ef*, *cf*) $\in$
$\{(e,\ ereal\ (\mathrm{c}\ e))\ |e.\ e \in set\ ingoing\text{-}edges \wedge nb\ e \wedge ereal\ (f\ e) < \mathrm{u}\ e\}$
using *result-is*(*2*) *assms*(*2*)
by(*auto simp add*: *edges-and-costs-def*)
hence *ef-props*: *ef* $\in$ *set ingoing-edges nb ef ereal* (*f ef*) < u *ef* **by** *auto*
have *realising-not-none*: *realising-edges-lookup realising-edges* $(u,\ v) \neq$ *None*
using *ef-props*
by(*auto split*: *option.split simp add*: *ingoing-edges-def*) *metis*
then obtain *list* **where** *list-prop*: *realising-edges-lookup realising-edges* $(u,\ v)$ = *Some list*
by *auto*
have *set ingoing-edges* = $\{e\ |e.\ e \in set\ \mathcal{E}\text{-}list \wedge make\text{-}pair\ e = (u,\ v)\}$
using *realising-edges-result*[*OF list-prop*] *list-prop*
by(*auto simp add*: *ingoing-edges-def*)
hence *ef-inE*:*ef* $\in \mathcal{E}$ *make-pair ef* = $(u,\ v)$
using *ef-props*(*1*)
by(*simp add*: $\mathcal{E}$*-def* $\mathcal{E}$*-impl-basic*(*1*) $\mathcal{E}$*-list-def to-list*(*1*))+
show *?thesis*
using *ef-inE*(*1*) *ef-inE*(*2*)[*simplified make-pair-fst-snd*] *ef-in-a-Set*
by(*auto simp add*: *ef-props* *ereal-diff-gr0 result-is*[*symmetric*])
next
case *False*
hence *result-is*:*B eb* = *e cb* = *c eb* = *d*

using *result-simp assms*(*3*) **by** *auto*
define *edges-and-costs* **where** *edges-and-costs* =
Set.insert (*create-edge v u*, *PInfty*)
{(*e*, *ereal* (− c *e*)) |*e*. *e* ∈ *set outgoing-edges* ∧ *nb e* ∧ *ereal* (*f e*) > *0*}
have *ef-cf-prop*:(*eb*, *cb*) ∈ *edges-and-costs*
using *find-cheapest-backward-props*[*of eb cb f nb outgoing-edges*
create-edge v u PInfty edges-and-costs]
by (*auto simp add*: *cb-def edges-and-costs-def eb-def*)
hence *ef-in-a-Set*:(*eb*, *cb*) ∈
{(*e*, *ereal* (− c *e*)) |*e*. *e* ∈ *set outgoing-edges* ∧ *nb e* ∧ *ereal* (*f e*) > *0*}
using *result-is*(*2*) *assms*(*2*)
by(*auto simp add*: *edges-and-costs-def*)
hence *ef-props*: *eb* ∈ *set outgoing-edges nb eb ereal* (*f eb*) > *0* **by** *auto*
have *realising-not-none*: *realising-edges-lookup realising-edges* (*v*, *u*) ≠ *None*
using *ef-props*
by(*auto split*: *option.split simp add*: *outgoing-edges-def*) *metis*
then obtain *list* **where** *list-prop*: *realising-edges-lookup realising-edges* (*v*, *u*)
= *Some list*
by *auto*
have *set outgoing-edges* = {*e* |*e*. *e* ∈ *set* $\mathcal{E}$*-list* ∧ *make-pair e* = (*v*, *u*)}
using *realising-edges-result*[*OF list-prop*] *list-prop*
by(*auto simp add*: *outgoing-edges-def*)
hence *ef-inE*:*eb* ∈ $\mathcal{E}$ *make-pair eb* = (*v*, *u*)
using *ef-props*(*1*)
by(*simp add*: $\mathcal{E}$*-def* $\mathcal{E}$*-impl-basic*(*1*) $\mathcal{E}$*-list-def to-list*(*1*))+
show *?thesis*
using *ef-inE*(*1*) *ef-inE*(*2*)[*simplified make-pair-fst-snd*] *ef-in-a-Set*
by(*auto simp add*: *ef-props ereal-diff-gr0 result-is*[*symmetric*])
qed
qed

lemma *get-edge-and-costs-backward-result-props*:
assumes *get-edge-and-costs-backward nb f v u* = (*e*, *c*) *c* ≠*PInfty oedge e* = *d*
shows *nb d* ∧ *cost-flow-network.rcap f e* > *0* ∧ *fstv e* = *u* ∧ *sndv e* = *v* ∧ *d* ∈
$\mathcal{E}$ ∧ *c* = $\mathfrak{c}$ *e*
proof−
define *ingoing-edges* **where** *ingoing-edges* =
(*case realising-edges-lookup realising-edges*
(*u*, *v*) *of*
None ⇒ [] | *Some list* ⇒ *list*)
define *outgoing-edges* **where** *outgoing-edges* =
(*case realising-edges-lookup realising-edges*
(*v*, *u*) *of*
None ⇒ [] | *Some list* ⇒ *list*)
define *ef* **where** *ef* = *prod.fst* (*find-cheapest-forward f nb ingoing-edges*
(*create-edge u v*) *PInfty*)
define *cf* **where** *cf* = *prod.snd* (*find-cheapest-forward f nb ingoing-edges*
(*create-edge u v*) *PInfty*)
define *eb* **where** *eb* = *prod.fst* (*find-cheapest-backward f nb outgoing-edges*

```
          (create-edge v u) PInfty)
  define cb where cb = prod.snd (find-cheapest-backward f nb outgoing-edges
          (create-edge v u) PInfty)
  have result-simp:(e, c) = (if cf ≤ cb then (F ef, cf) else (B eb, cb))
    by(auto split: option.split prod.split
        simp add: get-edge-and-costs-backward-def sym[OF assms(1)] cf-def cb-def
ingoing-edges-def outgoing-edges-def ef-def eb-def )
  show ?thesis
  proof(cases cf ≤ cb)
    case True
    hence result-is:F ef = e cf = c ef = d
      using result-simp  assms(3) by auto
    define edges-and-costs where edges-and-costs =
       Set.insert (create-edge u v, PInfty)
          {(e, ereal (c e)) |e. e ∈ set ingoing-edges ∧ nb e ∧ ereal (f e) < u e}
    have ef-cf-prop:(ef, cf) ∈ edges-and-costs
      using find-cheapest-forward-props[of ef cf f nb ingoing-edges
                         create-edge u v PInfty edges-and-costs]
      by (auto simp add: cf-def edges-and-costs-def ef-def)
    hence ef-in-a-Set:(ef, cf) ∈
        {(e, ereal (c e)) |e. e ∈ set ingoing-edges ∧ nb e ∧ ereal (f e) < u e}
      using result-is(2) assms(2)
      by(auto simp add: edges-and-costs-def)
    hence ef-props: ef ∈ set ingoing-edges nb ef ereal (f ef) < u ef by auto
    have realising-not-none: realising-edges-lookup realising-edges (u, v) ≠ None
      using  ef-props
      by(auto split: option.split simp add: ingoing-edges-def) metis
     then obtain list where list-prop: realising-edges-lookup realising-edges (u, v)
= Some list
      by auto
    have set ingoing-edges = {e |e. e ∈ set ℰ-list ∧ make-pair e = (u, v)}
      using realising-edges-result[OF list-prop] list-prop
      by(auto simp add: ingoing-edges-def)
    hence ef-inE:ef ∈ ℰ make-pair ef = (u, v)
      using ef-props(1)
      by(simp add: ℰ-def ℰ-impl-basic(1) ℰ-list-def to-list(1))+
    show ?thesis
      using  ef-inE(1) ef-inE(2)[simplified make-pair-fst-snd] ef-in-a-Set
      by(auto simp add: ef-props  ereal-diff-gr0 result-is[symmetric])
  next
    case False
    hence result-is:B eb = e cb = c eb = d
      using result-simp  assms(3) by auto
    define edges-and-costs where edges-and-costs =
       Set.insert (create-edge v u, PInfty)
        {(e, ereal (− c e)) |e. e ∈ set outgoing-edges ∧ nb e ∧ ereal (f e) > 0}
    have ef-cf-prop:(eb, cb) ∈ edges-and-costs
      using find-cheapest-backward-props[of eb cb f nb outgoing-edges
                         create-edge v u PInfty edges-and-costs]
```

by (*auto simp add*: *cb-def edges-and-costs-def eb-def*)
hence *ef-in-a-Set*:(*eb*, *cb*) $\in$
$\{(e,\ ereal\ (-\ \mathrm{c}\ e))\ |e.\ e \in set\ outgoing\text{-}edges \wedge nb\ e \wedge ereal\ (f\ e) > 0\}$
using *result-is*(*2*) *assms*(*2*)
by(*auto simp add*: *edges-and-costs-def*)
hence *ef-props*: *eb* $\in$ *set outgoing-edges nb eb ereal* (*f eb*) $>$ *0* **by** *auto*
have *realising-not-none*: *realising-edges-lookup realising-edges* (*v*, *u*) $\neq$ *None*
using *ef-props*
by(*auto split*: *option.split simp add*: *outgoing-edges-def*) *metis*
then obtain *list* **where** *list-prop*: *realising-edges-lookup realising-edges* (*v*, *u*) = *Some list*
by *auto*
have *set outgoing-edges* = $\{e\ |e.\ e \in set\ \mathcal{E}\text{-}list \wedge make\text{-}pair\ e = (v,\ u)\}$
using *realising-edges-result*[*OF list-prop*] *list-prop*
by(*auto simp add*: *outgoing-edges-def*)
hence *ef-inE*:*eb* $\in \mathcal{E}$ *make-pair eb* = (*v*, *u*)
using *ef-props*(*1*)
by(*simp add*: $\mathcal{E}$*-def* $\mathcal{E}$*-impl-basic*(*1*) $\mathcal{E}$*-list-def to-list*(*1*))+
show *?thesis*
using *ef-inE*(*1*) *ef-inE*(*2*)[*simplified make-pair-fst-snd*] *ef-in-a-Set*
by(*auto simp add*: *ef-props ereal-diff-gr0 result-is*[*symmetric*])
qed
qed

lemmas *EEE-def* = *flow-network-spec*.$\mathfrak{E}$*-def*

lemma *es-E-frac*: *cost-flow-network.to-vertex-pair* ‘ *EEE* = *set es*
proof(*goal-cases*)
case *1*
have *help1*: $[\![$ (*a*, *b*) = *prod.swap* (*make-pair d*); *prod.swap* (*make-pair d*) $\notin$ *make-pair* ‘ $\mathcal{E}$; *d* $\in \mathcal{E}]\!]$
$\Longrightarrow$ (*b*, *a*) $\in$ *make-pair* ‘ *local*.$\mathcal{E}$ **for** *a b d*
using *cost-flow-network.to-vertex-pair.simps*
by (*metis imageI swap-simp swap-swap*)
have *help2*: $[\![$(*a*, *b*) = *make-pair x* ; *x* $\in$ *local*.$\mathcal{E}$ $]\!]\Longrightarrow$
make-pair x $\in$ *to-edge* ‘
($\{F\ d\ |d.\ d \in local.\mathcal{E}\} \cup \{B\ d\ |d.\ d \in local.\mathcal{E}\}$)
for *a b x*
using *cost-flow-network.to-vertex-pair.simps make-pairs-are*
by(*metis* (*mono-tags*, *lifting*) *UnI1 imageI mem-Collect-eq*)
have *help3*: $[\![$ (*b*, *a*) = *make-pair x* ; *x* $\in$ *local*.$\mathcal{E}]\!]$ $\Longrightarrow$
(*a*, *b*) $\in$ *to-edge* ‘
($\{F\ d\ |d.\ d \in local.\mathcal{E}\} \cup \{B\ d\ |d.\ d \in local.\mathcal{E}\}$)
for *a b x*
by (*smt* (*verit*, *del-insts*) *cost-flow-network*.$\mathfrak{E}$*-def cost-flow-network.o-edge-res make-pairs-are*
flow-network-spec.oedge.simps(*2*) *cost-flow-network.to-vertex-pair.simps*(*2*) *image-iff swap-simp*)
show *?case*

by(*auto simp add*: *cost-flow-network.to-vertex-pair.simps es-is-E EEE-def*
cost-flow-network.$\mathfrak{E}$*-def make-pairs-are Instantiation.make-pair-def*
intro: *help1 help2 help3*)
qed

lemma *to-edge-get-edge-and-costs-forward*:
to-edge (*prod.fst* ((*get-edge-and-costs-forward nb f u v*))) = (*u*, *v*)
unfolding *get-edge-and-costs-forward-def Let-def*
proof(*goal-cases*)
case *1*
have *help4*: ⟦*realising-edges-lookup local.realising-edges* (*u*, *v*) = *None* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *Some x2* ; ¬ *x2a* ≤ *x2b* ;
local.find-cheapest-backward f nb x2 (*create-edge v u*) ∞ = (*x1a*, *x2b*) ;
local.find-cheapest-forward f nb [] (*create-edge u v*) ∞ = (*x1*, *x2a*)⟧ ⟹
prod.swap (*make-pair x1a*) = (*u*, *v*)
for *x2 x1 x2a x1a x2b*
using *realising-edges-dom*[*of v u*] *realising-edges-result*[*of v u x2*]
find-cheapest-backward-props[*of x1a x2b f nb x2 create-edge v u PInfty*, *OF*
- *refl*]
by (*fastforce simp add*:)
have *help5*: ⟦*realising-edges-lookup local.realising-edges* (*u*, *v*) = *Some x2* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *None* ;*x2a* ≤ *x2b* ;
local.find-cheapest-backward f nb [] (*create-edge v u*) ∞ = (*x1a*, *x2b*) ;
local.find-cheapest-forward f nb x2 (*create-edge u v*) ∞ = (*x1*, *x2a*) ⟧⟹
make-pair x1 = (*u*, *v*)
for *x2 x1 x2a x1a x2b*
using *realising-edges-dom*[*of u v*] *realising-edges-result*[*of u v x2*]
find-cheapest-forward-props[*of x1 x2a f nb x2 create-edge u v PInfty*, *OF* -
refl]
by (*auto simp add*: *create-edge*)
have *help6*: ⟦*realising-edges-lookup local.realising-edges* (*u*, *v*) = *Some x2* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *Some x2a* ; *x2b* ≤ *x2c* ;
local.find-cheapest-backward f nb x2a (*create-edge v u*) ∞ = (*x1a*, *x2c*) ;
local.find-cheapest-forward f nb x2 (*create-edge u v*) ∞ = (*x1*, *x2b*) ⟧⟹
make-pair x1 = (*u*, *v*)
for *x2 x2a x1 x2b x1a x2c*
using *realising-edges-result*[*of u v x2*] *realising-edges-result*[*of v u x2a*]
find-cheapest-forward-props[*of x1 x2b f nb x2 create-edge u v PInfty*, *OF* -
refl]
find-cheapest-backward-props[*of x1a x2c f nb x2a create-edge v u PInfty*,
OF - *refl*]
by(*auto simp add*: *create-edge*)
have *help7*: ⟦ *realising-edges-lookup local.realising-edges* (*u*, *v*) = *Some x2* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *Some x2a* ;¬ *x2b* ≤ *x2c* ;
local.find-cheapest-backward f nb x2a (*create-edge v u*) ∞ = (*x1a*, *x2c*) ;
local.find-cheapest-forward f nb x2 (*create-edge u v*) ∞ = (*x1*, *x2b*) ⟧ ⟹
prod.swap (*make-pair x1a*) = (*u*, *v*)
for *x2 x2a x1 x2b x1a x2c*
using *realising-edges-result*[*of u v x2*] *realising-edges-result*[*of v u x2a*]

find-cheapest-forward-props[*of x1 x2b f nb x2 create-edge u v PInfty*, *OF* - *refl*]
find-cheapest-backward-props[*of x1a x2c f nb x2a create-edge v u PInfty*, *OF* - *refl*]
by(*auto simp add*: *create-edge*)
show *?case*
by(*auto split*: *if-split prod.split option.split*
simp add: *create-edge make-pairs-are find-cheapest-backward-def find-cheapest-forward-def*
Instantiation.make-pair-def
intro: *help4 help5 help6 help7*)
qed

lemma *to-edge-get-edge-and-costs-backward*:
to-edge (*prod.fst* ((*get-edge-and-costs-backward nb f v u*))) = (*u*, *v*)
unfolding *get-edge-and-costs-backward-def Let-def*
proof(*goal-cases*)
case *1*
have *help1*: ⟦ *realising-edges-lookup local.realising-edges* (*u*, *v*) = *None* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *Some x2* ;¬ *x2a* ≤ *x2b* ;
local.find-cheapest-backward f nb x2 (*create-edge v u*) ∞ = (*x1a*, *x2b*) ;
local.find-cheapest-forward f nb [] (*create-edge u v*) ∞ = (*x1*, *x2a*)⟧ ⟹
prod.swap (*make-pair x1a*) = (*u*, *v*)
for *x2 x1 x2a x1a x2b*
using *realising-edges-dom*[*of v u*] *realising-edges-result*[*of v u x2*]
find-cheapest-backward-props[*of x1a x2b f nb x2 create-edge v u PInfty*, *OF* - *refl*]
by (*fastforce simp add*:)
have *help2*: ⟦ *realising-edges-lookup local.realising-edges* (*u*, *v*) = *Some x2* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *None* ; *x2a* ≤ *x2b* ;
local.find-cheapest-backward f nb [] (*create-edge v u*) ∞ = (*x1a*, *x2b*) ;
local.find-cheapest-forward f nb x2 (*create-edge u v*) ∞ = (*x1*, *x2a*)⟧ ⟹
make-pair x1 = (*u*, *v*)
for *x2 x1 x2a x1a x2b*
using *realising-edges-dom*[*of u v*]
using *realising-edges-result*[*of u v x2*]
using *find-cheapest-forward-props*[*of x1 x2a f nb x2 create-edge u v PInfty*, *OF* - *refl*]
by (*auto simp add*: *create-edge*)
have *help3*: ⟦ *realising-edges-lookup local.realising-edges* (*u*, *v*) = *Some x2* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *Some x2a* ; *x2b* ≤ *x2c* ;
local.find-cheapest-backward f nb x2a (*create-edge v u*) ∞ = (*x1a*, *x2c*) ;
local.find-cheapest-forward f nb x2 (*create-edge u v*) ∞ = (*x1*, *x2b*) ⟧ ⟹
make-pair x1 = (*u*, *v*)
for *x2 x2a x1 x2b x1a x2c*
using *realising-edges-result*[*of u v x2*] *realising-edges-result*[*of v u x2a*]
using *find-cheapest-forward-props*[*of x1 x2b f nb x2 create-edge u v PInfty*, *OF* - *refl*]
using *find-cheapest-backward-props*[*of x1a x2c f nb x2a create-edge v u PInfty*, *OF* - *refl*]

by(*auto simp add*: *create-edge*)
have *help4*: ⟦ *realising-edges-lookup local.realising-edges* (*u*, *v*) = *Some x2* ;
realising-edges-lookup local.realising-edges (*v*, *u*) = *Some x2a* ; ¬ *x2b* ≤ *x2c* ;
local.find-cheapest-backward f nb x2a (*create-edge v u*) ∞ = (*x1a*, *x2c*) ;
local.find-cheapest-forward f nb x2 (*create-edge u v*) ∞ = (*x1*, *x2b*) ⟧ ⟹
prod.swap (*make-pair x1a*) = (*u*, *v*)
for *x2 x2a x1 x2b x1a x2c*
using *realising-edges-result*[*of u v x2*] *realising-edges-result*[*of v u x2a*]
using *find-cheapest-forward-props*[*of x1 x2b f nb x2 create-edge u v PInfty*, *OF* - *refl*]
using *find-cheapest-backward-props*[*of x1a x2c f nb x2a create-edge v u PInfty*, *OF* - *refl*]
by(*auto simp add*: *multigraph.create-edge*)
show *?case*
by(*auto split*: *if-split prod.split option.split*
simp add: *create-edge make-pairs-are find-cheapest-forward-def*
find-cheapest-backward-def Instantiation.make-pair-def
intro: *help1 help2 help3 help4*)
qed

lemma *costs-forward-less-PInfty-in-es*:
prod.snd (*get-edge-and-costs-forward nb f u v*) < *PInfty* ⟹ (*u*, *v*) ∈ *set es*
using *get-edge-and-costs-forward-result-props*[*OF prod.collapse*[*symmetric*] - *refl*, *of nb f u v*,
simplified cost-flow-network.o-edge-res]
es-E-frac to-edge-get-edge-and-costs-forward[*of nb f u v*]
by *force*

lemma *costs-backward-less-PInfty-in-es*:
prod.snd (*get-edge-and-costs-backward nb f u v*) < *PInfty* ⟹ (*v*, *u*) ∈ *set es*
using *get-edge-and-costs-backward-result-props*[*OF prod.collapse*[*symmetric*] - *refl*, *of nb f u v*,
simplified cost-flow-network.o-edge-res]
es-E-frac to-edge-get-edge-and-costs-backward[*of nb f u v*]
by *force*

lemma *bellman-ford*:
shows *bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v*. *prod.snd* (*get-edge-and-costs-forward nb f u v*)) *connection-update*
proof–
have *MInfty*:*MInfty* < *prod.snd* (*get-edge-and-costs-forward nb f u v*) **for** *u v*
using *get-edge-and-costs-forward-not-MInfty* **by** *auto*
show *?thesis*
using *Map-connection MInfty vs-and-es costs-forward-less-PInfty-in-es*
by (*auto simp add*: *bellman-ford-def bellman-ford-spec-def bellman-ford-axioms-def*)
qed

interpretation *bf-fw*: *bellman-ford*

where *connection-update=connection-update*
and *connection-empty=connection-empty*
and *connection-lookup=connection-lookup*
and *connection-delete=connection-delete*
and *connection-invar=connection-invar*
and *es= es*
and *vs=vs*
and *edge-costs*=(λ *u v. prod.snd* (*get-edge-and-costs-forward nb f u v*))
for *nb f*
using *bellman-ford* **by** *auto*

lemma *es-sym*: *prod.swap e* $\in$ *set es* $\Longrightarrow$ *e* $\in$ *set es*
unfolding *es-def to-list-def* $\mathcal{E}$*-def*
by (*cases e*) (*auto simp add*: *make-pair-fst-snd*)

lemma *bellman-ford-backward*:
shows *bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v. prod.snd* (*get-edge-and-costs-backward nb f u v*)) *connection-update*
proof−
have *MInfty*:*MInfty* < *prod.snd* (*get-edge-and-costs-backward nb f u v*) **for** *u v*
using *get-edge-and-costs-backward-not-MInfty* **by** *auto*
show *?thesis*
using *Map-connection MInfty vs-and-es costs-backward-less-PInfty-in-es*
by (*auto simp add*: *bellman-ford-def es-sym bellman-ford-spec-def bellman-ford-axioms-def intro*: *es-sym*)
qed

interpretation *bf-bw*: *bellman-ford*
where *connection-update=connection-update*
and *connection-empty=connection-empty*
and *connection-lookup=connection-lookup*
and *connection-delete=connection-delete*
and *connection-invar=connection-invar*
and *es= es*
and *vs=vs*
and *edge-costs*= (λ *u v. prod.snd* (*get-edge-and-costs-backward nb f u v*))
for *nb f*
using *bellman-ford-backward* **by** *auto*

lemma *get-source-aux*:
($\exists$ $x \in$ *set xs*. $b\ x > (1 - \varepsilon) * \gamma$) $\Longrightarrow$ (*get-source-aux b* γ *xs*) $\neq$ *None*
Some res = (*get-source-aux b* γ *xs*) $\Longrightarrow$ *b res* $> (1 - \varepsilon) * \gamma \land$ *res* $\in$ *set xs*
$\neg$ ($\exists$ $x \in$ *set xs*. $b\ x > (1 - \varepsilon) * \gamma$) $\Longrightarrow$ (*get-source-aux b* γ *xs*) = *None*
unfolding *get-source-aux-def*
by(*induction b* γ *xs rule*: *get-source-aux-aux.induct*) *force+*

lemma *get-target-aux*:

```
(∃ x ∈ set xs. b x < − (1 − ε) ∗ γ ) =⇒ (get-target-aux b γ xs)≠ None
Some res = (get-target-aux b γ xs) =⇒ b res < − (1 − ε) ∗ γ ∧ res ∈ set xs
¬ (∃ x ∈ set xs. b x < − (1 − ε) ∗ γ ) =⇒ (get-target-aux b γ xs) = None
 unfolding get-target-aux-def
 by(induction b γ xs rule: get-target-aux-aux.induct) force+

abbreviation underlying-invars (state)≡ algo.underlying-invars state
abbreviation invar-isOptflow (state)≡ algo.invar-isOptflow state
abbreviation ℱ state ≡ algo.ℱ (state)
abbreviation resreach ≡ cost-flow-network.resreach
abbreviation augpath ≡ cost-flow-network.augpath
abbreviation invar-gamma (state)== algo.invar-gamma state
abbreviation augcycle == cost-flow-network.augcycle
abbreviation prepath == cost-flow-network.prepath

lemmas ℱ-def = algo.ℱ-def
lemmas ℱ-redges-def = algo.ℱ-redges-def

lemmas prepath-def = cost-flow-network.prepath-def
lemmas augpath-def = cost-flow-network.augpath-def

lemma realising-edges-invar: realising-edges-invar realising-edges
 by (simp add: realising-edges-def realising-edges-general-invar)

lemma both-realising-edges-none-iff-not-in-es:
(u, v) ∈ set es ←→ ( realising-edges-lookup realising-edges (u, v) ≠ None ∨
                    realising-edges-lookup realising-edges (v, u) ≠ None)
 using realising-edges-dom make-pair-fst-snd
 by(auto simp add: es-def ℰ-list-def) blast

lemma get-edge-and-costs-forward-makes-cheaper:
 assumes oedge e = d d ∈ ℰ nb d cost-flow-network.rcap f e > 0
        (C , c) = get-edge-and-costs-forward nb f (fstv e) (sndv e)
  shows c ≤ 𝔠 e ∧ c ≠ MInfty
 unfolding snd-conv[of C c, symmetric, simplified assms(5)]
 unfolding get-edge-and-costs-forward-def
proof(cases (fstv e, sndv e) ∉ set es, goal-cases)
 case 1
 then show ?case
  using cost-flow-network.o-edge-res cost-flow-network.to-vertex-pair-fst-snd assms(1)
assms(2) es-E-frac
  by(auto split: prod.split option.split simp add: find-cheapest-backward-def find-cheapest-forward-def)
next
 case 2
 note ines = this[simplified]
 define ingoing-edges where ingoing-edges =
          (case realising-edges-lookup realising-edges
                (fstv e, sndv e) of
          None ⇒ [] | Some list ⇒ list)
```

define *outgoing-edges* **where** *outgoing-edges* =
(*case realising-edges-lookup realising-edges*
(*sndv e*, *fstv e*) *of*
None ⇒ [] | *Some list* ⇒ *list*)
define *ef* **where** *ef* = *prod.fst* (*find-cheapest-forward f nb ingoing-edges*
(*create-edge* (*fstv e*) (*sndv e*)) *PInfty*)
define *cf* **where** *cf* = *prod.snd* (*find-cheapest-forward f nb ingoing-edges*
(*create-edge* (*fstv e*) (*sndv e*)) *PInfty*)
define *eb* **where** *eb* = *prod.fst* (*find-cheapest-backward f nb outgoing-edges*
(*create-edge* (*sndv e*) (*fstv e*)) *PInfty*)
define *cb* **where** *cb* = *prod.snd* (*find-cheapest-backward f nb outgoing-edges*
(*create-edge* (*sndv e*) (*fstv e*)) *PInfty*)
have *goalI*: *prod.snd* (*if cf* ≤ *cb then* (*F ef*, *cf*) *else* (*B eb*, *cb*))
≤ *ereal* (𝔠 *e*) ∧
prod.snd (*if cf* ≤ *cb then* (*F ef*, *cf*) *else* (*B eb*, *cb*)) ≠ *MInfty* ⟹
?case
by(*auto split*: *prod.split simp add*: *cf-def cb-def ef-def eb-def*
ingoing-edges-def outgoing-edges-def)
show *?case*
proof(*cases e*, *all* ‹*rule goalI*›, *all* ‹*simp only*: *cost-flow-network*.𝔠.*simps*›, *goal-cases*)
case (*1 ee*)
define *edges-and-costs* **where** *edges-and-costs* =
Set.insert (*create-edge* (*fst ee*) (*snd ee*), *PInfty*)
{(*e*, *ereal* (c *e*)) |*e*. *e* ∈ *set ingoing-edges* ∧ *nb e* ∧ *ereal* (*f e*) < u *e*}
have *ef-cf-prop*:(*ef*, *cf*) ∈ *edges-and-costs* ⋀ *ee cc*. (*ee*, *cc*)∈*edges-and-costs* ⟹
cf ≤ *cc*
using *find-cheapest-forward-props*[*of ef cf f nb ingoing-edges*
create-edge (*fst ee*) (*snd ee*) *PInfty edges-and-costs*]
by (*auto simp add*: *1 cf-def edges-and-costs-def ef-def*)
obtain *list* **where** *listexists*:*realising-edges-lookup realising-edges*
(*fstv e*, *sndv e*) = *Some list*
using *realising-edges-dom*[*of fstv e sndv e*] *assms*(*1*,*2*) *1*
by (*auto simp add*: *es-def* 𝓔*-list-def make-pair-fst-snd* 𝓔*-def* 𝓔*-impl*(*1*)
to-list(*1*))
have *ee-in-ingoing*:*ee* ∈ *set ingoing-edges*
unfolding *ingoing-edges-def*
using *realising-edges-dom*[*of fstv e sndv e*, *simplified listexists*, *simplified*]
realising-edges-result[*OF listexists*]
1 cost-flow-network.to-vertex-pair.simps cost-flow-network.to-vertex-pair-fst-snd
𝓔*-def*
𝓔*-impl*(*1*) 𝓔*-list-def assms*(*1*) *assms*(*2*) *to-list*(*1*) *listexists*
by (*fastforce simp add*: *ingoing-edges-def make-pairs-are*)
have *cf* ≤ c *ee*
using *1 assms*(*1*−*4*) *ee-in-ingoing*
by (*auto intro*: *ef-cf-prop*(*2*)[*of ee*] *simp add*: *algo.infinite-u edges-and-costs-def*)
moreover have *cf* ≠*MInfty*
using *ef-cf-prop*(*1*) **by**(*auto simp add*: *edges-and-costs-def*)
ultimately show *?case*
using *find-cheapest-backward-props*[*OF prod.collapse refl*, *of f nb outgoing-edges*

create-edge (*sndv e*) (*fstv e*) *PInfty*]
by *auto* (*auto simp add*: *cb-def*)
next
case (*2 ee*)
define *edges-and-costs* **where** *edges-and-costs* =
Set.insert (*create-edge* (*fst ee*) (*snd ee*), *PInfty*)
{(*e*, *ereal* (− c *e*)) |*e*. *e* ∈ *set outgoing-edges* ∧ *nb e* ∧ *0* < *ereal* (*f e*)}
have *ef-cf-prop*:(*eb*, *cb*) ∈ *edges-and-costs* ⋀ *ee cc*. (*ee*, *cc*)∈*edges-and-costs* ⟹ *cb* ≤ *cc*
using *find-cheapest-backward-props*[*of eb cb f nb outgoing-edges*
create-edge (*fst ee*) (*snd ee*) *PInfty edges-and-costs*]
by(*auto simp add*: *2 cb-def edges-and-costs-def eb-def*)

obtain *list* **where** *listexists*:*realising-edges-lookup realising-edges*
(*sndv e*, *fstv e*) = *Some list*
using *realising-edges-dom*[*of sndv e fstv e*] *assms*(*1*,*2*) *2*
by (*auto simp add*: *es-def* $\mathcal{E}$*-list-def make-pair-fst-snd* $\mathcal{E}$*-def* $\mathcal{E}$*-impl*(*1*) *to-list*(*1*))
have *ee-in-ingoing*:*ee* ∈ *set outgoing-edges*
unfolding *ingoing-edges-def*
using *realising-edges-dom*[*of sndv e fstv e*, *simplified listexists*, *simplified*]
realising-edges-result[*OF listexists*]
2 cost-flow-network.to-vertex-pair.simps cost-flow-network.to-vertex-pair-fst-snd $\mathcal{E}$*-def*
$\mathcal{E}$*-impl*(*1*) $\mathcal{E}$*-list-def assms*(*1*) *assms*(*2*) *to-list*(*1*) *listexists*
by (*fastforce simp add*: *outgoing-edges-def make-pairs-are*)
have *cb* ≤ − c *ee*
using *2 assms*(*1*−*4*) *ee-in-ingoing*
by (*auto intro*: *ef-cf-prop*(*2*)[*of ee*] *simp add*: *algo.infinite-u edges-and-costs-def*)
moreover have *cb* ≠*MInfty*
using *ef-cf-prop*(*1*) **by**(*auto simp add*: *edges-and-costs-def*)
ultimately show *?case*
using *find-cheapest-forward-props*[*OF prod.collapse refl*, *of f nb ingoing-edges*
create-edge (*fstv e*) (*sndv e*) *PInfty*]
by *auto* (*auto simp add*: *cf-def*)
qed
qed

lemma *get-edge-and-costs-backward-makes-cheaper*:
assumes *oedge e* = *d d* ∈ $\mathcal{E}$ *nb d cost-flow-network.rcap f e* > *0*
(*C* , *c*) = *get-edge-and-costs-backward nb f* (*sndv e*) (*fstv e*)
shows *c* ≤ 𝔠 *e* ∧ *c* ≠ *MInfty*
unfolding *snd-conv*[*of C c*, *symmetric*, *simplified assms*(*5*)]
unfolding *get-edge-and-costs-backward-def*
proof(*cases* (*fstv e*, *sndv e*) ∉ *set es*, *goal-cases*)
case *1*
then show *?case*
using *cost-flow-network.o-edge-res cost-flow-network.vs-to-vertex-pair-pres*(*1*)
cost-flow-network.vs-to-vertex-pair-pres(*2*) *assms*(*1*) *assms*(*2*) *es-E-frac* **by**

auto
next
case *2*
note *ines* = *this*[*simplified*]
define *ingoing-edges* **where** *ingoing-edges* =
(*case realising-edges-lookup realising-edges*
(*fstv e*, *sndv e*) *of*
None ⇒ [] | *Some list* ⇒ *list*)
define *outgoing-edges* **where** *outgoing-edges* =
(*case realising-edges-lookup realising-edges*
(*sndv e*, *fstv e*) *of*
None ⇒ [] | *Some list* ⇒ *list*)
define *ef* **where** *ef* = *prod.fst* (*find-cheapest-forward f nb ingoing-edges*
(*create-edge* (*fstv e*) (*sndv e*)) *PInfty*)
define *cf* **where** *cf* = *prod.snd* (*find-cheapest-forward f nb ingoing-edges*
(*create-edge* (*fstv e*) (*sndv e*)) *PInfty*)
define *eb* **where** *eb* = *prod.fst* (*find-cheapest-backward f nb outgoing-edges*
(*create-edge* (*sndv e*) (*fstv e*)) *PInfty*)
define *cb* **where** *cb* = *prod.snd* (*find-cheapest-backward f nb outgoing-edges*
(*create-edge* (*sndv e*) (*fstv e*)) *PInfty*)
have *goalI*: *prod.snd* (*if cf* ≤ *cb then* (*F ef*, *cf*) *else* (*B eb*, *cb*))
≤ *ereal* (𝔠 *e*) ∧ *prod.snd* (*if cf* ≤ *cb then* (*F ef*, *cf*) *else* (*B eb*, *cb*))
≠ *MInfty* ⟹ *?case*
by(*auto split*: *prod.split simp add*: *cf-def cb-def ef-def eb-def*
ingoing-edges-def outgoing-edges-def)
show *?case*
proof(*cases e*, *all* ‹*rule goalI*›, *all* ‹*simp only*: *cost-flow-network*.𝔠.*simps*›, *goal-cases*)
case (*1 ee*)
define *edges-and-costs* **where** *edges-and-costs* =
Set.insert (*create-edge* (*fst ee*) (*snd ee*), *PInfty*)
{(*e*, *ereal* (c *e*)) |*e*. *e* ∈ *set ingoing-edges* ∧ *nb e* ∧ *ereal* (*f e*) < u *e*}
have *ef-cf-prop*:(*ef*, *cf*) ∈ *edges-and-costs* ⋀ *ee cc*. (*ee*, *cc*)∈*edges-and-costs* ⟹
cf ≤ *cc*
using *find-cheapest-forward-props*[*of ef cf f nb ingoing-edges*
create-edge (*fst ee*) (*snd ee*) *PInfty edges-and-costs*]
by (*auto simp add*: *1 cf-def edges-and-costs-def ef-def*)
obtain *list* **where** *listexists*:*realising-edges-lookup realising-edges*
(*fstv e*, *sndv e*) = *Some list*
using *realising-edges-dom*[*of fstv e sndv e*] *assms*(*1*,*2*) *1*
by (*auto simp add*: *es-def* ℰ*-list-def make-pair-fst-snd* ℰ*-def* ℰ*-impl*(*1*)
to-list(*1*))
have *ee-in-ingoing*:*ee* ∈ *set ingoing-edges*
unfolding *ingoing-edges-def*
using *realising-edges-dom*[*of fstv e sndv e*, *simplified listexists*, *simplified*]
realising-edges-result[*OF listexists*]
1 cost-flow-network.to-vertex-pair.simps cost-flow-network.to-vertex-pair-fst-snd
ℰ*-def*
ℰ*-impl*(*1*) ℰ*-list-def assms*(*1*) *assms*(*2*) *to-list*(*1*) *listexists*
by (*fastforce simp add*: *ingoing-edges-def make-pairs-are*)

have *cf* ≤ c *ee*
using *1 assms*(*1*−*4*) *ee-in-ingoing*
by (*auto intro*: *ef-cf-prop*(*2*)[*of ee*] *simp add*: *algo.infinite-u edges-and-costs-def*)
moreover have *cf* ≠*MInfty*
using *ef-cf-prop*(*1*) **by**(*auto simp add*: *edges-and-costs-def*)
ultimately show *?case*
using *find-cheapest-backward-props*[*OF prod.collapse refl*, *of f nb outgoing-edges create-edge* (*sndv e*) (*fstv e*) *PInfty*]
by *auto* (*auto simp add*: *cb-def*)
next
case (*2 ee*)
define *edges-and-costs* **where** *edges-and-costs* =
Set.insert (*create-edge* (*fst ee*) (*snd ee*), *PInfty*)
{(*e*, *ereal* (− c *e*)) |*e*. *e* ∈ *set outgoing-edges* ∧ *nb e* ∧ *0* < *ereal* (*f e*)}
have *ef-cf-prop*:(*eb*, *cb*) ∈ *edges-and-costs* ⋀ *ee cc*. (*ee*, *cc*)∈*edges-and-costs* ⟹ *cb* ≤ *cc*
using *find-cheapest-backward-props*[*of eb cb f nb outgoing-edges create-edge* (*fst ee*) (*snd ee*) *PInfty edges-and-costs*]
by(*auto simp add*: *2 cb-def edges-and-costs-def eb-def*)

obtain *list* **where** *listexists*:*realising-edges-lookup realising-edges* (*sndv e*, *fstv e*) = *Some list*
using *realising-edges-dom*[*of sndv e fstv e*] *assms*(*1*,*2*) *2*
by (*auto simp add*: *es-def* $\mathcal{E}$*-list-def make-pair-fst-snd* $\mathcal{E}$*-def* $\mathcal{E}$*-impl*(*1*) *to-list*(*1*))
have *ee-in-ingoing*:*ee* ∈ *set outgoing-edges*
unfolding *ingoing-edges-def*
using *realising-edges-dom*[*of sndv e fstv e*, *simplified listexists*, *simplified*] *realising-edges-result*[*OF listexists*]
2 cost-flow-network.to-vertex-pair.simps cost-flow-network.to-vertex-pair-fst-snd $\mathcal{E}$*-def*
$\mathcal{E}$*-impl*(*1*) $\mathcal{E}$*-list-def assms*(*1*) *assms*(*2*) *to-list*(*1*) *listexists*
by (*fastforce simp add*: *outgoing-edges-def make-pairs-are*)
have *cb* ≤ − c *ee*
using *2 assms*(*1*−*4*) *ee-in-ingoing*
by (*auto intro*: *ef-cf-prop*(*2*)[*of ee*] *simp add*: *algo.infinite-u edges-and-costs-def*)
moreover have *cb* ≠*MInfty*
using *ef-cf-prop*(*1*) **by**(*auto simp add*: *edges-and-costs-def*)
ultimately show *?case*
using *find-cheapest-forward-props*[*OF prod.collapse refl*, *of f nb ingoing-edges create-edge* (*fstv e*) (*sndv e*) *PInfty*]
by *auto* (*auto simp add*: *cf-def*)
qed
qed

lemma *less-PInfty-not-blocked*:
prod.snd (*get-edge-and-costs-forward nb f* (*fst e*) (*snd e*)) ≠ *PInfty*
⟹ *nb* (*oedge* (*prod.fst* (*get-edge-and-costs-forward nb f* (*fst e*) (*snd e*))))
using *get-edge-and-costs-forward-result-props prod.exhaust-sel* **by** *blast*

lemma *less-PInfty-not-blocked-backward*:
$prod.snd\ (get\text{-}edge\text{-}and\text{-}costs\text{-}backward\ nb\ f\ (fst\ e)\ (snd\ e)) \neq PInfty$
$\Longrightarrow nb\ (oedge\ (prod.fst\ (get\text{-}edge\text{-}and\text{-}costs\text{-}backward\ nb\ f\ (fst\ e)\ (snd\ e))))$
using *get-edge-and-costs-backward-result-props prod.exhaust-sel* **by** *blast*

abbreviation *weight nb f* $\equiv$ *bellman-ford.weight* (λ *u v. prod.snd* (*get-edge-and-costs-forward nb f u v*))

abbreviation *weight-backward nb f* $\equiv$ *bellman-ford.weight* (λ *u v. prod.snd* (*get-edge-and-costs-backward nb f u v*))

lemma *get-target-for-source-aux-aux*:
$(\exists\ x \in set\ xs.\ b\ x < -\varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ x)) < PInfty)$
$\longleftrightarrow(\ (get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs) \neq None)$
$(\ (get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs) \neq None)$
$\Longrightarrow (let\ x = the\ (get\text{-}target\text{-}for\text{-}source\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs)$
$in\ x \in set\ xs \wedge b\ x < -\varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ x)) < PInfty)$
by(*all* ‹*induction connections b* γ *xs rule*: *get-target-for-source-aux-aux.induct*›) *auto*

lemma *get-target-for-source-aux*:
$[\![(\exists\ x \in set\ xs.\ b\ x < -\varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ x)) < PInfty);$
$res = (get\text{-}target\text{-}for\text{-}source\text{-}aux\ connections\ b\ \gamma\ xs)\ ]\!]$
$\Longrightarrow b\ res < -\varepsilon * \gamma \wedge res \in set\ xs \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ res)) < PInfty$
by (*subst* (*asm*) *get-target-for-source-aux-def*,
induction connections b γ *xs rule*: *get-target-for-source-aux-aux.induct*) *force+*

lemma *get-source-for-target-aux-aux*:
$(\exists\ x \in set\ xs.\ b\ x > \varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ x)) < PInfty)$
$\longleftrightarrow(\ (get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs) \neq None)$
$(\ (get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs) \neq None)$
$\Longrightarrow (let\ x = the\ (get\text{-}source\text{-}for\text{-}target\text{-}aux\text{-}aux\ connections\ b\ \gamma\ xs)$
$in\ x \in set\ xs \wedge b\ x > \varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ x)) < PInfty)$
by (*all* ‹*induction connections b* γ *xs rule*: *get-source-for-target-aux-aux.induct*›) *auto*

lemma *get-source-for-target-aux*:
$[\![(\exists\ x \in set\ xs.\ b\ x > \varepsilon * \gamma \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ x)) < PInfty);$
$res = (get\text{-}source\text{-}for\text{-}target\text{-}aux\ connections\ b\ \gamma\ xs)]\!]$
$\Longrightarrow b\ res > \varepsilon * \gamma \wedge res \in set\ xs \wedge prod.snd\ (the\ (connection\text{-}lookup\ connections\ res)) < PInfty$

by (*subst* (*asm*) *get-source-for-target-aux-def*,
induction connections b γ *xs rule*: *get-source-for-target-aux-aux.induct*) *force+*

interpretation *send-flow-spec*: *send-flow-spec*
where $\mathcal{E} = \mathcal{E}$
and c = c
and u = u
and *edge-map-update* = *edge-map-update*
and *vset-empty* = *vset-empty*
and *vset-delete*= *vset-delete*
and *vset-insert* = *vset-insert*
and *vset-inv* = *vset-inv*
and *isin* = *isin*
and *get-from-set*=*get-from-set*
and *filter*=*filter*
and *are-all*=*are-all*
and *set-invar*=*set-invar*
and *to-set*=*to-set*
and *lookup*=*lookup*
and *t-set*=*t-set*
and *sel*=*sel*
and *adjmap-inv*=*adj-inv*
and $\varepsilon = \varepsilon$
and $\mathcal{E}$*-impl*=$\mathcal{E}$*-impl*
and *empty-forest*=*map-empty*
and b = b
and $N = N$
and *snd* = *snd*
and *fst* = *fst*
and *create-edge*=*create-edge*

and *flow-empty* = *flow-empty*
and *flow-lookup* = *flow-lookup*
and *flow-update* = *flow-update*
and *flow-delete*=*flow-delete*
and *flow-invar* = *flow-invar*

and *bal-empty* = *bal-empty*
and *bal-lookup* = *bal-lookup*
and *bal-update* = *bal-update*
and *bal-delete*=*bal-delete*
and *bal-invar* = *bal-invar*

and *conv-empty* = *conv-empty*
and *conv-lookup* = *conv-lookup*
and *conv-update* = *conv-update*
and *conv-delete*=*conv-delete*
and *conv-invar* = *conv-invar*

and *rep-comp-empty* = *rep-comp-empty*
and *rep-comp-lookup* = *rep-comp-lookup*
and *rep-comp-update* = *rep-comp-update*
and *rep-comp-delete*=*rep-comp-delete*
and *rep-comp-invar* = *rep-comp-invar*

and *not-blocked-empty* = *not-blocked-empty*
and *not-blocked-lookup* = *not-blocked-lookup*
and *not-blocked-update* = *not-blocked-update*
and *not-blocked-delete*=*not-blocked-delete*
and *not-blocked-invar* = *not-blocked-invar*

and *get-source-target-path-a* = *get-source-target-path-a*
and *get-source-target-path-b*=*get-source-target-path-b*
and *get-source* = *get-source*
and *get-target*=*get-target*
and *test-all-vertices-zero-balance*=*test-all-vertices-zero-balance*
by(*auto intro*!: *send-flow-spec.intro simp add*: *algo.algo-spec-axioms*)

lemmas *send-flow* = *send-flow-spec.send-flow-spec-axioms*

abbreviation *send-flow-call1-cond state* ≡ *send-flow-spec.send-flow-call1-cond state*
abbreviation *send-flow-fail1-cond state* ≡ *send-flow-spec.send-flow-fail1-cond state*
abbreviation *send-flow-call2-cond state* ≡ *send-flow-spec.send-flow-call2-cond state*
abbreviation *send-flow-fail2-cond state* ≡ *send-flow-spec.send-flow-fail2-cond state*
abbreviation *get-target-cond state* ≡ *send-flow-spec.get-target-cond state*
abbreviation *get-source-cond state* ≡ *send-flow-spec.get-source-cond state*
abbreviation *vertex-selection-cond* ≡ *send-flow-spec.vertex-selection-cond*
abbreviation *abstract-bal-map* ≡ *algo.abstract-bal-map*
abbreviation *abstract-flow-map* ≡ *algo.abstract-flow-map*
abbreviation *abstract-conv-map* ≡ *algo.abstract-conv-map*
abbreviation *abstract-not-blocked-map* ≡ *algo.abstract-not-blocked-map*
abbreviation *a-balance state* ≡ *algo.a-balance state*
abbreviation *a-current-flow state* ≡ *algo.a-current-flow state*
abbreviation *a-not-blocked state* ≡ *algo.a-not-blocked state*
abbreviation $\mathcal{V}$ ≡ *multigraph*.$\mathcal{V}$

lemmas *send-flow-fail1-condE* = *send-flow-spec.send-flow-fail1-condE*
lemmas *send-flow-call1-condE* = *send-flow-spec.send-flow-call1-condE*
lemmas *send-flow-fail1-cond-def* = *send-flow-spec.send-flow-fail1-cond-def*
lemmas *send-flow-call1-cond-def*= *send-flow-spec.send-flow-call1-cond-def*

lemmas *send-flow-fail2-condE* = *send-flow-spec.send-flow-fail2-condE*
lemmas *send-flow-call2-condE* = *send-flow-spec.send-flow-call2-condE*
lemmas *send-flow-fail2-cond-def* = *send-flow-spec.send-flow-fail2-cond-def*
lemmas *send-flow-call2-cond-def*= *send-flow-spec.send-flow-call2-cond-def*
lemmas *get-source-condE* = *send-flow-spec.get-source-condE*
lemmas *get-target-condE* = *send-flow-spec.get-target-condE*
lemmas *vertex-selection-condE* = *send-flow-spec.vertex-selection-condE*

lemmas *invar-gamma-def* = *algo.invar-gamma-def*
lemmas *invar-isOptflow-def* = *algo.invar-isOptflow-def*
lemmas *is-Opt-def* = *cost-flow-network.is-Opt-def*
lemmas *from-underlying-invars'* = *algo.from-underlying-invars'*

abbreviation *to-graph* == *Adj-Map-Specs2.to-graph*
abbreviation *digraph-abs* == *Adj-Map-Specs2.digraph-abs*

lemma *get-source-axioms-red*:
$[\![b = \textit{balance state}; \gamma = \textit{current-}\gamma\ \textit{state}; \textit{Some } s = \textit{get-source state}]\!]$
$\Longrightarrow s \in \mathcal{V} \wedge \textit{abstract-bal-map } b\ s > (1 - \varepsilon) * \gamma$
$[\![b = \textit{balance state}; \gamma = \textit{current-}\gamma\ \textit{state}; \textit{Some } s = \textit{get-source state}]\!]$
$\Longrightarrow \neg\ (\exists\ s \in \mathcal{V}.\ \textit{abstract-bal-map } b\ s > (1 - \varepsilon) * \gamma) \longleftrightarrow ((\textit{get-source state}) = \textit{None})$
using *get-source-aux*(*2*)[*of s a-balance state current-*γ *state vs*] *vs-is-V*
get-source-aux(*1*,*3*)[*of vs current-*γ *state a-balance state*]
by(*fastforce elim*: *get-source-condE elim*: *vertex-selection-condE*
simp add: *get-source-def get-source-aux-def make-pairs-are*)+

lemma *get-source-axioms*:
$\textit{get-source-cond } s\ \textit{state } b\ \gamma \Longrightarrow s \in \mathcal{V} \wedge \textit{abstract-bal-map } b\ s > (1 - \varepsilon) * \gamma$
vertex-selection-cond state b γ
$\Longrightarrow \neg\ (\exists\ s \in \mathcal{V}.\ \textit{abstract-bal-map } b\ s > (1 - \varepsilon) * \gamma) \longleftrightarrow ((\textit{get-source state}) = \textit{None})$
using *get-source-aux*(*2*)[*of s a-balance state current-*γ *state vs*] *vs-is-V*
get-source-aux(*1*,*3*)[*of vs current-*γ *state a-balance state*]
by(*fastforce elim*: *get-source-condE elim*: *vertex-selection-condE*
simp add: *get-source-def get-source-aux-def make-pairs-are*)+

lemma *get-target-axioms*:
$\textit{get-target-cond } t\ \textit{state } b\ \gamma \Longrightarrow t \in \mathcal{V} \wedge \textit{abstract-bal-map } b\ t < -\ (1 - \varepsilon) * \gamma$
vertex-selection-cond state b γ
$\Longrightarrow \neg\ (\exists\ t \in \mathcal{V}.\ \textit{abstract-bal-map } b\ t < -(1 - \varepsilon) * \gamma) \longleftrightarrow ((\textit{get-target state}) = \textit{None})$
using *get-target-aux*(*2*)[*of t a-balance state current-*γ *state vs*] *vs-is-V*
get-target-aux(*1*,*3*)[*of vs a-balance state current-*γ *state*]
by(*fastforce elim*: *get-target-condE elim*: *vertex-selection-condE*
simp add: *get-target-def get-target-aux-def make-pairs-are*)+

lemma *get-target-axioms-red*:
$[\![b = \textit{balance state}; \gamma = \textit{current-}\gamma\ \textit{state}; \textit{Some } t = \textit{get-target state}]\!]$
$\Longrightarrow t \in \mathcal{V} \wedge \textit{abstract-bal-map } b\ t < -\ (1 - \varepsilon) * \gamma$
$[\![b = \textit{balance state}; \gamma = \textit{current-}\gamma\ \textit{state}]\!]$
$\Longrightarrow \neg\ (\exists\ t \in \mathcal{V}.\ \textit{abstract-bal-map } b\ t < -(1 - \varepsilon) * \gamma) \longleftrightarrow ((\textit{get-target state}) = \textit{None})$
using *get-target-aux*(*2*)[*of t a-balance state current-*γ *state vs*] *vs-is-V*
get-target-aux(*1*,*3*)[*of vs a-balance state current-*γ *state*]
by(*fastforce elim*: *get-target-condE elim*: *vertex-selection-condE*
simp add: *get-target-def get-target-aux-def make-pairs-are*)+

lemma *path-flow-network-path-bf*:
assumes *e-weight*: $\bigwedge$ *e. e* $\in$ *set pp* $\Longrightarrow$ *prod.snd* (*get-edge-and-costs-forward nb f* (*fstv e*) (*sndv e*)) < *PInfty*
and *is-a-walk*: *awalk UNIV s* (*map to-edge pp*) *tt*
and *s-is-fstv*: *s* = *fstv* (*hd pp*)
and *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar*
connection-delete es vs (λ *u v. prod.snd*
(*get-edge-and-costs-forward nb f u v*)) *connection-update*

shows *weight nb f* (*awalk-verts s* (*map cost-flow-network.to-vertex-pair pp*)) < *PInfty*
using *assms*(*1*,*2*)[*simplified assms*(*3*)]
proof(*subst assms*(*3*), *induction pp rule*: *list-induct3*)
case *1*
then show *?case*
using *bellman-ford.weight.simps*[*OF bellman-ford*] **by** *auto*
next
case (*2 x*)
then show *?case*
using *bellman-ford.weight.simps*[*OF bellman-ford*] **apply** *auto*[*1*]
apply(*induction x rule*: *cost-flow-network.to-vertex-pair.induct*)
apply(*simp add*: *bellman-ford.weight.simps*[*OF bellman-ford*] *make-pair-fst-snd*

make-pairs-are Instantiation.make-pair-def)+
done
next
case (*3 e d es*)
have *same-ends*:*sndv e* = *fstv d*
using *3*(*3*)
by(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
(*auto intro*: *cost-flow-network.to-vertex-pair.induct*[*OF* , *of - d*]
simp add: *bellman-ford.weight.simps*[*OF bellman-ford*]
Awalk.awalk-simps make-pair-fst-snd Instantiation.make-pair-def
cost-flow-network.vs-to-vertex-pair-pres(*1*) *make-pairs-are*)
have *weight nb f*
(*awalk-verts* (*fstv* (*hd* ((*e* # *d* # *es*)))) (*map cost-flow-network.to-vertex-pair* (*e* # *d* # *es*))) =
prod.snd (*get-edge-and-costs-forward nb f* (*fstv e*) (*sndv e*))
+ *weight nb f* (*awalk-verts* (*fstv* (*hd* ((*d* # *es*)))) (*map cost-flow-network.to-vertex-pair* (*d* # *es*)))
using *same-ends*
by(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
(*auto intro*: *cost-flow-network.to-vertex-pair.induct*[*OF* , *of - d*]
simp add: *bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.to-vertex-pair-fst-snd multigraph.make-pair)
moreover have *prod.snd* (*get-edge-and-costs-forward nb f* (*fstv e*) (*sndv e*)) < *PInfty*

using *3.prems*(*1*) **by** *force*
moreover have *weight nb f* (*awalk-verts* (*fstv* (*hd* ((*d* # *es*)))) (*map* *cost-flow-network.to-vertex-pair* (*d* # *es*))) < *PInfty*
using *3*(*2*,*3*)
by(*intro* *3*(*1*), *auto intro*: *cost-flow-network.to-vertex-pair.induct*[*OF* , *of* - *e*]

simp add: *bellman-ford.weight.simps*[*OF bellman-ford*] *Awalk.awalk-simps*(*2*)[*of* *UNIV*]
cost-flow-network.vs-to-vertex-pair-pres(*1*))
ultimately show *?case* **by** *simp*
qed

lemma *path-bf-flow-network-path*:
assumes *True*
and *len*: *length pp* ≥ *2*
and *weight nb f pp* < *PInfty ppp* = *edges-of-vwalk pp*
shows *awalk UNIV* (*hd pp*) *ppp* (*last pp*) ∧
weight nb f pp = *foldr* (λ *e acc*. 𝔠 *e* + *acc*)
(*map* (λ *e*. (*prod.fst* (*get-edge-and-costs-forward nb f* (*prod.fst e*) (*prod.snd e*)))) *ppp*) *0*
∧ (∀ *e* ∈ *set* (*map* (λ *e*. (*prod.fst* (*get-edge-and-costs-forward nb f* (*prod.fst e*) (*prod.snd e*)))) *ppp*).
nb (*oedge e*) ∧ *cost-flow-network.rcap* *f e* > *0*)
proof−
have *bellman-ford*:*bellman-ford* *connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v*. *prod.snd* (*get-edge-and-costs-forward nb f u v*)) *connection-update*
by (*simp add*: *bellman-ford*)
show *?thesis*
using *assms*(*3*−)
proof(*induction pp arbitrary*: *ppp rule*: *list-induct3-len-geq-2*)
case *1*
then show *?case*
using *len* **by** *simp*
next
case (*2 x y*)
have *awalk UNIV* (*hd* [*x*, *y*]) *ppp* (*last* [*x*, *y*])
using *2* **unfolding** *get-edge-and-costs-forward-def Let-def*
by (*auto simp add*: *arc-implies-awalk bellman-ford.weight.simps*[*OF bellman-ford*]

split: *if-split prod.split*)
moreover have *weight nb f* [*x*, *y*] =
ereal
(*foldr* (λ*e*. (+) (𝔠 *e*)) (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward nb f* (*prod.fst e*) (*prod.snd e*))) *ppp*) *0*)
using *2* *bellman-ford.weight.simps*[*OF bellman-ford*]
by(*auto simp add*: *arc-implies-awalk get-edge-and-costs-forward-result-props*)
moreover have (∀ *e*∈*set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward nb f* (*prod.fst e*) (*prod.snd e*))) *ppp*).

nb (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap f e*)
using *2 bellman-ford.weight.simps*[*OF bellman-ford*] *flow-network-spec.oedge.simps*
cost-flow-network.rcap.simps get-edge-and-costs-forward-result-props[*OF*
sym[*OF prod.collapse*], *of nb f x y*]
by(*auto simp add*: u-*def*)
ultimately show *?case* **by** *simp*
next
case (*3 x y xs*)
thm *3*(*1*)[*OF - refl*]
have *awalk UNIV* (*hd* (*x* # *y* # *xs*)) *ppp* (*last* (*x* # *y* # *xs*))
using *conjunct1*[*OF 3.IH*[*OF - refl*]] *3.prems*(*1*)
bellman-ford.weight.simps(*3*)[*OF bellman-ford*] *edges-of-vwalk.simps*(*3*)
by (*simp add*: *3.prems*(*2*) *Awalk.awalk-simps*(*2*))
moreover have *weight nb f* (*x* # *y* # *xs*) = *prod.snd* (*get-edge-and-costs-forward*
nb f x y) +
weight nb f (*y* # *xs*)
using *bellman-ford bellman-ford.weight.simps*(*3*) **by** *fastforce*
moreover have *weight nb f* (*y* # *xs*) =
ereal
(*foldr* (λ*e*. (+) (𝔠 *e*))
(*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward nb f* (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk* (*y* # *xs*))) *0*)
using *3.IH 3.prems*(*1*) *calculation*(*2*) **by** *fastforce*
moreover have *prod.snd* (*get-edge-and-costs-forward nb f x y*) =
𝔠 (*prod.fst* (*get-edge-and-costs-forward nb f x y*))
using *3.prems*(*1*) *bellman-ford.weight.simps*[*OF bellman-ford*]
by (*simp add*: *get-edge-and-costs-forward-result-props*)
moreover have (∀ *e*∈*set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward nb f*
(*prod.fst e*) (*prod.snd e*))) (*edges-of-vwalk* (*y* # *xs*))).
nb (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap f e*)
by (*simp add*: *3.IH calculation*(*3*))
moreover have *nb* (*flow-network-spec.oedge* (*prod.fst* (*get-edge-and-costs-forward*
nb f x y)))
using *3.prems*(*1*) *bellman-ford.weight.simps*[*OF bellman-ford*]
get-edge-and-costs-forward-result-props[*OF prod.collapse*[*symmetric*], *of*
nb f x y]
by *auto*
moreover have *0* < *cost-flow-network.rcap f* (*prod.fst* (*get-edge-and-costs-forward*
nb f x y))
using *3.prems*(*1*) *bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.rcap.simps
get-edge-and-costs-forward-result-props[*OF prod.collapse*[*symmetric*], *of nb*
f x y]
by (*auto simp add*: u-*def*)
ultimately show *?case*
by (*auto simp add*: *3*(*3*))
qed
qed

lemma *no-neg-cycle-in-bf*:
assumes *invar-isOptflow state underlying-invars state*
shows $\nexists c.$ *weight* (*a-not-blocked state*) (*a-current-flow state*) $c < 0 \wedge hd\ c =$ *last c*
proof(*rule nexistsI*, *goal-cases*)
case (*1 c*)
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v. prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
by (*simp add*: *bellman-ford*)
have *length-c*: *length* $c \geq 2$
using *1 bellman-ford.weight.simps*[*OF bellman-ford*]
by(*cases c rule*: *list-cases3*) *auto*
have *weight-le-PInfty*:*weight* (*a-not-blocked state*) (*a-current-flow state*) $c <$ *PInfty*
using *1*(*1*) **by** *fastforce*
have *path-with-props*:*awalk UNIV* (*hd c*) (*edges-of-vwalk c*) (*last c*)
weight (*a-not-blocked state*) (*a-current-flow state*) $c =$
ereal (*foldr* (λ*e.* (+) ($\mathfrak{c}$ *e*))
(*map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk c*)) *0*)
($\bigwedge$ *e.* $e \in$*set* (*map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*)
(*prod.snd e*)))
(*edges-of-vwalk c*)) $\Longrightarrow$
a-not-blocked state (*flow-network-spec.oedge e*) $\wedge$ $0 <$ *cost-flow-network.rcap* (*a-current-flow state*) *e*)
using *path-bf-flow-network-path*[*OF - length-c weight-le-PInfty refl*] **by** *auto*
define *cc* **where** *cc* = (*map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk c*))
have *map* (*to-edge* $\circ$ (λ*e. prod.fst* (*local.get-edge-and-costs-forward* (*a-not-blocked state*)
(*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))) (*edges-of-vwalk c*) =
edges-of-vwalk c
apply(*subst* (*2*) *sym*[*OF List.list.map-id*[*of edges-of-vwalk c*]])
apply(*rule map-ext*)
using *cost-flow-network.to-vertex-pair.simps cost-flow-network.*$\mathfrak{c}$*.simps*
by(*auto intro*: *map-ext simp add*: *to-edge-get-edge-and-costs-forward*)
hence *same-edges*:(*map cost-flow-network.to-vertex-pair cc*) = (*edges-of-vwalk c*)
by(*auto simp add*: *cc-def*)
have *c-non-empt*:*cc* $\neq$ []
using *path-with-props*(*1*) *1*(*1*) *awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.vs-to-vertex-pair-pres
by (*auto intro*: *edges-of-vwalk.elims* [*OF sym*[*OF same-edges*]])

moreover have *awalk-f*: *awalk UNIV* (*fstv* (*hd cc*)) (*map cost-flow-network.to-vertex-pair cc*) (*sndv* (*last cc*))
proof−
have *helper*: ⟦ $c = v \# v' \# l$; $cc = z \# zs$; *to-edge* $z = (v, v')$; *map to-edge zs* = *edges-of-vwalk* ($v' \# l$);
awalk UNIV v ((v, v') # *edges-of-vwalk* ($v' \# l$)) (*if* $l = []$ *then* v' *else last* l);$zs \neq []$⟧
$\Longrightarrow$ *awalk UNIV v* ((v, v') # *edges-of-vwalk* ($v' \# l$)) (*prod.snd* (*to-edge* (*last zs*)))
for $v\ v'\ l\ z\ zs$
by(*metis awalk-fst-last last-ConsR last-map list.simps*(*3*) *list.simps*(*9*))
show *?thesis*
apply(*rule edges-of-vwalk.elims* [*OF sym*[*OF same-edges*]])
using *path-with-props*(*1*) *same-edges*
using *1*(*1*) *awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.vs-to-vertex-pair-pres **apply** *auto*[*2*]
using *calculation path-with-props*(*1*) *same-edges*
by(*auto simp add*: *cost-flow-network.vs-to-vertex-pair-pres awalk-intros*(*1*) *intro*: *helper*)
qed
ultimately have *cost-flow-network.prepath cc*
using *prepath-def* **by** *blast*
moreover have *0* < *cost-flow-network.Rcap* (*a-current-flow state*) (*set cc*)
using *cc-def path-with-props*(*3*)
by(*auto simp add*: *cost-flow-network.Rcap-def*)
ultimately have *agpath*:*augpath* (*a-current-flow state*) *cc*
by(*simp add*: *augpath-def*)
have *cc-in-E*: *set cc* ⊆ *EEE*
proof(*rule*, *rule ccontr*, *goal-cases*)
case (*1 e*)
hence *to-edge e* ∈ *set* (*edges-of-vwalk c*)
by (*metis map-in-set same-edges*)
then obtain *c1 c2* **where** *c-split*:*c1*@[*prod.fst* (*to-edge e*)]@[*prod.snd* (*to-edge e*)]@*c2* = *c*
apply(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
subgoal for *e*
using *edges-in-vwalk-split*[*of fst e snd e c*] *cost-flow-network.to-vertex-pair.simps*
multigraph.make-pair **by** *auto*
subgoal for *e*
using *edges-in-vwalk-split*[*of snd e fst e c*] *cost-flow-network.to-vertex-pair.simps*
multigraph.make-pair **by** *auto*
done
have *le-infty*:*prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst* (*to-edge e*))
(*prod.snd* (*to-edge e*))) < *PInfty*
proof(*rule ccontr*, *goal-cases*)
case *1*
hence *prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst* (*cost-flow-network.to-vertex-pair e*))

```
        (prod.snd (cost-flow-network.to-vertex-pair e)))
    = PInfty by simp
   hence weight (a-not-blocked state) (a-current-flow state) c = PInfty
     using bellman-ford.edge-and-Costs-none-pinfty-weight[OF bellman-ford]
          c-split by auto
   thus False
     using weight-le-PInfty by force
  qed
  have one-not-blocked:a-not-blocked state (oedge e)
   using less-PInfty-not-blocked 1(1) cc-def path-with-props(3) by blast
  hence oedge e ∈ ℰ
   using assms(2)
   unfolding algo.underlying-invars-def algo.inv-unbl-iff-forest-active-def
          algo.inv-actives-in-E-def algo.inv-forest-in-E-def
   by auto
  thus ?case
   using 1(2) cost-flow-network.o-edge-res by blast
qed
  obtain C where augcycle (a-current-flow state) C
   apply(rule cost-flow-network.augcycle-from-non-distinct-cycle[OF agpath])
   using 1(1) awalk-f c-non-empt awalk-fst-last[OF - awalk-f]
          awalk-fst-last[OF - path-with-props(1)] same-edges cc-in-E 1(1) cc-def
path-with-props(2)
   by auto
  then show ?case
   using assms(1) invar-isOptflow-def cost-flow-network.min-cost-flow-no-augcycle
by blast
qed


lemma get-target-for-source-ax:
⟦b = balance state; γ = current-γ state; f = current-flow state; Some s = get-source
state;
  get-source-target-path-a state s = Some (t,P); invar-gamma state; invar-isOptflow
state;
   underlying-invars state⟧
  ⟹ t ∈ VV ∧ (abstract-bal-map b) t < − ε ∗ γ ∧ resreach (abstract-flow-map f)
s t ∧ s ≠ t
proof( goal-cases)
  case 1
  note one = this
  have s-prop: s ∈ 𝒱 (1 − local.ε) ∗ γ < abstract-bal-map b s
 using get-source-axioms-red(1)[OF 1(1,2,4)] by auto
 define bf where bf = bellman-ford-forward (a-not-blocked state) (a-current-flow
state) s
  define tt-opt where tt-opt = (get-target-for-source-aux-aux bf
                 (λ v. abstract-real-map (bal-lookup (balance state)) v)
                                (current-γ state) vs)
  show ?thesis
```

proof(*cases tt-opt*)
case *None*
hence *get-source-target-path-a state s = None*
by(*auto simp add*: *option-none-simp*[*of get-target-for-source-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if
tt-opt-def bf-def get-source-target-path-a-def)
hence *False*
using *1* **by** *simp*
thus *?thesis* **by** *simp*
next
case (*Some a*)
define *tt* **where** *tt = the tt-opt*
define *Pbf* **where** *Pbf = search-rev-path-exec s bf tt Nil*
define *PP* **where** *PP = map* (*λe. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*)
(*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*)
have *tt-opt-tt*:*tt-opt = Some tt*
by (*simp add*: *Some tt-def*)
have *Some* (*tt*, *PP*) = *Some* (*t*, *P*)
using *1*
by(*cases tt-opt*)
(*auto simp add*: *option-none-simp*[*of get-target-for-source-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if
tt-opt-def bf-def get-source-target-path-a-def tt-def
PP-def Pbf-def pair-to-realising-redge-forward-def)
hence *tt-is-t*: *tt = t* **and** *PP-is-P*: *PP = P* **by** *auto*
have *t-props*: *tt ∈ set local.vs*
a-balance state tt < − local.ε ∗ current-γ state
prod.snd (*the* (*connection-lookup bf tt*)) < *PInfty*
using *get-target-for-source-aux-aux*(*2*)[*of bf a-balance state current-γ state vs*]
Some
by(*auto simp add*: *tt-def tt-opt-def*)
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (*λ u v. prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
using *bellman-ford* **by** *blast*
define *connections* **where** *connections =*
(*bellman-ford-forward* (*a-not-blocked state*) (*a-current-flow state*) *s*)
have *tt-dist-le-PInfty*:*prod.snd* (*the* (*connection-lookup connections tt*)) < *PInfty*
using *bf-def connections-def t-props*(*3*) **by** *blast*
have *t-prop*:*a-balance state t < − ε ∗ current-γ state ∧*
t ∈ set vs ∧ prod.snd (*the* (*connection-lookup connections t*)) < *PInfty*
using *t-props* **by**(*auto simp add*: *tt-is-t connections-def bf-def*)
have *t-neq-s*: *t ≠ s*
using *t-prop s-prop 1*(*1*) *1*(*2*) *invar-gamma-def*
by (*smt* (*verit*, *best*) *1*(*6*) *mult-minus-left mult-mono′*)
have *t-in-dom*: *t ∈ dom* (*connection-lookup connections*)

using *t-prop*
by (*auto simp add*: *bellman-ford.bellman-ford-init-dom-is*[*OF bellman-ford*]
bellman-ford.same-domain-bellman-ford[*OF bellman-ford*]
connections-def bellman-ford-forward-def
bellman-ford-init-algo-def bellman-ford-algo-def)
hence *pred-of-t-not-None*: *prod.fst* (*the* (*connection-lookup connections t*)) $\neq$ *None*
using *t-neq-s t-prop bellman-ford.bellman-ford-pred-non-infty-pres*[*OF bellman-ford*, *of s length vs* $-$*1*]
by(*auto simp add*: *connections-def bellman-ford-forward-def*
bellman-ford.invar-pred-non-infty-def[*OF bellman-ford*]
bellman-ford-init-algo-def bellman-ford-algo-def)
define *Pbf* **where** *Pbf* = *rev* (*bellman-ford-spec.search-rev-path connection-lookup s connections t*)
have *weight* (*a-not-blocked state*)
(*a-current-flow state*) *Pbf* = *prod.snd* (*the* (*connection-lookup connections t*))
unfolding *Pbf-def*
using *t-prop t-neq-s s-prop vs-is-V pred-of-t-not-None 1*(*7*,*8*)
by(*fastforce simp add*: *bellman-ford-forward-def connections-def*
bellman-ford-init-algo-def bellman-ford-algo-def make-pairs-are
intro!: *bellman-ford.bellman-ford-search-rev-path-weight*[*OF*
bellman-ford no-neg-cycle-in-bf, *of connections s t*]) +
hence *weight-le-PInfty*: *weight* (*a-not-blocked state*) (*a-current-flow state*) *Pbf* < *PInfty*
using *t-prop* **by** *auto*
have *Pbf-opt-path*: *bellman-ford.opt-vs-path vs*
(λ*u v*. *prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *s t*
(*rev* (*bellman-ford-spec.search-rev-path connection-lookup s connections t*))
using *t-prop t-neq-s s-prop*(*1*) *vs-is-V pred-of-t-not-None 1*(*7*,*8*)
by (*auto simp add*: *bellman-ford-forward-def connections-def bellman-ford-init-algo-def*
bellman-ford-algo-def make-pairs-are
intro!: *bellman-ford.computation-of-optimum-path*[*OF bellman-ford no-neg-cycle-in-bf*])
hence *length-Pbf*:*2* $\leq$ *length Pbf*
by(*auto simp add*: *bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def*)
have *awalk UNIV* (*hd Pbf*) (*edges-of-vwalk Pbf*) (*last Pbf*) $\wedge$
weight (*a-not-blocked state*) (*a-current-flow state*) *Pbf* =
ereal (*foldr* (λ*e*. (+) ($\mathfrak{c}$ *e*))
(*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*)) *0*) $\wedge$
($\forall$ *e*$\in$*set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*)
(*prod.snd e*)))
(*edges-of-vwalk Pbf*)).
a-not-blocked state (*flow-network-spec.oedge e*) $\wedge$ *0* < *cost-flow-network.rcap*

(*a-current-flow state*) *e*)
by(*intro path-bf-flow-network-path*[*OF - length-Pbf weight-le-PInfty refl*]) *simp*
hence *Pbf-props*: *awalk UNIV* (*hd Pbf*) (*edges-of-vwalk Pbf*) (*last Pbf*)
weight (*a-not-blocked state*) (*a-current-flow state*) *Pbf* =
ereal (*foldr* (λ*e.* (+) (𝔠 *e*))
(*map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*)) *0*)
(⋀ *e. e* ∈ *set* (*map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*)
(*prod.snd e*)))
(*edges-of-vwalk Pbf*)) ⟹
a-not-blocked state (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap* (*a-current-flow state*) *e*)
by *auto*
define *P* **where** *P* = (*map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*))
have *same-edges*:(*map cost-flow-network.to-vertex-pair P*) = (*edges-of-vwalk Pbf*)
apply(*simp add*: *P-def* , *subst* (*2*) *sym*[*OF List.list.map-id*[*of edges-of-vwalk Pbf*]])
using *get-edge-and-costs-forward-result-props*[*OF prod.collapse*[*symmetric*] *- refl*]
to-edge-get-edge-and-costs-forward
by (*fastforce intro*!: *map-ext*)
moreover have *awalk-f*: *awalk UNIV* (*fstv* (*hd P*)) (*map cost-flow-network.to-vertex-pair P*)
(*sndv* (*last P*))
apply(*rule edges-of-vwalk.elims* [*OF sym*[*OF same-edges*]])
using *Pbf-props*(*1*) *same-edges length-Pbf 1*(*1*) *awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.vs-to-vertex-pair-pres **apply** *auto*[*2*]
using *calculation Pbf-props*(*1*) *same-edges*
by(*auto simp add*: *cost-flow-network.vs-to-vertex-pair-pres awalk-intros*(*1*)
arc-implies-awalk[*OF UNIV-I refl*])
(*metis awalk-fst-last last-ConsR last-map list.simps*(*3*) *list.simps*(*9*))
moreover have *P* ≠ []
using *edges-of-vwalk.simps*(*3*) *length-Pbf same-edges*
by(*cases Pbf rule*: *list-cases3*) *auto*
ultimately have *cost-flow-network.prepath P*
by(*auto simp add*: *cost-flow-network.prepath-def*)
moreover have *0* < *cost-flow-network.Rcap* (*a-current-flow state*) (*set P*)
using *P-def Pbf-props*(*3*)
by(*auto simp add*: *cost-flow-network.Rcap-def*)
ultimately have *augpath* (*a-current-flow state*) *P*
by(*auto simp add*: *cost-flow-network.augpath-def*)
moreover have *fstv* (*hd P*) = *s*
using *awalk-f same-edges Pbf-opt-path awalk-ends*[*OF Pbf-props*(*1*)] *t-neq-s*
by (*force simp add*: *P-def bellman-ford.opt-vs-path-def*[*OF bellman-ford*]

```
                    bellman-ford.vs-path-def[OF bellman-ford] Pbf-def)
  moreover have sndv (last P) = t
    using awalk-f same-edges Pbf-opt-path  awalk-ends[OF Pbf-props(1)] t-neq-s
    by (force simp add: P-def bellman-ford.opt-vs-path-def[OF bellman-ford]
                    bellman-ford.vs-path-def[OF bellman-ford] Pbf-def)
  moreover have set P ⊆ EEE
  proof(rule, rule ccontr, goal-cases)
    case (1 e)
    hence to-edge e ∈ set (edges-of-vwalk Pbf)
      by (metis map-in-set same-edges)
    then obtain c1 c2 where c-split:c1@[prod.fst (to-edge e)]@[prod.snd (to-edge
e)]@c2 = Pbf
      apply(induction e rule: cost-flow-network.to-vertex-pair.induct)
      subgoal for e
        using edges-in-vwalk-split[of fst e snd e Pbf] multigraph.make-pair
        by (auto simp add: Instantiation.make-pair-def)
      subgoal for e
        using edges-in-vwalk-split[of snd e fst e Pbf] multigraph.make-pair
        by (auto simp add: Instantiation.make-pair-def)
    done
 have le-infty:prod.snd (get-edge-and-costs-forward (a-not-blocked state) (a-current-flow
state) (prod.fst (cost-flow-network.to-vertex-pair e))
            (prod.snd (to-edge e))) < PInfty
  proof(rule ccontr, goal-cases)
    case 1
   hence prod.snd (get-edge-and-costs-forward (a-not-blocked state) (a-current-flow
state) (prod.fst (cost-flow-network.to-vertex-pair e))
          (prod.snd (cost-flow-network.to-vertex-pair e)))
     = PInfty by auto
    hence weight (a-not-blocked state) (a-current-flow state) Pbf = PInfty
      using bellman-ford.edge-and-Costs-none-pinfty-weight[OF bellman-ford]
            c-split by auto
    thus False
      using weight-le-PInfty by force
  qed
  have one-not-blocked:a-not-blocked state (oedge e)
    using less-PInfty-not-blocked  1(1) P-def Pbf-props(3) by blast
  hence oedge e ∈ ℰ
    using one(8)
   by(auto elim!: algo.underlying-invarsE algo.inv-unbl-iff-forest-activeE algo.inv-actives-in-EE
algo.inv-forest-in-EE)
  thus ?case
    using 1(2) cost-flow-network.o-edge-res by blast
  qed
  ultimately have resreach (abstract-flow-map f) s t
    using cost-flow-network.augpath-imp-resreach 1(3) by fast
  thus ?thesis
    using 1(1,2) t-prop vs-is-V  t-neq-s by blast
 qed
```

qed

lemma *bf-weight-leq-res-costs*:
assumes *set* (*map oedge qq*) $\subseteq$ *set* $\mathcal{E}$*-list*
$\bigwedge$ *e. e* $\in$ *set qq* $\Longrightarrow$ *a-not-blocked state* (*flow-network-spec.oedge e*)
$\bigwedge$ *e. e* $\in$ *set qq* $\Longrightarrow$ *0* $<$ *cost-flow-network.rcap* (*a-current-flow state*) *e*
unconstrained-awalk (*map cost-flow-network.to-vertex-pair qq*)
and *qq-len*: *length qq* $\geq$ *1*
shows *weight* (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*))
$\leq$ *foldr* (λ*x.* (+) ($\mathfrak{c}$ *x*)) *qq 0*
using *assms*
proof(*induction qq rule*: *list-induct2-len-geq-1*)
case *1*
then show *?case*
using *qq-len* **by** *blast*
next
case (*2 e*)
then show *?case*
by(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
(*fastforce intro*!: *conjunct1*[*OF get-edge-and-costs-forward-makes-cheaper*[*OF*

refl, *of - a-not-blocked state a-current-flow state*]]
intro: *surjective-pairing prod.collapse*
simp add: $\mathcal{E}$*-def* $\mathcal{E}$*-impl*(*1*) $\mathcal{E}$*-list-def to-list*(*1*) *make-pair-fst-snd make-pairs-are*
Instantiation.make-pair-def
simp del: *cost-flow-network.*$\mathfrak{c}$*.simps*)+
next
case (*3 e d xs*)
have *help1*:
$[\![$(*unconstrained-awalk* ((*fst ee*, *snd ee*) # *map to-edge xs*) $\Longrightarrow$
weight (*a-not-blocked state*) (*a-current-flow state*) (*fst ee* # *awalk-verts* (*snd ee*) (*map to-edge xs*))
$\leq$ *ereal* ($\mathfrak{c}$ (*F ee*) + *foldr* (λ*x.* (+) ($\mathfrak{c}$ *x*)) *xs 0*)) ;
($\bigwedge$*e. e* = *F dd* $\vee$ *e* = *F ee* $\vee$ *e* $\in$ *set xs* $\Longrightarrow$ *a-not-blocked state* (*oedge e*)) ;
($\bigwedge$*e. e* = *F dd* $\vee$ *e* = *F ee* $\vee$ *e* $\in$ *set xs* $\Longrightarrow$ *0* $<$ *rcap* (*a-current-flow state*) *e*) ;
unconstrained-awalk ((*fst dd*, *snd dd*) # (*fst ee*, *snd ee*) # *map to-edge xs*) ;
dd $\in$ *set* $\mathcal{E}$*-list* ; *ee* $\in$ *set* $\mathcal{E}$*-list* ; *oedge* ' *set xs* $\subseteq$ *set* $\mathcal{E}$*-list*$]\!]$ $\Longrightarrow$
prod.snd (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*fst dd*) (*fst ee*)) +
weight (*a-not-blocked state*) (*a-current-flow state*) (*fst ee* # *awalk-verts* (*snd ee*) (*map to-edge xs*))
$\leq$ *ereal* ($\mathfrak{c}$ (*F dd*) + ($\mathfrak{c}$ (*F ee*) + *foldr* (λ*x.* (+) ($\mathfrak{c}$ *x*)) *xs 0*))**for** *ee dd*
using *unconstrained-awalk-drop-hd*[*of* (*fst dd*, *snd dd*)]
by(*subst ereal-add-homo*[*of - - + -*])
(*fastforce intro*!: *ordered-ab-semigroup-add-class.add-mono conjunct1*[*OF get-edge-and-costs-forward-makes-cheaper*[*OF*
refl, *of - a-not-blocked state a-current-flow state*]]

intro: *trans*[*OF prod.collapse*]
cong[*OF refl unconstrained-awalk-snd-verts-eq*[*of fst dd snd dd*
fst ee snd ee, *symmetric*]]
simp add: $\mathcal{E}$-*def* $\mathcal{E}$-*impl*(*1*) $\mathcal{E}$-*list-def to-list*(*1*))
have *help2*:
⟦ (*unconstrained-awalk* ((*snd ee*, *fst ee*) # *map to-edge xs*) $\Longrightarrow$
weight (*a-not-blocked state*) (*a-current-flow state*) (*snd ee* # *awalk-verts* (*fst ee*) (*map to-edge xs*))
$\leq$ *ereal* ($\mathfrak{c}$ (*B ee*) + *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *xs 0*)) ;
($\bigwedge e.$ $e = F\ dd \lor e = B\ ee \lor e \in set\ xs \Longrightarrow$ *a-not-blocked state* (*oedge e*)) ;
($\bigwedge e.$ $e = F\ dd \lor e = B\ ee \lor e \in set\ xs \Longrightarrow 0 <$ *rcap* (*a-current-flow state*) *e*) ;
unconstrained-awalk ((*fst dd*, *snd dd*) # (*snd ee*, *fst ee*) # *map to-edge xs*) ;
$dd \in set\ \mathcal{E}$-*list* ; $ee \in set\ \mathcal{E}$-*list*; *oedge* ‘ *set xs* $\subseteq$ *set* $\mathcal{E}$-*list* ⟧ $\Longrightarrow$
prod.snd (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*fst dd*) (*snd ee*)) +
weight (*a-not-blocked state*) (*a-current-flow state*) (*snd ee* # *awalk-verts* (*fst ee*) (*map to-edge xs*))
$\leq$ *ereal* ($\mathfrak{c}$ (*F dd*) + ($\mathfrak{c}$ (*B ee*) + *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *xs 0*))**for** *ee dd*
using *unconstrained-awalk-drop-hd*[*of* (*fst dd*, *snd dd*)]
by(*subst ereal-add-homo*[*of* - - + -])
(*fastforce intro*!: *ordered-ab-semigroup-add-class.add-mono conjunct1*[*OF get-edge-and-costs-forward-makes-cheaper*[*OF*
refl, *of* - *a-not-blocked state a-current-flow state*]]
intro: *trans*[*OF prod.collapse*]
cong[*OF refl unconstrained-awalk-snd-verts-eq*[*of fst dd snd dd*
snd ee fst ee, *symmetric*]]
simp add: $\mathcal{E}$-*def* $\mathcal{E}$-*impl*(*1*) $\mathcal{E}$-*list-def to-list*(*1*))
have *help3*:
⟦(*unconstrained-awalk* ((*fst ee*, *snd ee*) # *map to-edge xs*) $\Longrightarrow$
weight (*a-not-blocked state*) (*a-current-flow state*) (*fst ee* # *awalk-verts* (*snd ee*) (*map to-edge xs*))
$\leq$ *ereal* ($\mathfrak{c}$ (*F ee*) + *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *xs 0*));
($\bigwedge e.$ $e = B\ dd \lor e = F\ ee \lor e \in set\ xs \Longrightarrow$ *a-not-blocked state* (*oedge e*));
($\bigwedge e.$ $e = B\ dd \lor e = F\ ee \lor e \in set\ xs \Longrightarrow 0 <$ *rcap* (*a-current-flow state*) *e*) ;
unconstrained-awalk ((*snd dd*, *fst dd*) # (*fst ee*, *snd ee*) # *map to-edge xs*);
$dd \in set\ \mathcal{E}$-*list*; $ee \in set\ \mathcal{E}$-*list*; *oedge* ‘ *set xs* $\subseteq$ *set* $\mathcal{E}$-*list*⟧ $\Longrightarrow$
prod.snd (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*snd dd*) (*fst ee*)) +
weight (*a-not-blocked state*) (*a-current-flow state*) (*fst ee* # *awalk-verts* (*snd ee*) (*map to-edge xs*))
$\leq$ *ereal* ($\mathfrak{c}$ (*B dd*) + ($\mathfrak{c}$ (*F ee*) + *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *xs 0*))
for *dd ee*
using *unconstrained-awalk-drop-hd*[*of* (*snd dd*, *fst dd*)]
by(*subst ereal-add-homo*[*of* - - + -])
(*fastforce intro*!: *ordered-ab-semigroup-add-class.add-mono conjunct1*[*OF*

get-edge-and-costs-forward-makes-cheaper[*OF*
refl, *of* - *a-not-blocked state a-current-flow state*]]
intro: *trans*[*OF prod.collapse*]
cong[*OF refl unconstrained-awalk-snd-verts-eq*[*of snd dd*
fst dd
fst ee snd ee, *symmetric*]]
simp add: $\mathcal{E}$-*def* $\mathcal{E}$-*impl*(*1*) $\mathcal{E}$-*list-def to-list*(*1*))
have *help4*:
⟦(*unconstrained-awalk* ((*snd ee*, *fst ee*) # *map to-edge xs*) ⟹
weight (*a-not-blocked state*) (*a-current-flow state*) (*snd ee* # *awalk-verts*
(*fst ee*) (*map to-edge xs*))
≤ *ereal* (𝔠 (*B ee*) + *foldr* (λ*x*. (+) (𝔠 *x*)) *xs 0*));
(⋀*e*. *e* = *B dd* ∨ *e* = *B ee* ∨ *e* ∈ *set xs* ⟹ *a-not-blocked state* (*oedge e*));
(⋀*e*. *e* = *B dd* ∨ *e* = *B ee* ∨ *e* ∈ *set xs* ⟹ *0* < *rcap* (*a-current-flow state*)
e);
unconstrained-awalk ((*snd dd*, *fst dd*) # (*snd ee*, *fst ee*) # *map to-edge*
xs);
dd ∈ *set* $\mathcal{E}$-*list* ; *ee* ∈ *set* $\mathcal{E}$-*list* ; *oedge* ‘ *set xs* ⊆ *set* $\mathcal{E}$-*list* ⟧ ⟹
prod.snd (*local.get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow*
state) (*snd dd*) (*snd ee*)) +
weight (*a-not-blocked state*) (*a-current-flow state*) (*snd ee* # *awalk-verts* (*fst*
ee) (*map to-edge xs*))
≤ *ereal* (𝔠 (*B dd*) + (𝔠 (*B ee*) + *foldr* (λ*x*. (+) (𝔠 *x*)) *xs 0*)) **for** *ee dd*
using *unconstrained-awalk-drop-hd*[*of* (*snd dd*, *fst dd*)]
by(*subst ereal-add-homo*[*of* - - + -])
(*fastforce intro*!: *ordered-ab-semigroup-add-class.add-mono conjunct1*[*OF*
get-edge-and-costs-forward-makes-cheaper[*OF*
refl, *of* - *a-not-blocked state a-current-flow state*]]
intro: *trans*[*OF prod.collapse*]
cong[*OF refl unconstrained-awalk-snd-verts-eq*[*of snd dd*
fst dd
snd ee fst ee, *symmetric*]]
simp add: $\mathcal{E}$-*def* $\mathcal{E}$-*impl*(*1*) $\mathcal{E}$-*list-def to-list*(*1*))+
show *?case*
using *3*
by(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*,
all ‹*induction d rule*: *cost-flow-network.to-vertex-pair.induct*›)
(*auto simp add*: *make-pair-fst-snd make-pairs-are Instantiation.make-pair-def*
simp del: *cost-flow-network*.𝔠.*simps*
intro: *help1 help2 help3 help4*)
qed

abbreviation *get-source-target-path-a-cond* ≡ *send-flow-spec.get-source-target-path-a-cond*
lemmas *get-source-target-path-a-cond-def* = *send-flow-spec.get-source-target-path-a-cond-def*
lemmas *get-source-target-path-a-condE* = *send-flow-spec.get-source-target-path-a-condE*

lemma *get-source-target-path-a-ax*:
assumes *get-source-target-path-a-cond state s t P b* γ *f*
shows *cost-flow-network.is-min-path* (*abstract-flow-map f*) *s t P* ∧

oedge ‘ set P $\subseteq$ *to-set* (*actives state*) $\cup$ $\mathcal{F}$ *state* $\wedge$
t $\in \mathcal{V}$ $\wedge$ *abstract-bal-map b t* $< - \varepsilon * \gamma$
proof–
define *bf* **where** *bf* = *bellman-ford-forward* (*a-not-blocked state*) (*a-current-flow state*) *s*
define *tt-opt* **where** *tt-opt* = (*get-target-for-source-aux-aux bf*
(λ *v. abstract-real-map* (*bal-lookup* (*balance state*)) *v*)
(*current-*γ *state*) *vs*)
show *?thesis*
proof(*cases tt-opt*)
case *None*
hence *get-source-target-path-a state s* = *None*
by(*auto simp add*: *option-none-simp*[*of get-target-for-source-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if
tt-opt-def bf-def get-source-target-path-a-def)
hence *False*
using *assms* **by** (*auto elim*: *get-source-target-path-a-condE*)
thus *?thesis* **by** *simp*
next
case (*Some a*)
define *tt* **where** *tt* = *the tt-opt*
define *Pbf* **where** *Pbf* = *search-rev-path-exec s bf tt Nil*
define *PP* **where** *PP* = *map* (λ*e. prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*)
(*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*)
have *tt-opt-tt*:*tt-opt* = *Some tt*
by (*simp add*: *Some tt-def*)
have *Some* (*tt*, *PP*) = *Some* (*t*, *P*)
using *assms*
by(*cases tt-opt*)
(*auto simp add*: *option-none-simp*[*of get-target-for-source-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if
tt-opt-def bf-def get-source-target-path-a-def tt-def
get-source-target-path-a-cond-def PP-def Pbf-def pair-to-realising-redge-forward-def)
hence *tt-is-t*: *tt* = *t* **and** *PP-is-P*: *PP* = *P* **by** *auto*
have *tt-props*: *tt* $\in$ *set local.vs*
a-balance state tt $< -$ *local.*ε $*$ *current-*γ *state*
prod.snd (*the* (*connection-lookup bf tt*)) < *PInfty*
using *get-target-for-source-aux-aux*(*2*)[*of bf a-balance state current-*γ *state vs*]
Some
by(*auto simp add*: *tt-def tt-opt-def*)
have *t-props*:*t* $\in \mathcal{V}$ *abstract-bal-map b t* $< -$ *local.*ε $*$ *current-*γ *state*
resreach (*abstract-flow-map f*) *s t s* $\neq$ *t current-*γ *state* > *0*
using *get-target-for-source-ax*[*of b state*, *OF - refl*, *of f s t P*] *assms*
by(*auto simp add*: *get-source-target-path-a-cond-def make-pairs-are elim*: *algo.invar-gammaE*)
hence *bt-neg*:*abstract-bal-map b t* < *0*
by (*smt* (*verit*, *del-insts*) *local.algo.*ε*-axiom*(*1*) *mult-neg-pos*)
have *s-props*: *s* $\in \mathcal{V}$ (*1* $-$ *local.*ε) $*$ *current-*γ *state* < *abstract-bal-map b s*

using *get-source-axioms-red*(*1*)[*of b state current-*γ *state s*] *assms*
by(*auto simp add*: *get-source-target-path-a-cond-def*)
hence *bs-pos*: *abstract-bal-map b s* $>$ *0*
using *t-props*(*5*) ε*-axiom s-props*(*2*)
by (*auto simp add*: *algebra-simps*)
(*smt* (*verit*, *best*) *mult-less-0-iff s-props*(*2*))
hence *a-balance-s-not-zero*:*a-balance state s* $\neq$ *0*
using *assms* **by**(*force simp add*: *get-source-target-path-a-cond-def*)
have *knowledge*: *True*
$s \in VV$ $t \in VV$ $s \neq t$
underlying-invars state
($\forall$ $e \in \mathcal{F}$ *state*. *0* $<$ *abstract-flow-map f e*)
resreach (*abstract-flow-map f*) *s t*
b $=$ *balance state*
γ $=$ *current-*γ *state*
Some s $=$ *get-source state*
f $=$ *current-flow state*
invar-gamma state
$\neg$ ($\forall$ $v \in VV$. (*abstract-bal-map b*) *v* $=$ *0*)
$\exists$ $s \in VV$. ($1 - \varepsilon$) $* \gamma <$ (*abstract-bal-map b*) *s*
$\exists$ $t \in VV$. *abstract-bal-map b t* $< - \varepsilon * \gamma \wedge$ *resreach* (*abstract-flow-map f*) *s t*
t $=$ *tt* *P* $=$ *PP*
using *assms t-props t-props a-balance-s-not-zero s-props*
by(*auto simp add*: *get-source-target-path-a-cond-def tt-is-t PP-is-P vs-is-V make-pairs-are*)
hence
($\forall$ $e \in$(*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*)).
0 $<$ *a-current-flow state* (*flow-network-spec.oedge e*))
by (*auto simp add*: $\mathcal{F}$*-def*)
have *f-is*: *abstract-flow-map f* $=$ *a-current-flow state*
and *not-blocked-is*: *abstract-not-blocked-map* (*not-blocked state*) $=$ *a-not-blocked state*
using *assms* **by**(*auto simp add*: *get-source-target-path-a-cond-def*)
have *t-prop*: *abstract-bal-map b t* $< - \varepsilon * \gamma$ *resreach* (*abstract-flow-map f*) *s t*
using *knowledge t-props*(*2*) **by** *auto*
then obtain *pp* **where** *pp-prop*:*augpath* (*abstract-flow-map f*) *pp fstv* (*hd pp*) $=$ *s sndv* (*last pp*) $=$ *t set pp* $\subseteq$ *EEE*
using *cost-flow-network.resreach-imp-augpath*[*OF* , *of abstract-flow-map f s t*] **by** *auto*
obtain *ppd* **where** *ppd-props*:*augpath* (*abstract-flow-map f*) *ppd fstv* (*hd ppd*) $=$ *s sndv* (*last ppd*) $=$ *t set ppd* $\subseteq$ *set pp*
distinct ppd
using *pp-prop*
by (*auto intro*: *cost-flow-network.there-is-s-t-path*[*OF* - - - *refl*, *of abstract-flow-map f pp s t*])
obtain *Q* **where** *Q-min*:*cost-flow-network.is-min-path* (*abstract-flow-map f*) *s t Q*
apply(*rule cost-flow-network.there-is-min-path*[*OF* , *of abstract-flow-map f s t ppd*])

using *pp-prop ppd-props cost-flow-network.is-s-t-path-def*
by *auto*
hence *Q-prop*:*augpath* (*abstract-flow-map f*) *Q fstv* (*hd Q*) = *s sndv* (*last Q*) = *t*
set Q ⊆ *EEE distinct Q*
by(*auto simp add*: *cost-flow-network.is-min-path-def*
cost-flow-network.is-s-t-path-def)
have *no-augcycle*: ∄ *C*. *augcycle* (*abstract-flow-map f*) *C*
using *assms cost-flow-network.min-cost-flow-no-augcycle*
by(*auto simp add*: *invar-isOptflow-def get-source-target-path-a-cond-def*)
obtain *qq* **where** *qq-prop*:*augpath* (*abstract-flow-map f*) *qq*
fstv (*hd qq*) = *s*
sndv (*last qq*) = *t*
set qq
⊆ {*e* |*e*. *e* ∈ *EEE* ∧ *flow-network-spec.oedge e* ∈ *to-set* (*actives state*)} ∪
(*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*))
foldr (λ*x*. (+) ($\mathfrak{c}$ *x*)) *qq 0* ≤ *foldr* (λ*x*. (+) ($\mathfrak{c}$ *x*)) *Q 0 qq* ≠ []
using *algo.simulate-inactives-costs*[*OF Q-prop*(*1*−*4*) *knowledge*(*5*) *refl*
f-is refl refl refl refl refl refl knowledge(*4*) - *no-augcycle*]
knowledge(*6*)
by (*auto simp add*: *algo*.$\mathcal{F}$*-redges-def*)
have *qq-len*: *length qq* ≥ *1*
using *qq-prop*(*2*,*3*,*6*) *knowledge*(*4*)
by(*cases qq rule*: *list-cases3*) *auto*
hence *e-in*:*e* ∈ *set qq* ⟹
e ∈ {*e* |*e*. *e* ∈ *EEE* ∧ *flow-network-spec.oedge e* ∈ *to-set* (*actives state*)}
∪ (*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*))
for *e*
using *qq-prop*(*4*) **by** *auto*
hence *e-es*:*e* ∈ *set qq* ⟹ *cost-flow-network.to-vertex-pair e* ∈ *set es* **for** *e*
using *es-E-frac algo.underlying-invars-subs knowledge*(*5*) **by** (*fastforce simp add*: *algo*.$\mathcal{F}$*-redges-def*)
have *e-es'*:*e* ∈ *set qq* ⟹ *oedge e* ∈ $\mathcal{E}$ **for** *e*
using *algo.from-underlying-invars'*(*2*) *cost-flow-network.o-edge-res e-in knowledge*(*5*) **by** *auto*
have *e-in-pp-weight*:*e* ∈ *set qq* ⟹ *prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*)
(*a-current-flow state*) (*fstv e*) (*sndv e*)) < *PInfty* **for** *e*
proof(*goal-cases*)
case *1*
note *e-es*[*OF 1*]
moreover have *oedge-where*: *oedge e* ∈ *to-set* (*actives state*) ∨ *oedge e* ∈ $\mathcal{F}$ *state*
using *e-in 1* **by**(*auto simp add*: $\mathcal{F}$*-def*)
hence *nb*:*a-not-blocked state* (*oedge e*)
using *algo.from-underlying-invars'*(*20*) *knowledge*(*5*) **by** *auto*
have *oedgeE*:*oedge e* ∈ $\mathcal{E}$
using *oedge-where from-underlying-invars'*(*1*,*3*)[*OF knowledge*(*5*)] **by** *auto*
have *prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*)

($fstv$ e) ($sndv$ e)) $\leq \mathfrak{c}$ e
using *nb cost-flow-network.augpath-rcap-pos-strict'*[*OF qq-prop*(*1*) *1*] *knowledge*(*11*)
by(*auto intro*!: *conjunct1*[*OF get-edge-and-costs-forward-makes-cheaper*
[*OF refl oedgeE*, *of a-not-blocked state a-current-flow state*]] *prod.collapse*)
thus *?case* **by** *auto*
qed
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v*. *prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
by (*simp add*: *bellman-ford knowledge*(*2*) *knowledge*(*3*))
have *is-a-walk*:*awalk UNIV s* (*map to-edge qq*) *tt*
using *augpath-def knowledge*(*16*) *prepath-def qq-prop*(*1*) *qq-prop*(*2*) *qq-prop*(*3*) **by** *blast*
hence *vwalk-bet UNIV s* (*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*)) *tt*
using *awalk-imp-vwalk* **by** *force*
moreover have *weight-le-PInfty*:*weight* (*a-not-blocked state*)
(*a-current-flow state*) (*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*)) < *PInfty*
using *path-flow-network-path-bf e-in-pp-weight is-a-walk bellman-ford qq-prop*(*2*) **by** *blast*
have *no-neg-cycle-in-bf*: ∄ *c*. *weight* (*a-not-blocked state*) (*a-current-flow state*) *c* < *0* ∧ *hd c* = *last c*
using *knowledge no-neg-cycle-in-bf assms*
by(*auto elim*!: *get-source-target-path-a-condE*)
have *long-enough*: *2* ≤ *length* (*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*))
using *knowledge*(*4*) *awalk-verts-non-Nil calculation knowledge*(*16*)
hd-of-vwalk-bet'[*OF calculation*] *last-of-vwalk-bet*[*OF calculation*]
by (*cases awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*) *rule*: *list-cases3*) *auto*
have *tt-dist-le-PInfty*:*prod.snd* (*the* (*connection-lookup bf tt*)) < *PInfty*
unfolding *bf-def bellman-ford-forward-def bellman-ford-init-algo-def bellman-ford-algo-def*
using *no-neg-cycle-in-bf knowledge*(*4*,*16*,*2*,*3*) *vs-is-V weight-le-PInfty is-a-walk long-enough*
by (*fastforce intro*!: *bellman-ford.bellamn-ford-path-exists-result-le-PInfty*[*OF bellman-ford*, *of*
- - (*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*))]])
have *t-dist-le-qq-weight*:*prod.snd* (*the* (*connection-lookup bf t*)) ≤
weight (*a-not-blocked state*)
(*a-current-flow state*) (*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*))
using *knowledge*(*4*,*16*,*2*,*3*) *vs-is-V weight-le-PInfty is-a-walk*
bellman-ford.bellman-ford-computes-length-of-optpath[*OF bellman-ford no-neg-cycle-in-bf*, *of s t*]
bellman-ford.opt-vs-path-def[*OF bellman-ford*, *of s t*]
bellman-ford.vsp-pathI[*OF bellman-ford long-enough*, *of s t*]

```
           bellman-ford.weight-le-PInfty-in-vs[OF bellman-ford long-enough, of]
           calculation
   by (auto simp add: vwalk-bet-def bf-def bellman-ford-forward-def bellman-ford-init-algo-def
bellman-ford-algo-def)
  hence t-prop:prod.snd (the (connection-lookup bf t)) < PInfty
    using knowledge(16) tt-dist-le-PInfty by blast
  have t-in-dom: t ∈ dom (connection-lookup  bf)
   using knowledge(3) vs-is-V by (auto simp add: bellman-ford.bellman-ford-init-dom-is[OF
bellman-ford]
                       bellman-ford.same-domain-bellman-ford[OF bellman-ford]
                           bf-def bellman-ford-forward-def bellman-ford-init-algo-def
bellman-ford-algo-def)
  hence pred-of-t-not-None: prod.fst (the (connection-lookup bf t)) ≠ None
      using t-prop knowledge(4) bellman-ford.bellman-ford-pred-non-infty-pres[OF
bellman-ford, of s length vs −1]
    by(auto simp add: bf-def bellman-ford-forward-def
        bellman-ford.invar-pred-non-infty-def[OF bellman-ford]
         bellman-ford-init-algo-def bellman-ford-algo-def)
  have Pbf-def:Pbf = rev (bellford.search-rev-path s bf t)
    unfolding Pbf-def
    using vs-is-V  pred-of-t-not-None t-props
   apply(subst sym[OF arg-cong[of - - rev, OF bellford.function-to-partial-function,
simplified]])
    by(auto simp add: bellman-ford-forward-def bf-def bellman-ford-algo-def
                      bellman-ford-init-algo-def tt-is-t make-pairs-are
             intro!:  bf-fw.search-rev-path-dom-bellman-ford[OF no-neg-cycle-in-bf]
)
  have weight-Pbf-snd: weight (a-not-blocked state)
         (a-current-flow state) Pbf = prod.snd (the (connection-lookup bf t))
    unfolding Pbf-def
    using t-prop   vs-is-V  pred-of-t-not-None knowledge(2,3,4)
   by(fastforce simp add: bellman-ford-forward-def bf-def bellman-ford-init-algo-def
bellman-ford-algo-def
                intro!: bellman-ford.bellman-ford-search-rev-path-weight[OF
               bellman-ford no-neg-cycle-in-bf, of bf s t])+
  hence weight-le-PInfty: weight (a-not-blocked state) (a-current-flow state) Pbf <
PInfty
    using t-prop by auto
  have Pbf-opt-path: bellman-ford.opt-vs-path vs
 (λu v. prod.snd (get-edge-and-costs-forward (a-not-blocked state) (a-current-flow
state) u v)) s t
 (rev (bellford.search-rev-path s bf t))
   using t-prop  vs-is-V  pred-of-t-not-None knowledge(2,3,4)
    by (auto simp add: bellman-ford-forward-def bf-def bellman-ford-init-algo-def
bellman-ford-algo-def
                  intro!: bellman-ford.computation-of-optimum-path[OF bellman-ford
no-neg-cycle-in-bf])
  hence length-Pbf:2 ≤ length Pbf
    using pred-of-t-not-None knowledge(3) vs-is-V
```

unfolding *Pbf-def bf-def bellman-ford-forward-def*
by(*fastforce simp add*: *bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def*
intro: *bf-fw.search-rev-path-dom-bellman-ford*[*OF no-neg-cycle-in-bf*])+
have *awalk UNIV* (*hd Pbf*) (*edges-of-vwalk Pbf*) (*last Pbf*) ∧
weight (*a-not-blocked state*) (*a-current-flow state*) *Pbf* =
ereal (*foldr* (λ*e*. (+) (𝔠 *e*))
(*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*)) *0*) ∧
(∀ *e*∈*set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*)
(*prod.snd e*)))
(*edges-of-vwalk Pbf*)).
a-not-blocked state (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap* (*a-current-flow state*) *e*)
by(*intro path-bf-flow-network-path*[*OF - length-Pbf weight-le-PInfty refl*]) *simp*
hence *Pbf-props*: *awalk UNIV* (*hd Pbf*) (*edges-of-vwalk Pbf*) (*last Pbf*)
weight (*a-not-blocked state*) (*a-current-flow state*) *Pbf* =
ereal (*foldr* (λ*e*. (+) (𝔠 *e*))
(*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*) (*prod.snd e*)))
(*edges-of-vwalk Pbf*)) *0*)
(⋀ *e*. *e* ∈ *set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*)
(*prod.snd e*)))
(*edges-of-vwalk Pbf*)) ⟹
a-not-blocked state (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap* (*a-current-flow state*) *e*)
by *auto*
have *map* (*to-edge* ∘
(λ*e*. *prod.fst* (*local.get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.fst e*)
(*prod.snd e*)))) (*edges-of-vwalk Pbf*) =
edges-of-vwalk Pbf
apply(*subst* (*2*) *sym*[*OF List.list.map-id*[*of edges-of-vwalk Pbf*]])
apply(*rule map-ext*)
using *cost-flow-network.to-vertex-pair.simps cost-flow-network.*𝔠*.simps*
by(*auto intro*: *map-ext simp add*: *to-edge-get-edge-and-costs-forward*)
hence *same-edges*:(*map cost-flow-network.to-vertex-pair PP*) = (*edges-of-vwalk Pbf*)
by(*auto simp add*: *PP-def*)
moreover have *awalk-f*: *awalk UNIV* (*fstv* (*hd PP*)) (*map cost-flow-network.to-vertex-pair PP*)
(*sndv* (*last PP*))
apply(*rule edges-of-vwalk.elims* [*OF sym*[*OF same-edges*]])
using *Pbf-props*(*1*) *same-edges length-Pbf awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.vs-to-vertex-pair-pres **apply** *auto*[*2*]

```
    using calculation  Pbf-props(1) same-edges
    by(auto simp add: cost-flow-network.vs-to-vertex-pair-pres awalk-intros(1)
                      arc-implies-awalk[OF UNIV-I refl])
    (metis awalk-fst-last last-ConsR last-map list.simps(3) list.simps(9))
  moreover have PP ≠ []
    using  edges-of-vwalk.simps(3) length-Pbf same-edges
    by(cases Pbf rule: list-cases3) auto
  ultimately have cost-flow-network.prepath PP
  by(auto simp add:cost-flow-network.prepath-def )
  moreover have Rcap-P:0 < cost-flow-network.Rcap (a-current-flow state) (set PP)
    using PP-def Pbf-props(3)
    by(auto simp add: cost-flow-network.Rcap-def)
  ultimately have augpath (a-current-flow state) PP
    by(auto simp add: cost-flow-network.augpath-def)
  moreover have fstv (hd PP) = s
     using awalk-f same-edges Pbf-opt-path  awalk-ends[OF Pbf-props(1)] knowledge(4)
    by (force simp add: PP-def bellman-ford.opt-vs-path-def[OF bellman-ford]
                      bellman-ford.vs-path-def[OF bellman-ford] Pbf-def)
  moreover have sndv (last PP) = t
     using awalk-f same-edges Pbf-opt-path  awalk-ends[OF Pbf-props(1)]  knowledge(4)
    by (force simp add: PP-def bellman-ford.opt-vs-path-def[OF bellman-ford]
                      bellman-ford.vs-path-def[OF bellman-ford] Pbf-def)
  moreover have oedge-of-p-allowed:oedge ‘ (set PP) ⊆ to-set (actives state) ∪ ℱ state
  proof(rule, rule ccontr, goal-cases)
    case (1 e)
    have a-not-blocked state e
      using map-in-set same-edges 1(1) PP-def Pbf-props(3) list.set-map by blast
    thus ?case
      using 1(2) algo.from-underlying-invars'(20) knowledge(5) by force
  qed
  have distinct-Pbf: distinct Pbf
    using no-neg-cycle-in-bf knowledge(2,3,4) vs-is-V pred-of-t-not-None
    unfolding Pbf-def bf-def
    by (fastforce intro!: bellman-ford.search-rev-path-distinct[OF bellman-ford]
           simp add: bellman-ford-forward-def bf-def bellman-ford-init-algo-def bellman-ford-algo-def)
  have distinctP:distinct PP
    using distinct-edges-of-vwalk[OF distinct-Pbf, simplified sym[OF same-edges ]]
        distinct-map by auto
  have qq-in-E:set (map cost-flow-network.to-vertex-pair qq) ⊆ set es
    using e-es by auto
  have qq-in-E':set (map flow-network-spec.oedge qq) ⊆ ℰ
    using e-es' by auto
  have not-blocked-qq: ⋀ e . e ∈ set qq ⟹ a-not-blocked state (oedge e)
    using  algo.from-underlying-invars'(20) e-in knowledge(5) by (fastforce simp
```

add: *ℱ-def*)
have *rcap-qq*: $\bigwedge$ *e* . *e* ∈ *set qq* ⟹ *cost-flow-network.rcap* (*a-current-flow state*) *e* > *0*
using *cost-flow-network.augpath-rcap-pos-strict′*[*OF qq-prop*(*1*)] *knowledge* **by** *simp*
have *awalk′*: *unconstrained-awalk* (*map cost-flow-network.to-vertex-pair qq*)
by (*meson unconstrained-awalk-def awalkE′ is-a-walk*)
have *bf-weight-leq-res-costs*:*weight* (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map cost-flow-network.to-vertex-pair qq*))
≤ *foldr* (λ*x*. (+) (𝔠 *x*)) *qq 0*
using *qq-in-E not-blocked-qq rcap-qq awalk′ qq-len e-es′*
by(*auto intro*!: *bf-weight-leq-res-costs simp add*: *ℰ-def ℰ-impl*(*1*) *ℰ-list-def to-list*(*1*))
have *oedge-of-EE*: *flow-network-spec.oedge* ‘ *EEE* = *ℰ*
by (*meson cost-flow-network.oedge-on-𝔈*)
have *flow-network-spec.oedge* ‘ *set PP* ⊆ *ℰ*
using *from-underlying-invars′*(*1*,*3*)[*OF knowledge*(*5*)] *oedge-of-p-allowed* **by** *blast*
hence *P-in-E*: *set PP* ⊆ *EEE*
by (*meson image-subset-iff cost-flow-network.o-edge-res subsetI*)
have (*foldr* (λ*e*. (+) (𝔠 *e*)) *PP 0*) ≤ *foldr* (λ*x*. (+) (𝔠 *x*)) *Q 0*
using *weight-Pbf-snd t-dist-le-qq-weight Pbf-props*(*2*)[*simplified sym*[*OF PP-def*]]
qq-prop(*5*) *bf-weight-leq-res-costs*
by (*smt* (*verit*, *best*) *leD le-ereal-less*)
moreover have (*foldr* (λ*e*. (+) (𝔠 *e*)) *PP 0*) = *cost-flow-network*.ℭ *PP*
unfolding *cost-flow-network*.ℭ*-def*
by(*subst distinct-sum*, *simp add*: *distinctP*, *meson add.commute*)
moreover have (*foldr* (λ*e*. (+) (𝔠 *e*)) *Q 0*) = *cost-flow-network*.ℭ *Q*
unfolding *cost-flow-network*.ℭ*-def*
by(*subst distinct-sum*, *simp add*: *Q-prop*(*5*), *meson add.commute*)
ultimately have *P-min*: *cost-flow-network.is-min-path* (*abstract-flow-map f*) *s t PP*
using *Q-min P-in-E knowledge*(*11*) *distinctP*
by(*auto simp add*: *cost-flow-network.is-min-path-def*
cost-flow-network.is-s-t-path-def)
show *?thesis*
using *P-min distinctP Rcap-P oedge-of-p-allowed PP-is-P knowledge*(*9*)
t-props(*1*,*2*) **by** *fastforce*
qed
qed

lemma *path-flow-network-path-bf-backward*:
assumes *e-weight*:$\bigwedge$ *e*. *e* ∈ *set pp* ⟹ *prod.snd* (*get-edge-and-costs-backward nb f* (*fstv e*) (*sndv e*)) < *PInfty*
and *is-a-walk*:*awalk UNIV s* (*map to-edge pp*) *tt*
and *s-is-fstv*: *s* = *fstv* (*hd pp*)
and *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar*
connection-delete es vs (λ *u v*. *prod.snd*

```
                     (get-edge-and-costs-backward nb f u v)) connection-update

 shows weight-backward nb f (awalk-verts s (map cost-flow-network.to-vertex-pair
pp)) < PInfty
 using assms(1,2)[simplified assms(3)]
proof(subst assms(3), induction pp  rule: list-induct3)
 case 1
 then show ?case
  using  bellman-ford.weight.simps[OF bellman-ford] by auto
next
 case (2 x)
 then show ?case
  using  bellman-ford.weight.simps[OF bellman-ford] apply auto[1]
  apply(induction x rule: cost-flow-network.to-vertex-pair.induct)
  apply(simp add: cost-flow-network.to-vertex-pair.simps make-pairs-are
          bellman-ford.weight.simps[OF bellman-ford] make-pair-fst-snd
          Instantiation.make-pair-def)+
  done
next
 case (3 e d es)
  have same-ends:sndv e = fstv d
  using 3(3)
  by(induction e rule: cost-flow-network.to-vertex-pair.induct)
   (auto intro: cost-flow-network.to-vertex-pair.induct[OF , of - d]
     simp add: bellman-ford.weight.simps[OF bellman-ford]  Awalk.awalk-simps
make-pair-fst-snd
          cost-flow-network.vs-to-vertex-pair-pres(1) make-pairs-are Instantia-
tion.make-pair-def)
 have weight-backward nb f
    (awalk-verts (fstv (hd ((e # d # es)))) (map cost-flow-network.to-vertex-pair
(e # d # es))) =
     prod.snd (get-edge-and-costs-backward nb f (fstv e) (sndv e))
   + weight-backward nb f (awalk-verts (fstv (hd (( d # es)))) (map cost-flow-network.to-vertex-pair
(d # es)))
   using same-ends
  by(induction e rule: cost-flow-network.to-vertex-pair.induct)
   (auto intro:  cost-flow-network.to-vertex-pair.induct[OF , of - d]
      simp add:  bellman-ford.weight.simps[OF bellman-ford]
          cost-flow-network.to-vertex-pair-fst-snd multigraph.make-pair)
  moreover have prod.snd (get-edge-and-costs-backward nb f (fstv e) (sndv e))
< PInfty
   using 3.prems(1) by force
  moreover have weight-backward nb f (awalk-verts (fstv (hd (( d # es)))) (map
cost-flow-network.to-vertex-pair (d # es))) < PInfty
      using 3(2,3)
      by(intro  3(1), auto intro: cost-flow-network.to-vertex-pair.induct[OF , of
- e]
       simp add: bellman-ford.weight.simps[OF bellman-ford] Awalk.awalk-simps(2)[of
UNIV]
```

cost-flow-network.vs-to-vertex-pair-pres(*1*))
ultimately show *?case* **by** *simp*
qed

lemma *path-bf-flow-network-path-backward*:
assumes *True*
and *len*: *length pp* ≥ *2*
and *weight-backward nb f pp* < *PInfty*
ppp = *edges-of-vwalk pp*
shows *awalk UNIV* (*last pp*) (*map prod.swap* (*rev ppp*)) (*hd pp*) ∧
weight-backward nb f pp = *foldr* (λ *e acc*. 𝔠 *e* + *acc*)
(*map* (λ *e*. (*prod.fst* (*get-edge-and-costs-backward nb f* (*prod.snd e*)
(*prod.fst e*)))) (*map prod.swap* (*rev ppp*))) *0*
∧ (∀ *e* ∈ *set* (*map* (λ *e*. (*prod.fst* (*get-edge-and-costs-backward nb f*
(*prod.snd e*)(*prod.fst e*)))) (*map prod.swap* (*rev ppp*))).
nb (*oedge e*) ∧ *cost-flow-network.rcap f e* > *0*)
proof−
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v*. *prod.snd* (*get-edge-and-costs-backward nb f u v*)) *connection-update*
by (*simp add*: *bellman-ford-backward*)
show *?thesis*
using *assms*(*3*−)
proof(*induction pp arbitrary*: *ppp rule*: *list-induct3-len-geq-2*)
case *1*
then show *?case*
using *len* **by** *simp*
next
case (*2 x y*)
have *awalk UNIV* (*last* [*x*, *y*]) (*map prod.swap* (*rev ppp*)) (*hd* [*x*, *y*])
using *2* **unfolding** *get-edge-and-costs-forward-def Let-def*
by (*auto simp add*: *arc-implies-awalk bellman-ford.weight.simps*[*OF bellman-ford*]

split: *if-split prod.split*)
moreover have *weight-backward nb f* [*x*, *y*] =
ereal
(*foldr* (λ*e*. (+) (𝔠 *e*)) (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-backward nb f*
(*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev ppp*))) *0*)
using *2.prems*(*1*)
by(*auto simp add*: *es-sym*[*of* (*y*,*x*)] *bellman-ford.weight.simps*[*OF bellman-ford*]
2(*2*) *get-edge-and-costs-backward-result-props*)
moreover have (∀ *e*∈*set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-backward nb f*
(*prod.snd e*) (*prod.fst e*))) (*map prod.swap* (*rev ppp*))).
nb (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap f e*)
using *2* *get-edge-and-costs-backward-result-props*[*OF prod.collapse*[*symmetric*]
- *refl*, *of nb f x y*]
by *auto*
ultimately show *?case* **by** *simp*

next
case (*3 x y xs*)
thm *3*(*1*)[*OF* - *refl*]
have *awalk UNIV* (*last* (*x* # *y* # *xs*)) (*map prod.swap* (*rev ppp*)) (*hd* (*x* # *y* # *xs*))
using *3.IH 3.prems*(*1*) *3.prems*(*2*) *Awalk.awalk-simps*(*2*)
bellman-ford.weight.simps(*3*)[*OF bellman-ford*] *edges-of-vwalk.simps*(*3*)
by (*auto simp add*: *arc-implies-awalk*)

moreover have *weight-backward nb f* (*x* # *y* # *xs*) = *prod.snd* (*get-edge-and-costs-backward nb f x y*) +
weight-backward nb f (*y* # *xs*)
using *bellman-ford bellman-ford.weight.simps*(*3*) **by** *fastforce*
moreover have *weight-backward nb f* (*y* # *xs*) =
ereal
(*foldr* (*λe.* (+) (𝔠 *e*))
(*map* (*λe. prod.fst* (*get-edge-and-costs-backward nb f* (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk* (*y* # *xs*))))) *0*)
using *3.IH 3.prems*(*1*) *calculation*(*2*) **by** *fastforce*
moreover have *prod.snd* (*get-edge-and-costs-backward nb f x y*) =
𝔠 (*prod.fst* (*get-edge-and-costs-backward nb f x y*))
using *3 get-edge-and-costs-backward-result-props*[*OF prod.collapse*[*symmetric*] - *refl*, *of nb f x y*]
by *auto*
moreover have (∀ *e*∈*set* (*map* (*λe. prod.fst* (*get-edge-and-costs-backward nb f* (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk* (*y* # *xs*))))).
nb (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap f e*)
by (*simp add*: *3.IH calculation*(*3*))
moreover have *nb* (*flow-network-spec.oedge* (*prod.fst* (*get-edge-and-costs-backward nb f x y*)))
using *3 get-edge-and-costs-backward-result-props*[*OF prod.collapse*[*symmetric*] - *refl*, *of nb f x y*]
by *auto*
moreover have *0* < *cost-flow-network.rcap f* (*prod.fst* (*get-edge-and-costs-backward nb f x y*))
using *3 get-edge-and-costs-backward-result-props*[*OF prod.collapse*[*symmetric*] - *refl*, *of nb f x y*]
by *auto*
ultimately show *?case*
by (*auto simp add*: *3*(*3*) *foldr-plus-add-right*[**where** *b* = *0*, *simplified*])
qed
qed

lemma *edges-of-vwalk-rev-swap*:(*map prod.swap* (*rev* (*edges-of-vwalk c*))) = *edges-of-vwalk* (*rev c*)
apply(*induction c rule*: *edges-of-vwalk.induct*, *simp*, *simp*)
subgoal for *x y rest*
using *edges-of-vwalk-append-2*[*of* [*y*,*x*]]

by *auto*
done

lemma *no-neg-cycle-in-bf-backward*:
assumes *invar-isOptflow state underlying-invars state*
shows $\nexists$*c. weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *c* < *0* ∧ *hd c* = *last c*
proof(*rule nexistsI*, *goal-cases*)
case (*1 c*)
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v*. *prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
by (*simp add*: *bellman-ford-backward*)
have *length-c*: *length c* ≥ *2*
using *1 bellman-ford.weight.simps*[*OF bellman-ford*]
by(*cases c rule*: *list-cases3*) *auto*
have *weight-le-PInfty*:*weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *c* < *PInfty*
using *1*(*1*) **by** *fastforce*
have *path-with-props*:*awalk UNIV* (*last c*) (*map prod.swap* (*rev* (*edges-of-vwalk c*))) (*hd c*)
weight-backward (*a-not-blocked state*) (*a-current-flow state*) *c* =
ereal
(*foldr* (λ*e*. (+) (𝔠 *e*))
(*map* (λ*e*. *prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk c*))))
0)
(⋀ *e*. *e*∈*set* (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk c*)))) ⟹
a-not-blocked state (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap* (*a-current-flow state*) *e*)
using *path-bf-flow-network-path-backward*[*OF - length-c weight-le-PInfty refl*]
by *auto*
define *cc* **where** *cc* = (*map* (λ*e*. *prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk c*))))
have *same-edges*:(*map cost-flow-network.to-vertex-pair cc*) = (*map prod.swap* (*rev* (*edges-of-vwalk c*)))
using *to-edge-get-edge-and-costs-backward* **by** (*force simp add*: *cc-def*)
have *c-non-empt*:*cc* ≠ []
using *path-with-props*(*1*) *1*(*1*) *awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.vs-to-vertex-pair-pres
by (*auto intro*: *edges-of-vwalk.elims*[*OF sym*[*OF same-edges*[*simplified edges-of-vwalk-rev-swap*]]])
moreover have *awalk-f*: *awalk UNIV* (*fstv* (*hd cc*)) (*map cost-flow-network.to-vertex-pair cc*) (*sndv* (*last cc*))

apply(*rule edges-of-vwalk.elims* [*OF sym*[*OF same-edges*[*simplified edges-of-vwalk-rev-swap*]]])
using *path-with-props*(*1*) *same-edges*
using *1*(*1*) *awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
apply *auto*[*2*]
using *calculation path-with-props*(*1*) *same-edges*
by (*auto simp add*: *cost-flow-network.vs-to-vertex-pair-pres awalk-intros*(*1*)
arc-implies-awalk[*OF UNIV-I refl*])
(*metis awalk-fst-last last-ConsR last-map list.simps*(*3*) *list.simps*(*9*))
ultimately have *cost-flow-network.prepath cc*
using *prepath-def* **by** *blast*
moreover have $0 <$ *cost-flow-network.Rcap* (*a-current-flow state*) (*set cc*)
using *cc-def path-with-props*(*3*)
by(*auto simp add*: *cost-flow-network.Rcap-def*)
ultimately have *agpath*:*augpath* (*a-current-flow state*) *cc*
by(*simp add*: *augpath-def*)
have *cc-in-E*: *set cc* $\subseteq$ *EEE*
proof(*rule*, *rule ccontr*, *goal-cases*)
case (*1 e*)
hence *to-edge e* $\in$ *set* (*edges-of-vwalk* (*rev c*))
by (*metis map-in-set same-edges*[*simplified edges-of-vwalk-rev-swap*])
then obtain *c1 c2* **where** *c-split*:*c1*@[*prod.fst* (*to-edge e*)]@[*prod.snd* (*to-edge e*)]@*c2* = *rev c*
apply(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
subgoal for *e*
using *edges-in-vwalk-split*[*of fst e snd e rev c*] *multigraph.make-pair′*
by (*auto simp add*: *Instantiation.make-pair-def*)
subgoal for *e*
using *edges-in-vwalk-split*[*of snd e fst e rev c*] *multigraph.make-pair′*
by (*auto simp add*: *Instantiation.make-pair-def*)
done
have *c-split*:*rev c2*@[*prod.snd* (*to-edge e*)]@[*prod.fst* (*to-edge e*)]@ *rev c1* = *c*
apply(*subst sym*[*OF rev-rev-ident*[*of c*]])
apply(*subst sym*[*OF c-split*])
by *simp*
have *le-infty*:*prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd* (*to-edge e*))
(*prod.fst* (*to-edge e*))) < *PInfty*
proof(*rule ccontr*, *goal-cases*)
case *1*
hence *prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd* (*cost-flow-network.to-vertex-pair e*))
(*prod.fst* (*cost-flow-network.to-vertex-pair e*)))
= *PInfty* **by** *simp*
hence *weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *c* = *PInfty*
using *bellman-ford.edge-and-Costs-none-pinfty-weight*[*OF bellman-ford*]
c-split **by** *auto*
thus *False*
using *weight-le-PInfty* **by** *force*
qed

have *one-not-blocked*:*a-not-blocked state* (*oedge e*)
using *less-PInfty-not-blocked 1*(*1*) *cc-def path-with-props*(*3*) **by** *blast*
hence *oedge e* ∈ $\mathcal{E}$
using *assms*(*2*)
by(*auto elim*!: *algo.underlying-invarsE algo.inv-unbl-iff-forest-activeE algo.inv-actives-in-EE algo.inv-forest-in-EE*)
thus *?case*
using *1*(*2*) *cost-flow-network.o-edge-res* **by** *blast*
qed
obtain *C* **where** *augcycle* (*a-current-flow state*) *C*
apply(*rule cost-flow-network.augcycle-from-non-distinct-cycle*[*OF agpath*])
using *1*(*1*) *awalk-f c-non-empt awalk-fst-last*[*OF - awalk-f*]
awalk-fst-last[*OF - path-with-props*(*1*)] *cc-in-E 1*(*1*) *cc-def path-with-props*(*2*)
by (*auto*, *metis list.map-comp same-edges*)
then show *?case*
using *assms*(*1*) *invar-isOptflow-def cost-flow-network.min-cost-flow-no-augcycle*
by *blast*
qed

lemma *to-edge-of-get-edge-and-costs-backward*:
cost-flow-network.to-vertex-pair (*prod.fst* (*get-edge-and-costs-backward* (*not-blocked state*)
(*current-flow state*) *a b*)) = (*b*, *a*)
using *to-edge-get-edge-and-costs-backward* **by** *force*

lemma *get-source-for-target-ax*:
⟦*b* = *balance state*; γ = *current-*γ *state*; *f* = *current-flow state*; *Some t* = *get-target state*;
get-source-target-path-b state t = *Some* (*s*,*P*); *invar-gamma state*; *invar-isOptflow state*;
underlying-invars state⟧
⟹ *s* ∈ *VV* ∧ (*abstract-bal-map b*) *s* > $\varepsilon * \gamma$ ∧ *resreach* (*abstract-flow-map f*) *s t* ∧ *s* ≠ *t*
proof(*goal-cases*)
case *1*
note *one* = *this*
have *t-prop*: *t* ∈ $\mathcal{V}$ − (*1* − *local.*ε) ∗ γ > *abstract-bal-map b t*
using *get-target-axioms-red*(*1*)[*OF 1*(*1*,*2*,*4*)] **by** *auto*
define *bf* **where** *bf* = *bellman-ford-backward* (*a-not-blocked state*) (*a-current-flow state*) *t*
define *ss-opt* **where** *ss-opt* = (*get-source-for-target-aux-aux bf*
(λ *v*. *abstract-real-map* (*bal-lookup* (*balance state*)) *v*)
(*current-*γ *state*) *vs*)
show *?thesis*
proof(*cases ss-opt*)
case *None*
hence *get-source-target-path-b state t* = *None*
by(*auto simp add*: *option-none-simp*[*of get-source-for-target-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if

ss-opt-def bf-def get-source-target-path-b-def)
hence *False*
using *1* **by** *simp*
thus *?thesis* **by** *simp*
next
case (*Some a*)
define *ss* **where** *ss* = *the ss-opt*
define *Pbf* **where** *Pbf* = *rev* (*search-rev-path-exec t bf ss Nil*)
define *PP* **where** *PP* = *map* (λ*e. prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*)
(*prod.snd e*) (*prod.fst e*)))
(*edges-of-vwalk Pbf*)
have *ss-opt-ss*:*ss-opt* = *Some ss*
by (*simp add*: *Some ss-def*)
have *Some* (*ss*, *PP*) = *Some* (*s*, *P*)
using *1*
by(*cases ss-opt*)
(*auto simp add*: *option-none-simp*[*of get-source-for-target-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if
ss-opt-def bf-def get-source-target-path-b-def ss-def
PP-def Pbf-def pair-to-realising-redge-backward-def)
hence *ss-is-s*: *ss* = *s* **and** *PP-is-P*: *PP* = *P* **by** *auto*
have *s-props*: *ss* $\in$ *set local.vs*
a-balance state ss > *local.*ε $*$ *current-*γ *state*
prod.snd (*the* (*connection-lookup bf ss*)) < *PInfty*
using *get-source-for-target-aux-aux*(*2*)[*of bf a-balance state current-*γ *state vs*]
Some
by(*auto simp add*: *ss-def ss-opt-def*)
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v. prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
using *bellman-ford-backward* **by** *blast*
define *connections* **where** *connections* =
(*bellman-ford-backward* (*a-not-blocked state*) (*a-current-flow state*) *t*)
have *ss-dist-le-PInfty*:*prod.snd* (*the* (*connection-lookup connections ss*)) < *PInfty*
using *bf-def connections-def s-props*(*3*) **by** *blast*
have *s-prop*:*a-balance state s* > ε $*$ *current-*γ *state* $\wedge$
s $\in$ *set vs* $\wedge$ *prod.snd* (*the* (*connection-lookup connections s*)) < *PInfty*
using *s-props* **by**(*auto simp add*: *ss-is-s connections-def bf-def*)
have *s-neq-t*: *s* $\neq$ *t*
using *t-prop s-prop 1*(*1*) *1*(*2*) *invar-gamma-def*
by (*smt* (*verit*, *best*) *1*(*6*) *mult-minus-left mult-mono'*)
have *s-in-dom*: *s* $\in$ *dom* (*connection-lookup connections*)
using *s-prop*
by (*auto simp add*: *bellman-ford.bellman-ford-init-dom-is*[*OF bellman-ford*]
bellman-ford.same-domain-bellman-ford[*OF bellman-ford*]
connections-def bellman-ford-backward-def
bellman-ford-init-algo-def bellman-ford-algo-def)

hence *pred-of-s-not-None*: *prod.fst* (*the* (*connection-lookup connections s*)) $\neq$ *None*

using *s-neq-t s-prop bellman-ford.bellman-ford-pred-non-infty-pres*[*OF bellman-ford*, *of t length vs* -1]

by(*simp add*: *connections-def bellman-ford-backward-def*
bellman-ford.invar-pred-non-infty-def[*OF bellman-ford*]
bellman-ford-init-algo-def bellman-ford-algo-def)

define *Pbf* **where** *Pbf* = *rev* (*bellman-ford-spec.search-rev-path connection-lookup t connections s*)

have *no-neg-cycle-in-bf*: $\nexists$ *c. weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *c* $<$ *0* $\wedge$ *hd c* = *last c*

using *1*(*7*,*8*) *no-neg-cycle-in-bf-backward* **by** *blast*

have *weight-backward* (*a-not-blocked state*)
(*a-current-flow state*) *Pbf* = *prod.snd* (*the* (*connection-lookup connections s*))

unfolding *Pbf-def*

using *s-prop s-neq-t t-prop vs-is-V pred-of-s-not-None 1*(*7*,*8*)

by(*fastforce simp add*: *bellman-ford-backward-def connections-def*
bellman-ford-init-algo-def bellman-ford-algo-def make-pairs-are
intro!: *bellman-ford.bellman-ford-search-rev-path-weight*[*OF*
bellman-ford no-neg-cycle-in-bf, *of connections t s*]) +

hence *weight-le-PInfty*: *weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *Pbf* $<$ *PInfty*

using *s-prop* **by** *auto*

have *Pbf-opt-path*: *bellman-ford.opt-vs-path vs*
(λ*u v. prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *t s*
(*rev* (*bellford.search-rev-path t connections s*))

using *t-prop s-neq-t s-prop*(*1*) *vs-is-V pred-of-s-not-None*

by (*auto simp add*: *bellman-ford-backward-def connections-def bellman-ford-algo-def*

bellman-ford-init-algo-def make-pairs-are
intro!: *bellman-ford.computation-of-optimum-path*[*OF bellman-ford no-neg-cycle-in-bf*])

hence *length-Pbf*:*2* $\leq$ *length Pbf*

by(*auto simp add*: *bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def*)

have *Pbf-props*: *awalk UNIV* (*last Pbf*) (*map prod.swap* (*rev* (*edges-of-vwalk Pbf*))) (*hd Pbf*)
weight-backward (*a-not-blocked state*) (*a-current-flow state*) *Pbf* = *ereal*
(*foldr* (λ*e.* (+) ($\mathfrak{c}$ *e*))
(*map* (λ*e. prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk Pbf*))))
0)
($\bigwedge$ *e. e* $\in$*set* (*map* (λ*e. prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk Pbf*)))) $\Longrightarrow$

a-not-blocked state (*flow-network-spec.oedge e*) ∧ *0* < *cost-flow-network.rcap* (*a-current-flow state*) *e*)
using *path-bf-flow-network-path-backward*[*OF - length-Pbf weight-le-PInfty refl*]
by *auto*
define *P* **where** *P* = (*map* (*λe. prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd e*) (*prod.fst e*)))
(*map prod.swap* (*rev* (*edges-of-vwalk Pbf*))))
have *same-edges*:(*map cost-flow-network.to-vertex-pair P*) = (*map prod.swap* (*rev* (*edges-of-vwalk Pbf*)))
using *to-edge-get-edge-and-costs-forward*
by (*auto simp add*: *get-edge-and-costs-forward-def P-def get-edge-and-costs-backward-def*)
moreover have *awalk-f*:
awalk UNIV (*fstv* (*hd P*)) (*map cost-flow-network.to-vertex-pair P*) (*sndv* (*last P*))
apply(*rule edges-of-vwalk.elims* [*OF sym*[*OF same-edges*[*simplified edges-of-vwalk-rev-swap*]]])
using *Pbf-props*(*1*) *same-edges length-Pbf 1*(*1*) *awalk-fst-last bellman-ford.weight.simps*[*OF bellman-ford*]
cost-flow-network.vs-to-vertex-pair-pres **apply** *auto*[*2*]
using *calculation Pbf-props*(*1*) *same-edges*
by(*auto simp add*: *cost-flow-network.vs-to-vertex-pair-pres awalk-intros*(*1*)
arc-implies-awalk[*OF UNIV-I refl*])
(*metis awalk-fst-last last-ConsR last-map list.simps*(*3*) *list.simps*(*9*))
moreover have *P* ≠ []
using *edges-of-vwalk.simps*(*3*) *length-Pbf same-edges*
by(*cases Pbf rule*: *list-cases3*) *auto*
ultimately have *cost-flow-network.prepath P*
by(*auto simp add*:*cost-flow-network.prepath-def*)
moreover have *0* < *cost-flow-network.Rcap* (*a-current-flow state*) (*set P*)
using *P-def Pbf-props*(*3*)
by(*auto simp add*: *cost-flow-network.Rcap-def*)
ultimately have *augpath* (*a-current-flow state*) *P*
by(*auto simp add*: *cost-flow-network.augpath-def*)
moreover have *fstv* (*hd P*) = *s*
using *awalk-f same-edges Pbf-opt-path awalk-ends*[*OF Pbf-props*(*1*)] *s-neq-t*
P-def bellman-ford.opt-vs-path-def[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def*
by (*metis* (*no-types*, *lifting*))
moreover have *sndv* (*last P*) = *t*
using *awalk-f same-edges Pbf-opt-path awalk-ends*[*OF Pbf-props*(*1*)] *s-neq-t*
using *P-def bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def*
by (*metis* (*no-types*, *lifting*))
moreover have *set P* ⊆ *EEE*
proof(*rule*, *rule ccontr*, *goal-cases*)
case (*1 e*)
hence *to-edge e* ∈ *set* ((*edges-of-vwalk* ((*rev Pbf*))))
by (*metis map-in-set same-edges edges-of-vwalk-rev-swap*)
then obtain *c1 c2* **where** *c-split*:*c1*@[*prod.fst* (*to-edge e*)]@[*prod.snd* (*to-edge*

$e)]@c2 = rev\ Pbf$
    apply(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
    subgoal for *e*
      using *edges-in-vwalk-split*[*of fst e snd e rev Pbf*] *multigraph.make-pair′*
      by (*auto simp add*: *Instantiation.make-pair-def*)
    subgoal for *e*
      using *edges-in-vwalk-split*[*of snd e fst e rev Pbf*] *multigraph.make-pair′*
      by (*auto simp add*: *Instantiation.make-pair-def*)
    done
  have *c-split*:*rev c2*@[*prod.snd* (*to-edge e*)]@[*prod.fst* (*to-edge e*)]@ *rev c1* = *Pbf*
    apply(*subst sym*[*OF rev-rev-ident*[*of Pbf*]])
    apply(*subst sym*[*OF c-split*])
    by *simp*
  have *le-infty*:*prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*prod.snd* (*to-edge e*))
        (*prod.fst* (*to-edge e*))) < *PInfty*
  proof(*rule ccontr*, *goal-cases*)
    case *1*
    hence *prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*)
        (*a-current-flow state*) (*prod.snd* (*cost-flow-network.to-vertex-pair e*))
        (*prod.fst* (*cost-flow-network.to-vertex-pair e*)))
         = *PInfty* **by** *simp*
   hence *weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *Pbf* = *PInfty*
     using *bellman-ford.edge-and-Costs-none-pinfty-weight*[*OF bellman-ford*]
        c-split **by** *auto*
    thus *False*
     using *weight-le-PInfty* **by** *force*
  qed
  have *one-not-blocked*:*a-not-blocked state* (*oedge e*)
    using *less-PInfty-not-blocked 1*(*1*) *P-def Pbf-props*(*3*) **by** *blast*
  hence *oedge e* ∈ $\mathcal{E}$
    using *one*
    by(*auto elim*!: *algo.underlying-invarsE algo.inv-unbl-iff-forest-activeE*
        algo.inv-actives-in-EE algo.inv-forest-in-EE)
  thus *?case*
    using *1*(*2*) *cost-flow-network.o-edge-res* **by** *blast*
  qed
  ultimately have *resreach* (*abstract-flow-map f*) *s t*
    using *cost-flow-network.augpath-imp-resreach 1*(*3*) **by** *fast*
  thus *?thesis*
    using *one*(*1*,*2*) *s-neq-t s-prop vs-is-V* **by** *blast*
 qed
qed

lemma *bf-weight-backward-leq-res-costs*:
 assumes *set* (*map flow-network-spec.oedge qq*) ⊆ $\mathcal{E}$
    ⋀ *e*. *e* ∈ *set qq* ⟹ *a-not-blocked state* (*flow-network-spec.oedge e*)
    ⋀ *e*. *e* ∈ *set qq* ⟹ *0* < *cost-flow-network.rcap* (*a-current-flow state*) *e*
    unconstrained-awalk (*map cost-flow-network.to-vertex-pair qq*)

and *qq-len*: *length qq* $\geq$ *1*
shows *weight-backward* (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map prod.swap* (*rev* (*map cost-flow-network.to-vertex-pair qq*))))
$\leq$ *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *qq 0*
using *assms*
proof(*induction qq arbitrary*: *s rule*: *list-induct2-len-geq-1*)
case *1*
then show *?case*
using *qq-len* **by** *blast*
next
case (*2 e*)
show *?case*
using *2*
by(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*)
(*auto intro*!: *conjunct1*[*OF get-edge-and-costs-backward-makes-cheaper*[*OF refl*, *of - a-not-blocked state a-current-flow state*]]
intro: *surjective-pairing prod.collapse*
simp add: $\mathcal{E}$*-def* $\mathcal{E}$*-impl*(*1*) $\mathcal{E}$*-list-def to-list*(*1*) *make-pair-fst-snd make-pairs-are Instantiation.make-pair-def*
simp del: *cost-flow-network.*$\mathfrak{c}$*.simps*)+
next
case (*3 e d ds*)
have *help1*:
weight-backward (*a-not-blocked state*) (*a-current-flow state*)
(*butlast* (*awalk-verts s* (*map* (*prod.swap* $\circ$ *to-edge*) (*rev ds*) @ [(*snd ee*, *fst ee*)])) @ [*snd dd*]) +
prod.snd (*local.get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*snd dd*) (*fst dd*))
$\leq$ *ereal* (*local.*c *dd* + (*local.*c *ee* + *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *ds 0*))
if *assms*:($\bigwedge$*s. unconstrained-awalk* ((*fst ee*, *snd ee*) # *map to-edge ds*) $\Longrightarrow$
weight-backward (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map* (*prod.swap* $\circ$ *to-edge*) (*rev ds*) @ [(*snd ee*, *fst ee*)]))
$\leq$ *ereal* (*local.*c *ee* + *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *ds 0*))
($\bigwedge$*e. e* = *F dd* $\vee$ *e* = *F ee* $\vee$ *e* $\in$ *set ds* $\Longrightarrow$
a-not-blocked state (*oedge e*))
($\bigwedge$*e. e* = *F dd* $\vee$ *e* = *F ee* $\vee$ *e* $\in$ *set ds* $\Longrightarrow$
0 < *rcap* (*a-current-flow state*) *e*)
unconstrained-awalk ((*fst dd*, *snd dd*) # (*fst ee*, *snd ee*) # *map to-edge ds*)
dd $\in$ *local.*$\mathcal{E}$ *ee* $\in$ *local.*$\mathcal{E}$ *oedge ‘ set ds* $\subseteq$ *local.*$\mathcal{E}$
for *ee dd*
using *assms unconstrained-awalk-snd-verts-eq unconstrained-awalk-drop-hd*[*of* (*fst -*, *snd -*) (*fst -*, *snd -*)#*map to-edge -*]
by(*subst ereal-add-homo*[*of - - + -*], *subst add.commute*)
(*fastforce intro*!: *add-mono*[*OF conjunct1*[*OF get-edge-and-costs-backward-makes-cheaper*[*OF*

refl, *of - a-not-blocked state a-current-flow state*, *of F -*, *simplified*]]] *prod.collapse simp add*: *awalk-verts-append-last*′)
have *help2*: *weight-backward* (*a-not-blocked state*) (*a-current-flow state*)

(*butlast* (*awalk-verts s* (*map* (*prod.swap* ∘ *to-edge*) (*rev ds*) @ [(*fst ee*, *snd ee*)]))
@ [*snd dd*]) +
prod.snd (*local.get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*snd dd*) (*fst dd*))
≤ *ereal* (*local*.c *dd* + (*foldr* (λ*x*. (+) (𝔠 *x*)) *ds 0* − *local*.c *ee*))
if *assms*: (⋀*s*. *unconstrained-awalk* ((*snd ee*, *fst ee*) # *map to-edge ds*) ⟹
weight-backward (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map* (*prod.swap* ∘ *to-edge*) (*rev ds*) @ [(*fst ee*, *snd ee*)]))
≤ *ereal* (*foldr* (λ*x*. (+) (𝔠 *x*)) *ds 0* − *local*.c *ee*))
(⋀*e*. *e* = *F dd* ∨ *e* = *B ee* ∨ *e* ∈ *set ds* ⟹ *a-not-blocked state* (*oedge e*))
(⋀*e*. *e* = *F dd* ∨ *e* = *B ee* ∨ *e* ∈ *set ds* ⟹ *0* < *rcap* (*a-current-flow state*) *e*)
unconstrained-awalk ((*fst dd*, *snd dd*) # (*snd ee*, *fst ee*) # *map to-edge ds*)
dd ∈ *local*.ℰ *ee* ∈ *local*.ℰ
oedge ‘ *set ds* ⊆ *local*.ℰ **for** *dd ee*
using *assms*
using *unconstrained-awalk-snd-verts-eq unconstrained-awalk-drop-hd*[*of* (*fst* -, *snd* -) (*snd* -, *fst* -)#*map to-edge* -]
by(*subst ereal-add-homo*[*of* - - - -], *subst add.commute*)
(*fastforce intro*!: *add-mono*[*OF conjunct1*[*OF get-edge-and-costs-backward-makes-cheaper*[*OF*

refl, *of* - *a-not-blocked state a-current-flow state*, *of F* -, *simplified*]]] *prod.collapse simp add*: *awalk-verts-append-last*′)
have *help3*: *weight-backward* (*a-not-blocked state*) (*a-current-flow state*)
(*butlast* (*awalk-verts s* (*map* (*prod.swap* ∘ *to-edge*) (*rev ds*) @ [(*snd ee*, *fst ee*)]))
@ [*fst dd*]) +
prod.snd (*local.get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*fst dd*) (*snd dd*))
≤ *ereal* (*local*.c *ee* + *foldr* (λ*x*. (+) (𝔠 *x*)) *ds 0* − *local*.c *dd*)
if *assms*: (⋀*s*. *unconstrained-awalk* ((*fst ee*, *snd ee*) # *map to-edge ds*) ⟹
weight-backward (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map* (*prod.swap* ∘ *to-edge*) (*rev ds*) @ [(*snd ee*, *fst ee*)]))
≤ *ereal* (*local*.c *ee* + *foldr* (λ*x*. (+) (𝔠 *x*)) *ds 0*))
(⋀*e*. *e* = *B dd* ∨ *e* = *F ee* ∨ *e* ∈ *set ds* ⟹
a-not-blocked state (*oedge e*))
(⋀*e*. *e* = *B dd* ∨ *e* = *F ee* ∨ *e* ∈ *set ds* ⟹
0 < *rcap* (*a-current-flow state*) *e*)
unconstrained-awalk ((*snd dd*, *fst dd*) # (*fst ee*, *snd ee*) # *map to-edge ds*)
dd ∈ *local*.ℰ *ee* ∈ *local*.ℰ *oedge* ‘ *set ds* ⊆ *local*.ℰ **for** *ee dd*
apply(*rule forw-subst*[*of* - *ereal* ((− c *dd*) + (c *ee* + (*foldr* (λ*x*. (+) (𝔠 *x*)) *ds 0*)))], *simp*)
using *unconstrained-awalk-snd-verts-eq*[*of snd* - *fst dd fst ee snd ee*]
using *unconstrained-awalk-drop-hd*[*of* (*snd* -, *fst* -) (*fst* -, *snd* -)#*map to-edge* -]
using *awalk-verts-append-last*′[*of* - -*snd* - *fst ee*] *assms*
using *unconstrained-awalk-drop-hd*[*of* (*snd* -, *fst* -) (*fst* -, *snd* -)#*map to-edge* -]
by (*subst ereal-add-homo*[*of* - (- + -)], *subst add.commute*)
(*fastforce intro*: *prod.collapse*

intro!: *add-mono*[*OF conjunct1*[*OF get-edge-and-costs-backward-makes-cheaper*[*OF*

refl, *of - a-not-blocked state a-current-flow state*, *of B -*,
simplified]]])
have *help4*: *weight-backward* (*a-not-blocked state*) (*a-current-flow state*)
(*butlast* (*awalk-verts s* (*map* (*prod.swap* ∘ *to-edge*) (*rev ds*) @ [(*fst ee*, *snd ee*)])) @ [*fst dd*]) +
prod.snd (*local.get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*fst dd*) (*snd dd*))
≤ *ereal* (*foldr* (*λx.* (+) (𝔠 *x*)) *ds 0* − *local.*c *ee* − *local.*c *dd*) **if** *assms*:
(⋀*s. unconstrained-awalk* ((*snd ee*, *fst ee*) # *map to-edge ds*) ⟹
weight-backward (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts s* (*map* (*prod.swap* ∘ *to-edge*) (*rev ds*) @ [(*fst ee*, *snd ee*)]))
≤ *ereal* (*foldr* (*λx.* (+) (𝔠 *x*)) *ds 0* − *local.*c *ee*))
(⋀*e. e* = *B dd* ∨ *e* = *B ee* ∨ *e* ∈ *set ds* ⟹ *a-not-blocked state* (*oedge e*))
(⋀*e. e* = *B dd* ∨ *e* = *B ee* ∨ *e* ∈ *set ds* ⟹ *0* < *rcap* (*a-current-flow state*) *e*)
unconstrained-awalk ((*snd dd*, *fst dd*) # (*snd ee*, *fst ee*) # *map to-edge ds*)
dd ∈ *local.ℰ ee* ∈ *local.ℰ* *oedge* ‘ *set ds* ⊆ *local.ℰ* **for** *dd ee*
apply(*rule forw-subst*[*of - ereal* ((− c *dd*) + (− c *ee* + (*foldr* (*λx.* (+) (𝔠 *x*)) *ds 0*)))], *simp*)
using *unconstrained-awalk-snd-verts-eq*[*of snd dd fst dd snd ee fst ee*]
using *unconstrained-awalk-drop-hd*[*of* (*snd dd*, *fst dd*) (*snd ee*, *fst ee*)#*map to-edge -*]
using *awalk-verts-append-last′*[*of - -fst - snd ee*] *assms*
using *unconstrained-awalk-drop-hd*[*of* (*snd -*, *fst -*) (*snd -*, *fst -*)#*map to-edge -*]
by (*subst ereal-add-homo*[*of -* (*- + -*)], *subst add.commute*)
(*fastforce intro*: *prod.collapse*
intro!: *add-mono*[*OF conjunct1*[*OF get-edge-and-costs-backward-makes-cheaper*[*OF*

refl, *of - a-not-blocked state a-current-flow state*, *of B -*,
simplified]]])
show *?case*
using *3*
by(*induction e rule*: *cost-flow-network.to-vertex-pair.induct*,
all ‹*induction d rule*: *cost-flow-network.to-vertex-pair.induct*›)
(*auto simp add*: *make-pair-fst-snd rev-map awalk-verts-append-last*[*of - -@[-] - - - -*, *simplified*]
sym[*OF bellman-ford.costs-last*[*OF bellman-ford-backward*]] *make-pairs-are*
Instantiation.make-pair-def
intro: *help1 help2 help3 help4*)
qed

lemma *Forest-conv-erev*:
assumes *cost-flow-network.consist forst conv symmetric-digraph forst*
⋀ *e. e* ∈ *forst* ⟹ *prod.fst e* ≠ *prod.snd e*
shows *e* ∈ *conv* ‘ *forst* ⟷ *cost-flow-network.erev e* ∈ *conv* ‘ *forst*

proof(*rule cost-flow-network.consistE*[*OF assms*(*1*)], *rule symmetric-digraphE*[*OF assms*(*2*)], *rule*,
all ‹*cases e*›, *goal-cases*)
case (*1 ee*)
hence *a*:*make-pair ee* ∈ *forst* **by** (*fastforce simp add*: *make-pairs-are*)
hence *b*:*prod.swap* (*make-pair ee*) ∈ *forst*
using *1* **by** (*fastforce simp add*: *make-pairs-are*)
hence *c*:*conv* (*prod.swap* (*make-pair ee*)) = *B ee*
using *1*(*2*)[*of fst ee snd ee ee*] *assms*(*3*)[*of make-pair ee*] *a 1*(*1*,*4*,*5*)
by (*metis* (*no-types*, *lifting*) *1*(*2*) *Redge.distinct*(*1*) *Redge.inject*(*1*) *imageE*
make-pairs-are(*1*) *prod.swap-def surjective-pairing*)
then show *?case*
using *1*(*5*) *b* **by** *force*
next
case (*2 ee*)
hence *a*:*prod.swap* (*make-pair ee*) ∈ *forst* **by** (*fastforce simp add*: *make-pairs-are*
)
hence *b*:*make-pair ee* ∈ *forst*
using *2* **by** (*fastforce simp add*: *make-pairs-are*)
hence *c*:*conv* (*make-pair ee*) = *F ee*
using *2*(*2*)[*of fst ee snd ee ee*] *assms*(*3*)[*of make-pair ee*] *a 2*(*1*,*4*,*5*)
by (*metis assms*(*1*) *cost-flow-network.fstv.simps*(*2*) *cost-flow-network.sndv.simps*(*2*)
image-iff
local.algo.consist-fstv local.algo.consist-sndv make-pair-fst-snd surjective-pairing)
then show *?case*
using *2*(*5*) *b* **by** *force*
next
case (*3 ee*)
hence *c*:*conv* (*prod.swap* (*make-pair ee*)) = *B ee*
using *3*(*2*)[*of fst ee snd ee ee*] *assms*(*3*)[*of make-pair ee*]
by (*metis assms*(*1*) *cost-flow-network.erev.simps*(*1*) *cost-flow-network.fstv.simps*(*2*)
cost-flow-network.sndv.simps(*2*) *image-iff local.algo.consist-fstv local.algo.consist-sndv*
local.multigraph.make-pair''(*1*,*2*) *make-pairs-are*(*1*) *prod.swap-def surjec-
tive-pairing*)
hence *b*:*prod.swap* (*make-pair ee*) ∈ *forst*
using *3* **by** (*fastforce simp add*: *make-pairs-are*)
hence *a*:*make-pair ee* ∈ *forst***using** *3* **by** (*fastforce simp add*: *make-pairs-are*)
moreover have *conv* (*make-pair ee*) = *F ee*
using *3*(*2*) *multigraph.make-pair'* *assms*(*3*) *c calculation*
by (*fastforce simp add*: *make-pairs-are*)
then show *?case*
using *3*(*5*) *calculation* **by** *force*
next
case (*4 ee*)
hence *c*:*conv* ((*make-pair ee*)) = *F ee*
by (*metis assms*(*1*) *cost-flow-network.erev.simps*(*2*) *cost-flow-network.fstv.simps*(*1*)
cost-flow-network.sndv.simps(*1*) *image-iff local.algo.consist-fstv local.algo.consist-sndv*
make-pair-fst-snd surjective-pairing)
hence *b*:(*make-pair ee*) ∈ *forst*

using *4* **by** (*fastforce simp add*: *make-pairs-are*)
hence *a*:*prod.swap* (*make-pair ee*) ∈ *first***using** *4* **by** (*fastforce simp add*: *make-pairs-are*)
moreover have *conv* (*prod.swap* (*make-pair ee*)) = *B ee*
using *4*(*2*) *multigraph.make-pair′ assms*(*3*) *b c calculation*
by (*fastforce simp add*: *make-pairs-are*)
then show *?case*
using *4*(*5*) *calculation* **by** *force*
qed

abbreviation *get-source-target-path-b-cond* ≡ *send-flow-spec.get-source-target-path-b-cond*

lemmas *get-source-target-path-b-cond-def* = *send-flow-spec.get-source-target-path-b-cond-def*
lemmas *get-source-target-path-b-condE* = *send-flow-spec.get-source-target-path-b-condE*

lemma *get-source-target-path-b-ax*:
assumes *get-source-target-path-b-cond state s t P b* γ *f*
shows *cost-flow-network.is-min-path* (*abstract-flow-map f*) *s t P* ∧
oedge ‘ set P ⊆ *to-set* (*actives state*) ∪ $\mathcal{F}$ *state* ∧
s ∈ $\mathcal{V}$ ∧ *abstract-bal-map b s* > $\varepsilon * \gamma$
proof−
define *bf* **where** *bf* = *bellman-ford-backward* (*a-not-blocked state*) (*a-current-flow state*) *t*
define *ss-opt* **where** *ss-opt* = (*get-source-for-target-aux-aux bf*
(λ *v*. *abstract-real-map* (*bal-lookup* (*balance state*)) *v*)
(*current-*γ *state*) *vs*)
show *?thesis*
proof(*cases ss-opt*)
case *None*
hence *get-source-target-path-b state t* = *None*
by(*auto simp add*: *option-none-simp*[*of get-source-for-target-aux-aux - - - -*]
algo.abstract-not-blocked-map-def option.case-eq-if
ss-opt-def bf-def get-source-target-path-b-def)
hence *False*
using *assms* **by** (*auto elim*: *get-source-target-path-b-condE*)
thus *?thesis* **by** *simp*
next
case (*Some a*)
define *ss* **where** *ss* = *the ss-opt*
define *Pbf* **where** *Pbf* = *rev* (*search-rev-path-exec t bf ss Nil*)
define *PP* **where** *PP* = *map* (λ*e*. *prod.fst* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*)
(*prod.snd e*) (*prod.fst e*)))
(*edges-of-vwalk Pbf*)
have *ss-opt-ss*:*ss-opt* = *Some ss*
by (*simp add*: *Some ss-def*)
have *Some* (*ss*, *PP*) = *Some* (*s*, *P*)
using *assms*
by(*cases ss-opt*)
(*auto simp add*: *option-none-simp*[*of get-source-for-target-aux-aux - - - -*]

algo.abstract-not-blocked-map-def option.case-eq-if
ss-opt-def bf-def get-source-target-path-b-def ss-def
get-source-target-path-b-cond-def PP-def Pbf-def pair-to-realising-redge-backward-def)
hence *ss-is-s*: $ss = s$ **and** *PP-is-P*: $PP = P$ **by** *auto*
have *ss-props*: $ss \in$ *set local.vs*
a-balance state ss $>$ *local.*ε $*$ *current-*γ *state*
prod.snd (*the* (*connection-lookup bf ss*)) $<$ *PInfty*
using *get-source-for-target-aux-aux*(*2*)[*of bf a-balance state current-*γ *state vs*]
Some
by(*auto simp add*: *ss-def ss-opt-def*)
have *s-props*:$s \in \mathcal{V}$ *abstract-bal-map b s* $>$ *local.*ε $*$ *current-*γ *state*
resreach (*abstract-flow-map f*) *s t* $s \neq t$ **and** *gamma-0*: *current-*γ *state* $>$ *0*
using *get-source-for-target-ax*[*of b state*, *OF - refl*, *of f t s P*] *assms*
by(*auto simp add*: *get-source-target-path-b-cond-def make-pairs-are elim*: *algo.invar-gammaE*)
hence *bs-neg*:*abstract-bal-map b s* $>$ *0*
using *dual-order.strict-trans2 local.algo.*ε*-axiom*(*1*) **by** *fastforce*
have *t-props*: $t \in \mathcal{V}$ $-$ (*1* $-$ *local.*ε) $*$ *current-*γ *state* $>$ *abstract-bal-map b t*
using *get-target-axioms-red*(*1*)[*of b state current-*γ *state t*] *assms*
by(*auto simp add*: *get-source-target-path-b-cond-def*)
hence *bt-pos*: *abstract-bal-map b t* $<$ *0*
using *gamma-0* ε*-axiom t-props*(*2*)
by (*auto simp add*: *algebra-simps*)
(*smt* (*verit*, *best*) *mult-less-0-iff t-props*(*2*))
hence *a-balance-s-not-zero*:*a-balance state t* $\neq$ *0*
using *assms* **by**(*force simp add*: *get-source-target-path-b-cond-def*)
have *knowledge*: *True*
$s \in VV$ $t \in VV$ $s \neq t$
underlying-invars state
($\forall$ $e \in \mathcal{F}$ *state. 0* $<$ *abstract-flow-map f e*)
resreach (*abstract-flow-map f*) *s t*
$b =$ *balance state*
$\gamma =$ *current-*γ *state*
Some t $=$ *get-target state*
$f =$ *current-flow state*
invar-gamma state
$\neg$ ($\forall$ $v \in VV$. *abstract-bal-map b v* $=$ *0*)
$\exists$ $t \in VV$. $-$(*1* $-$ ε) $*$ γ $>$ *abstract-bal-map b t*
$\exists$ $s \in VV$. *abstract-bal-map b s* $>$ $\varepsilon * \gamma \wedge$ *resreach* (*abstract-flow-map f*) *s t*
$s = ss$ $P = PP$
using *assms t-props t-props a-balance-s-not-zero s-props*
by(*auto simp add*: *ss-is-s PP-is-P vs-is-V get-source-target-path-b-cond-def make-pairs-are*)
hence
($\forall$ $e \in$ (*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*)).
0 $<$ *a-current-flow state* (*flow-network-spec.oedge e*))
by (*auto simp add*: $\mathcal{F}$*-def*)
have *f-is*: *abstract-flow-map f* $=$ *a-current-flow state*
and *not-blocked-is*: *abstract-not-blocked-map* (*not-blocked state*) $=$ *a-not-blocked state*

using *assms* **by**(*auto simp add*: *get-source-target-path-b-cond-def*)
have *s-prop*: *abstract-bal-map b s* $>$ $\varepsilon * \gamma$ *resreach* (*abstract-flow-map f*) *s t*
using *get-source-for-target-ax*[*OF knowledge*(*8*,*9*,*11*,*10*) - *knowledge*(*12*)]
knowledge(*9*) *s-props*(*2*,*3*)
by *auto*
then obtain *pp* **where** *pp-prop*:*augpath* (*abstract-flow-map f*) *pp fstv* (*hd pp*) = *s sndv* (*last pp*) = *t set pp* $\subseteq$ *EEE*
using *cost-flow-network.resreach-imp-augpath*[*OF* , *of abstract-flow-map f s t*] **by** *auto*
obtain *ppd* **where** *ppd-props*:*augpath* (*abstract-flow-map f*) *ppd fstv* (*hd ppd*) = *s sndv* (*last ppd*) = *t set ppd* $\subseteq$ *set pp*
distinct ppd
using *pp-prop*
by (*auto intro*: *cost-flow-network.there-is-s-t-path*[*OF* - - - *refl*, *of abstract-flow-map f pp s t*])
obtain *Q* **where** *Q-min*:*cost-flow-network.is-min-path* (*abstract-flow-map f*) *s t Q*
apply(*rule cost-flow-network.there-is-min-path*[*OF* , *of abstract-flow-map f s t ppd*])
using *pp-prop ppd-props cost-flow-network.is-s-t-path-def*
by *auto*
hence *Q-prop*:*augpath* (*abstract-flow-map f*) *Q fstv* (*hd Q*) = *s sndv* (*last Q*) = *t*
set Q $\subseteq$ *EEE distinct Q*
by(*auto simp add*: *cost-flow-network.is-min-path-def*
cost-flow-network.is-s-t-path-def)
have *no-augcycle*: $\nexists$ *C*. *augcycle* (*abstract-flow-map f*) *C*
using *assms* *cost-flow-network.min-cost-flow-no-augcycle*
by(*auto simp add*: *invar-isOptflow-def elim*!: *get-source-target-path-b-condE*)
obtain *qq* **where** *qq-prop*:*augpath* (*abstract-flow-map f*) *qq*
fstv (*hd qq*) = *s*
sndv (*last qq*) = *t*
set qq
$\subseteq$ {*e* |*e*. *e* $\in$ *EEE* $\wedge$ *flow-network-spec.oedge e* $\in$ *to-set* (*actives state*)} $\cup$
(*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*))
foldr (λx. (+) ($\mathfrak{c}$ *x*)) *qq 0* $\leq$ *foldr* (λx. (+) ($\mathfrak{c}$ *x*)) *Q 0 qq* $\neq$ []
using *algo.simulate-inactives-costs*[*OF Q-prop*(*1*—*4*) *knowledge*(*5*) *refl*
f-is refl refl refl refl refl refl knowledge(*4*) - *no-augcycle*]
knowledge(*6*)
by (*auto simp add*: *algo.*$\mathcal{F}$*-redges-def*)
have *qq-len*: *length qq* $\geq$ *1 qq* $\neq$ []
using *qq-prop*(*2*,*3*,*6*) *knowledge*(*4*)
by(*all* ‹*cases qq rule*: *list-cases3*›) *auto*
have *symmetric-digraph*: *symmetric-digraph* (*Instantiation.Adj-Map-Specs2.digraph-abs* ($\mathfrak{F}$ *state*))
using *algo.from-underlying-invars'*(*19*) *knowledge*(*5*) **by** *auto*
have *forest-no-loop*: ($\bigwedge$*e*. *e* $\in$ *Instantiation.Adj-Map-Specs2.digraph-abs* ($\mathfrak{F}$ *state*) $\Longrightarrow$
prod.fst e $\neq$ *prod.snd e*)
using *algo.from-underlying-invars'*(*14*)[*OF knowledge*(*5*)]

by(*auto elim*!: *algo.validFE*
simp add: *dblton-graph-def Adj-Map-Specs2.to-graph-def UD-def*) *blast*
have *consist*: *cost-flow-network.consist* (*digraph-abs* ($\mathfrak{F}$ *state*))
(*abstract-conv-map* (*conv-to-rdg state*))
using *from-underlying-invars*′(*6*) *knowledge*(*5*) **by** *auto*
hence *e-in-pre*:$e \in$ *set qq* $\Longrightarrow$ $e \in \{e\ |e.\ e \in$ *EEE* $\wedge$ *flow-network-spec.oedge* $e \in$ *to-set* (*actives state*)$\}$
$\cup$ (*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*))
for *e*
using *qq-prop*(*4*) **by** *auto*
have *e-in*:$e \in$ *set* (*map cost-flow-network.erev* (*rev qq*)) $\Longrightarrow$ $e \in \{e\ |e.\ e \in$ *EEE* $\wedge$ *flow-network-spec.oedge* $e \in$ *to-set* (*actives state*)$\}$
$\cup$ (*abstract-conv-map* (*conv-to-rdg state*)) ‘ (*digraph-abs* ($\mathfrak{F}$ *state*))
for *e*
using *e-in-pre*[*of e*] *cost-flow-network.Residuals-project-erev-sym*[*of e*]
Forest-conv-erev[*OF consist symmetric-digraph forest-no-loop*, *simplified*]
cost-flow-network.erev-$\mathfrak{E}$ *cost-flow-network.oedge-and-reversed qq-prop*(*4*)
by *auto*
hence *e-es*:$e \in$ *set* (*map cost-flow-network.erev* (*rev qq*)) $\Longrightarrow$ *oedge* $e \in \mathcal{E}$ **for** *e*
using *algo.from-underlying-invars*′(*2*) *cost-flow-network.o-edge-res knowledge*(*5*)
by *auto*
have *e-in-pp-weight*:$e \in$ *set* (*map cost-flow-network.erev* (*rev qq*)) $\Longrightarrow$
prod.snd (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) (*fstv e*)
(*sndv e*))
$<$ *PInfty* **for** *e*
proof(*goal-cases*)
case *1*
hence *11*: *cost-flow-network.erev* $e \in$ *set qq*
using *in-set-map cost-flow-network.erve-erve-id*[*OF*] *set-rev* **by** *metis*
note *e-es*[*OF 1*]
moreover have *oedgeF*:*oedge* $e \in$ *to-set* (*actives state*) $\vee$ *oedge* $e \in \mathcal{F}$ *state*
using *e-in 1* **by** (*auto simp add*: $\mathcal{F}$*-def*)
hence *oedgeE*:*oedge* $e \in \mathcal{E}$
using *calculation* **by** *blast*
hence *not-blocked*:*a-not-blocked state* (*oedge e*)
using *oedgeF from-underlying-invars*′(*20*)[*OF knowledge*(*5*)] **by** *auto*
moreover have *flowpos*:$\exists$ *d*. (*cost-flow-network.erev e*) = *B d*$\Longrightarrow$ *a-current-flow state* (*oedge* (*cost-flow-network.erev e*)) $>$ *0*
using *cost-flow-network.augpath-rcap-pos-strict*′[*OF qq-prop*(*1*) *11*] *knowledge*(*11*)
by(*induction rule*: *flow-network-spec.oedge.cases*[*OF* , *of e*]) *auto*
ultimately show *?case*
using *11 cost-flow-network.augpath-rcap-pos-strict cost-flow-network.oedge-and-reversed cost-flow-network.vs-erev*
get-edge-and-costs-backward-makes-cheaper[*OF refl - - - prod.collapse*,
of flow-network-spec.erev e a-not-blocked state a-current-flow state] *knowledge*(*11*) *qq-prop*(*1*)
by *auto*

qed
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v. prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
by (*simp add*: *bellman-ford-backward knowledge*(*2*) *knowledge*(*3*))
have *is-a-walk*:*awalk UNIV t* (*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*))) *ss*
using *awalk-UNIV-rev*[*of ss map to-edge qq t*, *simplified rev-map*, *simplified*]
using *knowledge*(*16*) *qq-prop*(*1*) *qq-prop*(*2*) *qq-prop*(*3*)
by(*auto simp add*: *cost-flow-network.to-vertex-pair-erev-swap prepath-def aug-path-def*)
hence *vwalk-bettt*:*vwalk-bet UNIV t* (*awalk-verts t* (*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*)))) *ss*
using *awalk-imp-vwalk* **by** *force*
moreover have *weight-le-PInfty*:*weight-backward* (*a-not-blocked state*)
(*a-current-flow state*) (*awalk-verts t* (*map cost-flow-network.to-vertex-pair*
(*map cost-flow-network.erev* (*rev qq*)))) < *PInfty*
using *e-in-pp-weight is-a-walk bellman-ford-backward qq-prop*(*3*)
cost-flow-network.rev-prepath-fst-to-lst[*OF qq-len*(*2*)]
by (*intro path-flow-network-path-bf-backward*) *auto*
have *no-neg-cycle-in-bf*: $\nexists$ *c. weight-backward* (*a-not-blocked state*) (*a-current-flow state*) *c* < *0* $\wedge$ *hd c* = *last c*
using *knowledge no-neg-cycle-in-bf-backward assms*
by(*auto elim*: *get-source-target-path-b-condE*)
have *long-enough*: *2* $\leq$ *length* (*awalk-verts t* (*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*))))
using *knowledge*(*4*) *awalk-verts-non-Nil calculation knowledge*(*16*)
hd-of-vwalk-bet'[*OF calculation*] *last-of-vwalk-bet*[*OF calculation*]
by (*cases* (*awalk-verts t* (*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*)))) *rule*: *list-cases3*) *auto*
have *ss-dist-le-PInfty*:*prod.snd* (*the* (*connection-lookup bf ss*)) < *PInfty*
unfolding *bf-def bellman-ford-backward-def bellman-ford-algo-def bellman-ford-init-algo-def*
using *no-neg-cycle-in-bf knowledge*(*4*,*16*,*2*,*3*) *vs-is-V weight-le-PInfty vwalk-bettt long-enough*
by (*fastforce intro*!: *bellman-ford.bellamn-ford-path-exists-result-le-PInfty*[*OF bellman-ford-backward*])
have *s-dist-le-qq-weight*:*prod.snd* (*the* (*connection-lookup bf ss*)) $\leq$
weight-backward (*a-not-blocked state*) (*a-current-flow state*) (*awalk-verts t*
(*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*))))
using *knowledge*(*4*,*16*,*2*,*3*) *vs-is-V weight-le-PInfty is-a-walk*
bellman-ford.bellman-ford-computes-length-of-optpath[*OF bellman-ford no-neg-cycle-in-bf*, *of t s*]
bellman-ford.opt-vs-path-def[*OF bellman-ford*, *of t s*]
bellman-ford.vsp-pathI[*OF bellman-ford long-enough*, *of t s*]
bellman-ford.weight-le-PInfty-in-vs[*OF bellman-ford long-enough*, *of*]
calculation
by (*auto simp add*: *vwalk-bet-def bf-def bellman-ford-backward-def bellman-ford-algo-def*

bellman-ford-init-algo-def)
hence *s-prop*:*prod.snd* (*the* (*connection-lookup bf s*)) < *PInfty*
using *knowledge*(*16*) *ss-dist-le-PInfty* **by** *blast*
have *s-in-dom*: *s* ∈ *dom* (*connection-lookup bf*)
using *knowledge*(*2*) *vs-is-V* **by** (*auto simp add*: *bellman-ford.bellman-ford-init-dom-is*[*OF bellman-ford*]
bellman-ford.same-domain-bellman-ford[*OF bellman-ford*]
bf-def bellman-ford-backward-def bellman-ford-algo-def bellman-ford-init-algo-def)
hence *pred-of-s-not-None*: *prod.fst* (*the* (*connection-lookup bf s*)) ≠ *None*
using *s-prop knowledge*(*4*) *bellman-ford.bellman-ford-pred-non-infty-pres*[*OF bellman-ford*, *of t length vs −1*]
by(*auto simp add*: *bf-def bellman-ford-backward-def bellman-ford-algo-def bellman-ford-init-algo-def*
bellman-ford.invar-pred-non-infty-def[*OF bellman-ford*])
have *Pbf-def*: *Pbf* = (*bellford.search-rev-path t bf s*)
unfolding *Pbf-def bf-def bellman-ford-backward-def*
using *vs-is-V pred-of-s-not-None knowledge*(*2*,*3*) *ss-is-s*
apply(*subst sym*[*OF arg-cong*[*of - - rev*, *OF bellford.function-to-partial-function*, *simplified*]])
subgoal
unfolding *bellman-ford-algo-def bellman-ford-init-algo-def*
apply(*rule bf-bw.search-rev-path-dom-bellman-ford*[*OF no-neg-cycle-in-bf*])
by(*auto simp add*: *bellman-ford-backward-def bf-def*
bellman-ford-algo-def bellman-ford-init-algo-def)
by *simp*
have *weight-Pbf-snd*: *weight-backward* (*a-not-blocked state*)
(*a-current-flow state*) (*rev Pbf*) = *prod.snd* (*the* (*connection-lookup bf s*))
unfolding *Pbf-def*
using *s-prop vs-is-V pred-of-s-not-None knowledge*(*2*,*3*,*4*)
by(*fastforce simp add*: *bellman-ford-backward-def bf-def bellman-ford-algo-def bellman-ford-init-algo-def*
intro!: *bellman-ford.bellman-ford-search-rev-path-weight*[*OF bellman-ford no-neg-cycle-in-bf*, *of bf t s*])+
hence *weight-le-PInfty*: *weight-backward* (*a-not-blocked state*) (*a-current-flow state*) (*rev Pbf*) < *PInfty*
using *s-prop* **by** *auto*
have *Pbf-opt-path*: *bellman-ford.opt-vs-path vs*
(λ*u v*. *prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *t s*
(*rev* (*bellford.search-rev-path t bf s*))
using *s-prop vs-is-V pred-of-s-not-None knowledge*(*2*,*3*,*4*)
by (*auto simp add*: *bellman-ford-backward-def bf-def bellman-ford-algo-def bellman-ford-init-algo-def*
intro!: *bellman-ford.computation-of-optimum-path*[*OF bellman-ford no-neg-cycle-in-bf*])
hence *length-Pbf*:*2* ≤ *length Pbf*
by(*auto simp add*: *bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def*)

```
have Pbf-props: awalk UNIV (hd Pbf) (edges-of-vwalk  Pbf)  (last Pbf)
        weight-backward (a-not-blocked state) (a-current-flow state) (rev Pbf) =
        ereal  (foldr (λe. (+) (𝔠 e))
    (map (λe. prod.fst (get-edge-and-costs-backward (a-not-blocked state) (a-current-flow
state) (prod.snd e) (prod.fst e)))
            ( edges-of-vwalk  Pbf) ) 0)
        ⋀ e. e∈set (map (λe. prod.fst (get-edge-and-costs-backward (a-not-blocked
state) (a-current-flow state) (prod.snd e)
                    (prod.fst e)))
         ( edges-of-vwalk  Pbf) ) ⟹
    a-not-blocked state (flow-network-spec.oedge e) ∧ 0 < cost-flow-network.rcap
(a-current-flow state) e
  using edges-of-vwalk-rev-swap[of rev Pbf]
          path-bf-flow-network-path-backward[OF - length-Pbf[simplified sym[OF
length-rev[of Pbf]]]
        weight-le-PInfty refl, simplified last-rev hd-rev]
    by auto
 have same-edges:(map cost-flow-network.to-vertex-pair PP) = (edges-of-vwalk
Pbf)
  unfolding PP-def
 apply(subst (2) sym[OF List.list.map-id[of edges-of-vwalk Pbf]], subst map-map)
   using get-edge-and-costs-backward-result-props[OF prod.collapse[symmetric] -
refl]
        to-edge-get-edge-and-costs-backward
  by (fastforce intro!: map-ext)
moreover have awalk-f: awalk UNIV (fstv (hd PP)) (map cost-flow-network.to-vertex-pair
PP)
             (sndv (last PP))
  apply(rule edges-of-vwalk.elims [OF sym[OF same-edges]])
 using Pbf-props(1) same-edges length-Pbf awalk-fst-last bellman-ford.weight.simps[OF
bellman-ford]
        cost-flow-network.vs-to-vertex-pair-pres apply auto[2]
  using calculation  Pbf-props(1) same-edges
  by (auto simp add: cost-flow-network.vs-to-vertex-pair-pres awalk-intros(1)
                arc-implies-awalk[OF UNIV-I refl])
     (metis awalk-fst-last last-ConsR last-map list.simps(3) list.simps(9))
 moreover have PP ≠ []
  using  edges-of-vwalk.simps(3) length-Pbf same-edges
  by(cases Pbf rule: list-cases3) auto
 ultimately have cost-flow-network.prepath PP
 by(auto simp add:cost-flow-network.prepath-def )
 moreover have Rcap-P:0 < cost-flow-network.Rcap (a-current-flow state) (set
PP)
  using PP-def Pbf-props(3)
  by(auto simp add: cost-flow-network.Rcap-def)
 ultimately have augpath (a-current-flow state) PP
  by(auto simp add: cost-flow-network.augpath-def)
 moreover have fstv (hd PP) = s
   using awalk-f same-edges Pbf-opt-path  awalk-ends[OF Pbf-props(1)] knowl-
```

edge(*4*)
by (*force simp add*: *PP-def bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def hd-rev last-rev*)
moreover have *sndv* (*last PP*) = *t*
using *awalk-f same-edges Pbf-opt-path awalk-ends*[*OF Pbf-props*(*1*)] *knowledge*(*4*)
by (*force simp add*: *PP-def bellman-ford.opt-vs-path-def*[*OF bellman-ford*]
bellman-ford.vs-path-def[*OF bellman-ford*] *Pbf-def hd-rev last-rev*)
moreover have *oedge-of-p-allowed*:*oedge* ' (*set PP*) $\subseteq$ *to-set* (*actives state*) $\cup$ $\mathcal{F}$ *state*
proof(*rule*, *rule ccontr*, *goal-cases*)
case (*1 e*)
have *a-not-blocked state e*
using *map-in-set same-edges 1*(*1*) *PP-def Pbf-props*(*3*) *list.set-map* **by** *blast*
thus *?case*
using *from-underlying-invars'*(*20*)[*of state*, *OF knowledge*(*5*)] *1* **by** *simp*
qed
have *distinct-Pbf*: *distinct Pbf*
using *no-neg-cycle-in-bf knowledge*(*2*,*3*,*4*) *vs-is-V pred-of-s-not-None*
bellman-ford.search-rev-path-distinct[*OF bellman-ford*]
by (*fastforce simp add*: *bellman-ford-backward-def bf-def Pbf-def bellman-ford-algo-def bellman-ford-init-algo-def*)
have *distinctP*:*distinct PP*
using *distinct-edges-of-vwalk*[*OF distinct-Pbf*, *simplified sym*[*OF same-edges*]]
distinct-map **by** *auto*
have *qq-in-E*:*set* (*map flow-network-spec.oedge* (*map cost-flow-network.erev* (*rev qq*))) $\subseteq$ $\mathcal{E}$
using *e-es* **by** *auto*
hence *qq-rev-in-E*:*set* (*map flow-network-spec.oedge qq*) $\subseteq$ $\mathcal{E}$
by(*auto simp add*: *es-sym image-subset-iff cost-flow-network.oedge-and-reversed*)
have *not-blocked-qq*: $\bigwedge$ *e* . *e* $\in$ *set qq* $\Longrightarrow$ *a-not-blocked state* (*oedge e*)
using *from-underlying-invars'*(*20*)[*OF knowledge*(*5*)] *qq-prop*(*4*) **by**(*auto simp add*: $\mathcal{F}$*-def*)
have *rcap-qq*: $\bigwedge$ *e* . *e* $\in$ *set qq* $\Longrightarrow$ *cost-flow-network.rcap* (*a-current-flow state*) *e* > *0*
using *cost-flow-network.augpath-rcap-pos-strict'*[*OF qq-prop*(*1*)] *knowledge* **by** *simp*
have *awalk'*: *unconstrained-awalk* (*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*)))
unconstrained-awalk (*map cost-flow-network.to-vertex-pair qq*)
using *unconstrained-awalk-def is-a-walk qq-prop*(*1*) *cost-flow-network.augpath-def cost-flow-network.prepath-def*
by *fastforce+*
have *bf-weight-leq-res-costs*:*weight-backward* (*a-not-blocked state*) (*a-current-flow state*)
(*awalk-verts t* (*map cost-flow-network.to-vertex-pair* (*map cost-flow-network.erev* (*rev qq*))))
$\leq$ *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *qq 0*
using *qq-rev-in-E not-blocked-qq rcap-qq awalk' qq-len*

by(*fastforce intro*!: *bf-weight-backward-leq-res-costs*[*simplified cost-flow-network.rev-erev-swap* , *simplified rev-map*, *of qq - t*])
have *oedge-of-EE*: *flow-network-spec.oedge* ' *EEE* = $\mathcal{E}$
by (*meson cost-flow-network.oedge-on-*$\mathfrak{E}$)
have *flow-network-spec.oedge* ' *set PP* $\subseteq \mathcal{E}$
using *from-underlying-invars*′(*1*,*3*)[*OF knowledge*(*5*)] *oedge-of-p-allowed* **by** *blast*
hence *P-in-E*: *set PP* $\subseteq$ *EEE*
by (*meson image-subset-iff cost-flow-network.o-edge-res subsetI*)
have (*foldr* ($\lambda e.$ (+) ($\mathfrak{c}$ *e*)) *PP 0*) $\leq$ *foldr* ($\lambda x.$ (+) ($\mathfrak{c}$ *x*)) *Q 0*
using *weight-Pbf-snd s-dist-le-qq-weight Pbf-props*(*2*)[*simplified sym*[*OF PP-def*]] *qq-prop*(*5*) *bf-weight-leq-res-costs knowledge*(*16*)
by (*smt* (*verit*, *best*) *leD le-ereal-less*)
moreover have (*foldr* ($\lambda e.$ (+) ($\mathfrak{c}$ *e*)) *PP 0*) = *cost-flow-network*.$\mathfrak{C}$ *PP*
unfolding *cost-flow-network*.$\mathfrak{C}$*-def*
by(*subst distinct-sum*, *simp add*: *distinctP*, *meson add.commute*)
moreover have (*foldr* ($\lambda e.$ (+) ($\mathfrak{c}$ *e*)) *Q 0*) = *cost-flow-network*.$\mathfrak{C}$ *Q*
unfolding *cost-flow-network*.$\mathfrak{C}$*-def*
by(*subst distinct-sum*, *simp add*: *Q-prop*(*5*), *meson add.commute*)
ultimately have *P-min*: *cost-flow-network.is-min-path* (*abstract-flow-map f*) *s t PP*
using *Q-min P-in-E knowledge*(*11*) *distinctP*
by(*auto simp add*: *cost-flow-network.is-min-path-def cost-flow-network.is-s-t-path-def*)
show *?thesis*
using *PP-is-P P-min knowledge*(*9*) *oedge-of-p-allowed s-props*(*1*,*2*) **by** *force*
qed
qed

lemma *get-source-aux-nexistence*: ($\neg$ ($\exists$ *s*$\in$ *set xs*. ($1 - \varepsilon$) $*$ γ < *b s*)) = (*get-source-aux-aux b* γ *xs* = *None*)
by(*induction xs*) *auto*

lemma *get-target-aux-nexistence*: ($\neg$ ($\exists$ *s*$\in$ *set xs*. $-$ ($1 - \varepsilon$) $*$ γ > *b s*)) = (*get-target-aux-aux b* γ *xs* = *None*)
by(*induction xs*) *auto*

lemma *impl-a-None-aux*:
⟦*b* = *balance state*; γ = *current-*γ *state*; *f* = *current-flow state*;
underlying-invars state; ($\forall$ *e* $\in \mathcal{F}$ *state* . *abstract-flow-map f e* > *0*);
Some s = *get-source state*; *invar-gamma state*⟧
$\Longrightarrow$ $\neg$ ($\exists$ *t* $\in$ *VV*. *abstract-bal-map b t* < $-\ \varepsilon * \gamma$ $\wedge$ *resreach* (*abstract-flow-map f*) *s t*)
$\longleftrightarrow$ *get-source-target-path-a state s* = *None*
proof(*goal-cases*)
case *1*
note *knowledge* = *this*
define *bf* **where** *bf* = *bellman-ford-forward* (*a-not-blocked state*) (*a-current-flow state*) *s*
define *tt* **where** *tt* = *get-target-for-source-aux-aux bf*

($\lambda v.$ *a-balance state v*) (*current-γ state*)
vs
have *not-blocked-in-E*: *a-not-blocked state e* $\Longrightarrow$ $e \in \mathcal{E}$ **for** *e*
using *knowledge(4)*
by(*auto elim!*: *algo.underlying-invarsE algo.inv-unbl-iff-forest-activeE algo.inv-actives-in-EE algo.inv-forest-in-EE*)
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v. prod.snd* (*get-edge-and-costs-forward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
by (*simp add*: *bellman-ford*)
have *s-prop*: $(1 - \varepsilon) * \gamma <$ *abstract-bal-map b s* $s \in VV$
using *knowledge(6,2,1) vs-is-V get-source-aux(2)*[*of s abstract-bal-map b current-γ state vs*]
by(*auto simp add*: *get-source-def get-source-aux-def*)
hence *bs0*:*abstract-bal-map b s* > 0
using *knowledge(7,2,1)* *ε-axiom(2,4)* *algo.invar-gamma-def*
by (*smt* (*verit*, *ccfv-SIG*) *divide-less-eq-1-pos mult-nonneg-nonneg*)
have $\neg$ ($\exists$ $t \in VV$. *abstract-bal-map b t* $< - \varepsilon * \gamma \wedge$ *resreach* (*abstract-flow-map f*) *s t*) $\longleftrightarrow$
(*tt* = *None*)
proof(*rule*, *all* ‹*rule ccontr*›, *goal-cases*)
case (*1*)
then obtain *t* **where** *tt* = *Some t* **by** *auto*
note *1* = *this 1*
hence ($\exists x \in$*set vs*.
abstract-bal-map b x $< - \varepsilon *$ *current-γ state* $\wedge$
prod.snd (*the* (*connection-lookup bf x*)) $<$ *PInfty*)
using *get-target-for-source-aux-aux(1) knowledge(1)*
by(*unfold tt-def*) *blast*
then obtain *x* **where** *x-prop*:$x \in$ *set vs abstract-bal-map b x* $< - \varepsilon *$ *current-γ state prod.snd* (*the* (*connection-lookup bf x*)) $<$ *PInfty*
by *auto*
hence *bx0*:*abstract-bal-map b x* < 0
using *knowledge(7,2,1)* *ε-axiom algo.invar-gamma-def*
by (*smt* (*verit*) *mult-minus-left mult-nonneg-nonneg*)
hence *x-not-s*:$x \neq s$
using *bs0* **by** *auto*
hence *x-in-dom*:$x \in$*dom* (*connection-lookup bf*) *prod.fst* (*the* (*connection-lookup bf x*)) $\neq$ *None*
using *x-prop bellman-ford.same-domain-bellman-ford*[*OF bellman-ford*, *of length vs* -1 *s*]
bellman-ford.bellman-ford-init-dom-is[*OF bellman-ford*, *of s*]
bellman-ford.bellman-ford-pred-non-infty-pres[*OF bellman-ford*, *of s length vs* $-$ *1*]
by(*auto simp add*: *bf-def bellman-ford-forward-def bellman-ford.invar-pred-non-infty-def*[*OF bellman-ford*]
bellman-ford-init-algo-def bellman-ford-algo-def)
obtain *p* **where** *p-prop*:*weight* (*a-not-blocked state*) (*a-current-flow state*) (*p* @

[x]) =
prod.snd (the (connection-lookup bf x))
last p = the (prod.fst (the (connection-lookup bf x)))
hd p = s 1 ≤ length p set (p @ [x]) ⊆ Set.insert s (set vs)
using bellman-ford.bellman-ford-invar-pred-path-pres[OF bellman-ford, of s length vs −1]
x-in-dom
by (auto simp add: bellman-ford.invar-pred-path-def[OF bellman-ford] bf-def
bellman-ford-forward-def bellman-ford-init-algo-def bellman-ford-algo-def)
hence pw-le-PInfty: weight (a-not-blocked state) (a-current-flow state) (p @ [x]) < PInfty
using x-prop **by** auto
define pp **where** pp = (map (λe. prod.fst (get-edge-and-costs-forward (a-not-blocked state) (a-current-flow state)
(prod.fst e) (prod.snd e)))
(edges-of-vwalk (p @ [x])))
have transformed: awalk UNIV (hd (p @ [x])) (edges-of-vwalk (p @ [x])) (last (p @ [x]))
(⋀e. e∈set pp ⟹ a-not-blocked state (flow-network-spec.oedge e) ∧
0 < cost-flow-network.rcap (a-current-flow state) e)
using path-bf-flow-network-path[OF - - pw-le-PInfty refl] p-prop pp-def **by** auto
have path-hd: hd (p @ [x]) = fstv (hd pp)
by(subst pp-def , subst hd-map, ((insert p-prop(4), cases p rule: list-cases3, auto)[1]),
((insert p-prop(4), cases p rule: list-cases3, auto)[1]),
auto simp add: cost-flow-network.vs-to-vertex-pair-pres to-edge-get-edge-and-costs-forward)
have path-last: last (p @ [x]) = sndv (last pp)
apply(subst pp-def , subst last-map)
subgoal
by ((insert p-prop(4), cases p rule: list-cases3, auto)[1])
using p-prop(4)
by (auto simp add: cost-flow-network.vs-to-vertex-pair-pres to-edge-get-edge-and-costs-forward sym[OF last-v-snd-last-e])
have same-edges: (edges-of-vwalk (p @ [x])) = map cost-flow-network.to-vertex-pair pp
using to-edge-get-edge-and-costs-forward **by** (auto simp add: o-def pp-def)
have prepath:prepath pp
using transformed(1) le-simps(3) p-prop(3) p-prop(4) path-hd path-last same-edges x-not-s
by (auto simp add: cost-flow-network.prepath-def)
moreover have 0 < cost-flow-network.Rcap (abstract-flow-map f) (set pp)
using transformed(2) knowledge(3)
by(auto intro: linorder-class.Min-gr-iff simp add: cost-flow-network.Rcap-def)
ultimately have augpath (abstract-flow-map f) pp
by(simp add: cost-flow-network.augpath-def)
moreover have e ∈ set pp ⟹ e ∈ EEE **for** e
using transformed(2)[of e] not-blocked-in-E cost-flow-network.o-edge-res **by**

blast
  ultimately have *resreach* (*abstract-flow-map f*) *s x*
   using *cost-flow-network.augpath-imp-resreach path-hd p-prop*(*3*,*4*) *path-last*
   by(*cases p*) *auto*
  thus *False*
   using *1 x-prop*(*1*,*2*) *knowledge*(*2*) *vs-is-V*
   by *simp*
 next
  case *2*
  then obtain *t* **where** $t \in$*local.multigraph.*$\mathcal{V}$
     abstract-bal-map b t $<$ $-$ *local.*$\varepsilon * \gamma$ *resreach* (*abstract-flow-map f*) *s t*
   by (*auto simp add*: *make-pairs-are*)
  note *2* = *this 2*
  hence *abstract-bal-map b t* $<$ *0*
   using *knowledge*(*7*,*2*,*1*) ε*-axiom algo.invar-gamma-def*
   by (*smt* (*verit*) *mult-minus-left mult-nonneg-nonneg*)
  hence *t-not-s*:*t* $\neq$ *s*
   using *bs0* **by** *auto*
  have *f-is*: *abstract-flow-map f* = *a-current-flow state*
   by (*simp add*: *knowledge*(*3*))
  obtain *q* **where** *q-props*:*augpath* (*abstract-flow-map f*) *q fstv* (*hd q*) = *s*
     sndv (*last q*) = *t set q* $\subseteq$ *EEE*
   using *cost-flow-network.resreach-imp-augpath*[*OF* *2*(*3*)] **by** *auto*
  then obtain *qq* **where** *qq-props*:*augpath* (*abstract-flow-map f*) *qq*
   fstv (*hd qq*) = *s*
   sndv (*last qq*) = *t*
   set qq $\subseteq$ {*e* |*e. e* $\in$ *EEE* $\wedge$ *flow-network-spec.oedge e* $\in$ *to-set* (*actives state*)}
     $\cup$ *abstract-conv-map* (*conv-to-rdg state*) ' (*digraph-abs* ($\mathfrak{F}$ *state*))
   qq $\neq$ []
   using *algo.simulate-inactives*[*OF q-props*(*1*$-$*4*) *1*(*4*) *refl f-is refl refl refl refl refl refl*]
    t-not-s knowledge(*5*) **by**(*auto simp add*: $\mathcal{F}$*-redges-def*)
 have *e-in-qq-not-blocked*: *e* $\in$ *set qq* $\Longrightarrow$ *a-not-blocked state* (*flow-network-spec.oedge e*) **for** *e*
   using *qq-props*(*4*)
   by(*induction e rule*: *flow-network-spec.oedge.induct*)
    (*fastforce simp add*: *spec*[*OF algo.from-underlying-invars'*(*20*)[*OF 1*(*4*)]] *flow-network-spec.oedge.simps*(*1*)
     image-iff $\mathcal{F}$*-def dest*!: *set-mp*)+
 have *e-in-qq-rcap*: *e* $\in$ *set qq* $\Longrightarrow$ *0* $<$ *cost-flow-network.rcap* (*abstract-flow-map f*) *e* **for** *e*
   using *qq-props*(*1*) *linorder-class.Min-gr-iff*
   by (*auto simp add*: *augpath-def cost-flow-network.Rcap-def*)
  obtain *Q* **where** *Q-prop*:*fstv* (*hd Q*) = *s sndv* (*last Q*) = *t*
     distinct Q set Q $\subseteq$ *set qq augpath* (*abstract-flow-map f*) *Q*
  using *cost-flow-network.there-is-s-t-path*[*OF* , *OF qq-props*(*1*$-$*3*) *refl*] **by** *auto*
  have *e-in-qq-E*: *e* $\in$ *set Q* $\Longrightarrow$ *oedge e* $\in$ $\mathcal{E}$ **for** *e*
   using *Q-prop*(*4*) *e-in-qq-not-blocked not-blocked-in-E* **by** *blast*
  have *costsQ*: *e* $\in$ *set Q* $\Longrightarrow$

prod.snd (*get-edge-and-costs-forward* (*a-not-blocked state*) (*abstract-flow-map f*) (*fstv e*) (*sndv e*)) < *PInfty* **for** *e*
apply(*rule order.strict-trans1*)
apply(*rule conjunct1*[*OF get-edge-and-costs-forward-makes-cheaper*[*OF refl - - ,*
of e a-not-blocked state abstract-flow-map f]])
using *e-in-qq-E e-in-qq-not-blocked e-in-qq-rcap Q-prop*(*4*)
by(*auto intro*: *prod.collapse*)
have *awalk*:*awalk UNIV s* (*map cost-flow-network.to-vertex-pair Q*) *t*
using *Q-prop*(*1*) *Q-prop*(*2*) *Q-prop*(*5*) *cost-flow-network.augpath-def cost-flow-network.prepath-def* **by** *blast*
have *weight* (*a-not-blocked state*) (*abstract-flow-map f*) (*awalk-verts s* (*map cost-flow-network.to-vertex-pair Q*)) < *PInfty*
using *costsQ awalk Q-prop*(*1*) *bellman-ford knowledge*(*3*)
by (*intro path-flow-network-path-bf*[*of Q a-not-blocked state abstract-flow-map f s*]) *auto*
moreover have (*hd* (*awalk-verts s* (*map cost-flow-network.to-vertex-pair Q*))) = *s*
using *awalk* **by** *auto*
moreover have *last* (*awalk-verts s* (*map cost-flow-network.to-vertex-pair Q*)) = *t*
using *awalk* **by** *force*
ultimately have *bellman-ford.OPT vs* (λ*u v. prod.snd* (*get-edge-and-costs-forward*
(*a-not-blocked state*) (*a-current-flow state*) *u v*)) (*length vs* − *1*) *s t* < *PInfty*
using *t-not-s 1*(*3*)
by(*intro bellman-ford.weight-le-PInfty-OPTle-PInfty*[*OF bellman-ford - - refl,*
of - tl (*butlast* (*awalk-verts s* (*map cost-flow-network.to-vertex-pair Q*)))],
cases awalk-verts s (*map cost-flow-network.to-vertex-pair Q*) *rule*: *list-cases-both-sides*) *auto*
moreover have *prod.snd* (*the* (*connection-lookup bf t*)) ≤
bellman-ford.OPT vs (λ*u v. prod.snd* (*get-edge-and-costs-forward*
(*a-not-blocked state*) (*a-current-flow state*) *u v*)) (*length vs* − *1*) *s t*
using *bellman-ford.bellman-ford-shortest*[*OF bellman-ford*, *of s length vs* −*1 t*] *vs-is-V*
knowledge(*4*) *s-prop*(*2*)
by(*auto simp add*: *bf-def bellman-ford-forward-def bellman-ford-init-algo-def*
bellman-ford-algo-def)
ultimately have *prod.snd* (*the* (*connection-lookup bf t*)) < *PInfty* **by** *auto*
hence *t* ∈ *set vs abstract-bal-map b t* < − ε ∗ *current-*γ *state*
prod.snd (*the* (*connection-lookup bf t*)) < *PInfty*
using *2 knowledge*(*2*) *vs-is-V* **by** (*auto simp add*: *make-pairs-are*)
hence (*tt* ≠ *None*)
using *get-target-for-source-aux-aux*(*1*)[*of vs abstract-bal-map b*
current-γ *state bf*] *knowledge*(*1*) *tt-def*
by *blast*
thus *False*

using *2* **by** *simp*
qed
thus *?thesis*
by(*simp add*: *tt-def bf-def local.get-source-target-path-a-def*
algo.abstract-not-blocked-map-def option.case-eq-if)
qed

abbreviation *impl-a-None-cond* ≡ *send-flow-spec.impl-a-None-cond*
lemmas *impl-a-None-cond-def* = *send-flow-spec.impl-a-None-cond-def*
lemmas *impl-a-None-condE*= *send-flow-spec.impl-a-None-condE*

lemma *impl-a-None*:
impl-a-None-cond state s b γ *f* $\Longrightarrow$
(¬ (∃ *t*∈ *VV*. *abstract-bal-map b t* < − ε ∗ γ ∧ *resreach* (*abstract-flow-map f*) *s t*))
= (*get-source-target-path-a state s* = *None*)
using *impl-a-None-aux*[*OF refl refl refl*]
by (*auto elim*!: *impl-a-None-condE*)

lemma *impl-b-None-aux*:
⟦*b* = *balance state*; γ = *current-*γ *state*; *f* = *current-flow state*;
underlying-invars state; (∀ *e* ∈ $\mathcal{F}$ *state* . *abstract-flow-map f e* > *0*);
Some t = *get-target state*; *invar-gamma state*⟧
$\Longrightarrow$ ¬ (∃ *s* ∈ *VV*. *abstract-bal-map b s* > ε ∗ γ ∧ *resreach* (*abstract-flow-map f*) *s t*)
⟷ *get-source-target-path-b state t* = *None*
proof(*goal-cases*)
case *1*
note *knowledge* = *this*
define *bf* **where** *bf* = *bellman-ford-backward* (*a-not-blocked state*) (*a-current-flow state*) *t*
define *ss* **where** *ss* = *get-source-for-target-aux-aux bf*
(λ*v*. *a-balance state v*) (*current-*γ *state*)
vs
have *not-blocked-in-E*: *a-not-blocked state e* $\Longrightarrow$ *e* ∈ $\mathcal{E}$ **for** *e*
using *knowledge*(*4*)
by(*auto elim*!: *algo.underlying-invarsE algo.inv-unbl-iff-forest-activeE algo.inv-actives-in-EE algo.inv-forest-in-EE*)
have *bellman-ford*:*bellman-ford connection-empty connection-lookup connection-invar connection-delete*
es vs (λ *u v*. *prod.snd* (*get-edge-and-costs-backward* (*a-not-blocked state*) (*a-current-flow state*) *u v*)) *connection-update*
by (*simp add*: *bellman-ford-backward*)
have *t-prop*: − (*1* − ε) ∗ γ > *abstract-bal-map b t t* ∈ *VV*
using *knowledge*(*6*,*2*,*1*) *vs-is-V get-target-aux*(*2*)[*of t abstract-bal-map b current-*γ *state vs*]
by(*auto simp add*: *get-target-def get-target-aux-def*)
hence *bt0*:*abstract-bal-map b t* < *0*
using *knowledge*(*7*,*2*,*1*) ε*-axiom algo.invar-gamma-def*

```
    by (smt (verit) divide-less-eq-1-pos mult-minus-left mult-nonneg-nonneg)+
  have ¬ (∃ s ∈ VV. abstract-bal-map b s > ε ∗ γ ∧ resreach (abstract-flow-map
f) s t) ⟷
          (ss = None)
  proof(rule, all ‹rule ccontr›, goal-cases)
    case 1
    then obtain s where ss = Some s by auto
    note 1 = this 1
    hence (∃x∈set vs. abstract-bal-map b x > ε ∗ current-γ state ∧
          prod.snd (the (connection-lookup bf x)) < PInfty)
      using get-source-for-target-aux-aux(1) knowledge(1)
      by(unfold ss-def) blast
    then obtain x where x-prop:x ∈ set vs abstract-bal-map b x > ε ∗ current-γ
state prod.snd (the (connection-lookup bf x)) < PInfty
      by auto
    hence bx0:abstract-bal-map b x > 0
      using knowledge(7,2,1) ε-axiom algo.invar-gamma-def
      by (smt (verit) mult-minus-left mult-nonneg-nonneg)
    hence x-not-s:x ≠ t
      using bt0 by auto
    hence x-in-dom:x∈dom (connection-lookup bf)
                  prod.fst (the (connection-lookup bf x)) ≠ None
       using x-prop bellman-ford.same-domain-bellman-ford[OF bellman-ford, of
length vs −1 t]
           bellman-ford.bellman-ford-init-dom-is[OF bellman-ford, of t]
          bellman-ford.bellman-ford-pred-non-infty-pres[OF bellman-ford, of t length
vs − 1]
    by(auto simp add: bf-def bellman-ford-backward-def bellman-ford.invar-pred-non-infty-def[OF
bellman-ford]
                      bellman-ford-init-algo-def bellman-ford-algo-def)
     obtain p where p-prop:weight-backward (a-not-blocked state) (a-current-flow
state) (p @ [x]) =
                   prod.snd (the (connection-lookup bf x))
          last p = the (prod.fst (the (connection-lookup bf x)))
          hd p = t 1 ≤ length p set (p @ [x]) ⊆ Set.insert t (set vs)
       using bellman-ford.bellman-ford-invar-pred-path-pres[OF bellman-ford, of t
length vs −1]
           x-in-dom
       by (auto simp add: bellman-ford.invar-pred-path-def[OF bellman-ford] bf-def
bellman-ford-backward-def
                       bellman-ford-algo-def bellman-ford-init-algo-def)
    hence pw-le-PInfty: weight-backward (a-not-blocked state) (a-current-flow state)
(p @ [x]) < PInfty
      using x-prop by auto
   define pp where pp = (map (λe. prod.fst (get-edge-and-costs-backward (a-not-blocked
state) (a-current-flow state) (prod.snd e) (prod.fst e)))
              (map prod.swap (rev (edges-of-vwalk (p @ [x])))))
   have transformed: awalk UNIV (last (p @ [x])) (map prod.swap (rev (edges-of-vwalk
(p @ [x])))) (hd (p @ [x]))
```

($\bigwedge$*e. e*$\in$*set pp* $\Longrightarrow$ *a-not-blocked state* (*flow-network-spec.oedge e*) $\wedge$
0 < *cost-flow-network.rcap* (*a-current-flow state*) *e*)
using *path-bf-flow-network-path-backward*[*OF - - pw-le-PInfty refl*] *p-prop pp-def* **by** *auto*
have *non-empt*: (*rev* (*edges-of-vwalk* (*p* @ [*x*]))) $\neq$ []
by(*insert p-prop*(*4*); *cases p rule*: *list-cases3*; *auto*)
have *path-hd*: *last* (*p* @ [*x*]) = *fstv* (*hd pp*)
using *last-v-snd-last-e*[*of p*@[*x*]] *p-prop*(*4*)
by(*auto simp add*: *pp-def last-map*[*OF non-empt*] *hd-rev hd-map*[*OF non-empt*] *cost-flow-network.vs-to-vertex-pair-pres to-edge-get-edge-and-costs-backward*)
have *path-last*: *hd* (*p* @ [*x*]) = *sndv* (*last pp*)
using *hd-v-fst-hd-e*[*of p*@[*x*]] *p-prop*(*4*)
by(*auto simp add*: *pp-def last-map*[*OF non-empt*] *last-rev cost-flow-network.vs-to-vertex-pair-pres to-edge-get-edge-and-costs-backward*)
have *same-edges*: (*map prod.swap* (*rev* (*edges-of-vwalk* (*p* @ [*x*])))) = *map cost-flow-network.to-vertex-pair pp*
by(*auto simp add*: *pp-def o-def to-edge-get-edge-and-costs-backward*)
have *prepath*:*prepath pp*
using *transformed*(*1*) *le-simps*(*3*) *p-prop*(*3*) *p-prop*(*4*) *path-hd path-last x-not-s same-edges*
by(*auto simp add*: *cost-flow-network.prepath-def*)
moreover have *0* < *cost-flow-network.Rcap* (*abstract-flow-map f*) (*set pp*)
using *transformed*(*2*) *knowledge*(*3*)
by(*auto intro*: *linorder-class.Min-gr-iff simp add*: *cost-flow-network.Rcap-def*)
ultimately have *augpath* (*abstract-flow-map f*) *pp*
by(*simp add*: *cost-flow-network.augpath-def*)
moreover have *e* $\in$ *set pp* $\Longrightarrow$ *e* $\in$ *EEE* **for** *e*
using *transformed*(*2*)[*of e*] *not-blocked-in-E cost-flow-network.o-edge-res* **by** *blast*
ultimately have *resreach* (*abstract-flow-map f*) *x t*
using *cost-flow-network.augpath-imp-resreach*[*OF* , *of* (*abstract-flow-map f*) *pp*]
path-hd p-prop(*3*,*4*) *path-last*
by (*metis One-nat-def hd-append2 last-snoc le-numeral-extra*(*4*) *list.size*(*3*) *not-less-eq-eq subsetI*)
thus *False*
using *1 x-prop*(*1*,*2*) *knowledge*(*2*) *vs-is-V*
by *simp*
next
case *2*
then obtain *s* **where** *s*$\in$*multigraph.*$\mathcal{V}$ $\varepsilon * \gamma$ < *abstract-bal-map b s*
resreach (*abstract-flow-map f*) *s t*
by (*auto simp add*: *make-pairs-are*)
note *2* = *2 this*
hence *abstract-bal-map b s* > *0*
using *knowledge*(*7*,*2*,*1*) ε-*axiom algo.invar-gamma-def*
by (*smt* (*verit*) *mult-minus-left mult-nonneg-nonneg*)
hence *t-not-s*:*t* $\neq$ *s*
using *bt0* **by** *auto*

```
    have f-is: abstract-flow-map f = a-current-flow state
      by (simp add: knowledge(3))
    obtain q where q-props:augpath (abstract-flow-map f) q fstv (hd q) = s
                  sndv (last q) = t set q ⊆ EEE
      using  cost-flow-network.resreach-imp-augpath[OF  2(5)] by auto
    then obtain qq where qq-props:augpath (abstract-flow-map f) qq
       fstv (hd qq) = s
       sndv (last qq) = t
       set qq ⊆ {e |e. e ∈ EEE ∧ flow-network-spec.oedge e ∈ to-set (actives state)}
            ∪ abstract-conv-map (conv-to-rdg state) ‘ (digraph-abs (𝔉 state))
       qq ≠ []
  using algo.simulate-inactives[OF q-props(1−4) 1(4) refl f-is refl refl refl refl refl
refl]
        t-not-s knowledge(5) by(auto simp add: ℱ-redges-def)
   have e-in-qq-not-blocked: e ∈ set qq ⟹ a-not-blocked state (flow-network-spec.oedge
e) for e
      using qq-props(4)
      by(induction e rule: flow-network-spec.oedge.induct)
          (fastforce simp add: spec[OF algo.from-underlying-invars'(20)[OF 1(4)]]
flow-network-spec.oedge.simps(1)
                  image-iff ℱ-def dest!: set-mp)+
    have e-in-qq-rcap: e ∈ set qq ⟹ 0 < cost-flow-network.rcap (abstract-flow-map
f) e for e
      using qq-props(1)  linorder-class.Min-gr-iff
      by (auto simp add: augpath-def cost-flow-network.Rcap-def)
    obtain Q where Q-prop:fstv (hd Q) = s sndv (last Q) = t
                  distinct Q set Q ⊆ set qq augpath (abstract-flow-map f) Q
     using cost-flow-network.there-is-s-t-path[OF , OF qq-props(1−3) refl] by auto
    define Q′ where Q′  = map cost-flow-network.erev (rev Q)
    have Q′-prop: fstv (hd Q′) = t sndv (last Q′) = s
                  distinct Q′
      using Q-prop(1,2,3,5)
    by(auto simp add:Q′-def cost-flow-network.augpath-def cost-flow-network.prepath-def
                      hd-map[of rev Q] hd-rev last-map[of rev Q] last-rev
                     cost-flow-network.vs-erev distinct-map cost-flow-network.inj-erev
o-def)
    have e-in-qq-E: e ∈ set Q ⟹ oedge e ∈ ℰ for e
      using Q-prop(4) e-in-qq-not-blocked not-blocked-in-E by auto
    have costsQ: e ∈ set Q ⟹
      prod.snd (get-edge-and-costs-backward (a-not-blocked state) (abstract-flow-map
f) (sndv e) (fstv e)) < PInfty for e
     apply(rule order.strict-trans1)
      apply(rule conjunct1[OF get-edge-and-costs-backward-makes-cheaper[OF refl
- - ,
         of e a-not-blocked state abstract-flow-map f]])
     using  e-in-qq-E  e-in-qq-not-blocked  e-in-qq-rcap Q-prop(4)
     by(auto intro: prod.collapse)
   have costsQ′: e ∈ set Q′ ⟹
      prod.snd (get-edge-and-costs-backward (a-not-blocked state) (abstract-flow-map
```

f)
(*fstv e*) (*sndv e*)) $<$ *PInfty* **for** *e*
proof(*goal-cases*)
case *1*
have *helper*: ⟦ ($\bigwedge$*e*. *e* $\in$ *set Q* $\Longrightarrow$
prod.snd (*get-edge-and-costs-backward* (*a-not-blocked state*) (*abstract-flow-map f*) (*cost-flow-network.sndv e*)
(*cost-flow-network.fstv e*)) $\neq \infty$); *x* $\in$ *set Q* ; *e* = *cost-flow-network.erev x*;
prod.snd (*get-edge-and-costs-backward* (*a-not-blocked state*) (*abstract-flow-map f*)
(*fstv* (*cost-flow-network.erev x*)) (*sndv* (*cost-flow-network.erev x*))) = ∞⟧
$\Longrightarrow$ *False* **for** *x e*
by(*induction e rule*: *cost-flow-network.erev.induct*,
all ‹*induction x rule*: *cost-flow-network.erev.induct*›) *fastforce*+
from *1* **show** *?thesis*
using *costsQ*
by(*auto simp add*: *Q′-def intro*: *helper*)
qed
have *awalk*:*awalk UNIV t* (*map cost-flow-network.to-vertex-pair Q′*) *s*
proof–
have *helper*: ⟦ *s* = *fstv* (*hd Q*); *Q* $\neq$ []; *0* $<$ *cost-flow-network.Rcap* (*abstract-flow-map f*) (*set Q*);
t = *sndv* (*last Q*); *awalk UNIV* (*fstv* (*hd Q*)) (*map to-edge Q*) (*sndv* (*last Q*))⟧ $\Longrightarrow$
awalk UNIV (*cost-flow-network.sndv* (*last Q*)) (*map* (*prod.swap* $\circ$ *to-edge*) (*rev Q*))
(*cost-flow-network.fstv* (*hd Q*))
by(*subst sym*[*OF list.map-comp*], *subst sym*[*OF rev-map*])
(*auto simp add*: *intro*: *awalk-UNIV-rev*)
show *?thesis*
using *Q-prop*(*1*) *Q-prop*(*2*) *Q-prop*(*5*)
by (*auto simp add*: *cost-flow-network.to-vertex-pair-erev-swap cost-flow-network.augpath-def*

cost-flow-network.prepath-def Q′-def intro: *helper*)
qed
have *weight-backward* (*a-not-blocked state*) (*abstract-flow-map f*)
(*awalk-verts t* (*map cost-flow-network.to-vertex-pair Q′*)) $<$ *PInfty*
using *costsQ′ awalk Q′-prop*(*1*) *bellman-ford knowledge*(*3*)
by (*intro path-flow-network-path-bf-backward*[*of Q′ a-not-blocked state abstract-flow-map f t*]) *auto*
moreover have (*hd* (*awalk-verts t* (*map cost-flow-network.to-vertex-pair Q′*))) = *t*
using *awalk* **by** *simp*
moreover have *last* (*awalk-verts t* (*map cost-flow-network.to-vertex-pair Q′*)) = *s*
using *awalk* **by** *simp*
ultimately have *bellman-ford.OPT vs* (*λu v. prod.snd* (*get-edge-and-costs-backward*

```
                     (a-not-blocked state) (a-current-flow state) u v)) (length vs − 1)
t s < PInfty
       using t-not-s 1(3)
         by(intro bellman-ford.weight-le-PInfty-OPTle-PInfty[OF bellman-ford - -
refl,
                 of - tl (butlast (awalk-verts t (map cost-flow-network.to-vertex-pair
Q')))],
                cases awalk-verts t (map cost-flow-network.to-vertex-pair Q') rule:
list-cases-both-sides) auto
    moreover have prod.snd (the (connection-lookup bf s)) ≤
          bellman-ford.OPT vs (λu v. prod.snd (get-edge-and-costs-backward
          (a-not-blocked state) (a-current-flow state) u v))
         (length vs − 1) t s
      using bellman-ford.bellman-ford-shortest[OF bellman-ford, of t length vs −1
s] vs-is-V
          knowledge(4) t-prop(2)
     by(auto simp add: bf-def bellman-ford-backward-def bellman-ford-algo-def
          bellman-ford-init-algo-def)
    ultimately have prod.snd (the (connection-lookup bf s)) < PInfty by auto
    hence s ∈ set vs abstract-bal-map b s > ε ∗ current-γ state
         prod.snd (the (connection-lookup bf s)) < PInfty
     using 2 knowledge(2) vs-is-V by (auto simp add: make-pairs-are)
    hence (ss ≠ None)
     using get-source-for-target-aux-aux(1)[of vs current-γ state abstract-bal-map b
                                 bf] knowledge(1) ss-def
     by blast
    thus False
     using 2 by simp
  qed
  thus ?thesis
    by(simp add: ss-def bf-def local.get-source-target-path-b-def
                algo.abstract-not-blocked-map-def option.case-eq-if)
qed

abbreviation impl-b-None-cond ≡ send-flow-spec.impl-b-None-cond
lemmas impl-b-None-cond-def = send-flow-spec.impl-b-None-cond-def
lemmas impl-b-None-condE= send-flow-spec.impl-b-None-condE

lemma impl-b-None:
  impl-b-None-cond state t b γ f=⇒
   (¬ (∃ s∈VV. ε ∗ γ < abstract-bal-map b s ∧ resreach (abstract-flow-map f) s t))
=
     (get-source-target-path-b state t = None)
  using impl-b-None-aux[OF refl refl refl]
  by (auto elim!: impl-b-None-condE)

lemma test-all-vertices-zero-balance-aux:
  test-all-vertices-zero-balance-aux b xs ←→ (∀ x ∈ set xs. b x = 0)
```

by(*induction b xs rule*: *test-all-vertices-zero-balance-aux.induct*) *auto*

lemma *test-all-vertices-zero-balance*:
b = *balance state*
$\Longrightarrow$ *test-all-vertices-zero-balance state* = ($\forall$ *v*$\in$*VV*. *abstract-bal-map b v* = *0*)
using *vs-is-V*
by(*auto simp add*: *test-all-vertices-zero-balance-def test-all-vertices-zero-balance-aux*)

lemma *send-flow-axioms*:
send-flow-axioms snd u $\mathcal{E}$ c $\emptyset_N$ *vset-inv isin set-invar*
to-set lookup t-set adj-inv flow-lookup flow-invar bal-lookup bal-invar rep-comp-lookup
rep-comp-invar conv-lookup conv-invar not-blocked-lookup not-blocked-invar b
ε *fst*
get-source-target-path-a get-source-target-path-b get-source get-target
test-all-vertices-zero-balance
proof(*rule send-flow-axioms.intro*, *goal-cases*)
case (*1 state s t P b* γ *f*)
then show *?case*
using *get-source-target-path-a-ax* **by** *blast*
next
case (*2 state s t P b* γ *f*)
then show *?case*
using *get-source-target-path-a-ax* **by** *blast*
next
case (*3 state s t P b* γ *f*)
then show *?case*
using *get-source-target-path-a-ax* **by** *blast*
next
case (*4 state s t P b* γ *f*)
then show *?case*
using *get-source-target-path-b-ax* **by** *blast*
next
case (*5 state s t P b* γ *f*)
then show *?case*
using *get-source-target-path-b-ax* **by** *blast*
next
case (*6 state s t P b* γ *f*)
then show *?case*
using *get-source-target-path-b-ax* **by** *blast*
next
case (*7 s state b* γ)
then show *?case*
using *get-source-axioms* **by** *blast*
next
case (*8 state b* γ)
then show *?case*
using *get-source-axioms* **by** *blast*
next
case (*9 t state b* γ)

then show *?case*
using *get-target-axioms* **by** *blast*
next
case (*10 state b* γ)
then show *?case*
using *get-target-axioms* **by** *blast*
next
case (*11 state s b* γ *f*)
then show *?case*
using *impl-a-None* **by** (*auto simp add*: *make-pairs-are*)
next
case (*12 state t b* γ *f*)
then show *?case*
using *impl-b-None* **by** (*auto simp add*: *make-pairs-are*)
next
case (*13 state b*)
then show *?case*
using *test-all-vertices-zero-balance* **by** (*auto simp add*: *make-pairs-are*)
qed

interpretation *send-flow*:
send-flow snd create-edge u $\mathcal{E}$ c *edge-map-update* $\emptyset_N$
vset-delete vset-insert vset-inv isin filter are-all set-invar to-set lookup t-set

sel adj-inv flow-update flow-delete flow-lookup flow-invar bal-update bal-delete

bal-lookup bal-invar rep-comp-update rep-comp-delete rep-comp-lookup rep-comp-invar
conv-update conv-delete conv-lookup conv-invar not-blocked-update not-blocked-delete
not-blocked-lookup not-blocked-invar rep-comp-upd-all flow-update-all not-blocked-upd-all local.b *get-max local*.ε
$\mathcal{E}$*-impl* $\emptyset_G$ *N fst get-from-set flow-empty bal-empty rep-comp-empty conv-empty*
not-blocked-empty get-source-target-path-a get-source-target-path-b
get-source local.get-target test-all-vertices-zero-balance
by(*auto intro*!: *send-flow.intro*
simp add: *send-flow algo send-flow-axioms*)

interpretation *rep-comp-map2*:
Map **where** *empty* = *rep-comp-empty* **and** *update=rep-comp-update* **and** *lookup= rep-comp-lookup*
and *delete= rep-comp-delete* **and** *invar* = *rep-comp-invar*
using *Map-axioms* **by** *fastforce*

lemma *init-impl-variables*:
$\bigwedge$ *xs. flow-invar* (*foldr* (λ *x fl. flow-update x* (*0*::*real*) *fl*) *xs flow-empty*)
$\bigwedge$ *ys. dom* (*flow-lookup* (*foldr* (λ *x fl. flow-update x* (*0*::*real*) *fl*) *ys flow-empty*)) = *set ys*

⋀ *vs. rep-comp-invar* (*foldr* (λ *x fl. rep-comp-update x* (*x*,*1*::*nat*) *fl*) *vs* (*rep-comp-empty*))
⋀ *vs. dom* (*rep-comp-lookup* (*foldr* (λ *x fl. rep-comp-update x* (*x*,*1*::*nat*) *fl*) *vs rep-comp-empty*)) = *set vs*
⋀ *vs. not-blocked-invar* (*foldr* (λ *x fl. not-blocked-update x False fl*) *vs* ((((*not-blocked-empty*)))))
⋀ *vs. dom* (*not-blocked-lookup* (*foldr* (λ *x fl. not-blocked-update x False fl*) *vs* ((((*not-blocked-empty*))))))
= *set vs*
⋀ *vs. e* ∈ *dom* (*not-blocked-lookup* (*foldr* (λ *x fl. not-blocked-update x False fl*) *vs* ((((*not-blocked-empty*))))))
⟹ *not-blocked-lookup* (*foldr* (λ *x fl. not-blocked-update x False fl*) *vs* ((((*not-blocked-empty*))))) *e* = *Some False*
subgoal *1* **for** *xs*
by(*induction xs*)
(*auto intro*: *Map-flow.invar-empty Map-flow.invar-update*)
subgoal *2* **for** *ys*
using *1* **by**(*induction ys*)
(*auto simp add*: *Map-flow.map-update Map-flow.map-empty dom-def*)
subgoal *3* **for** *vs*
by(*induction vs*)
(*auto intro*: *invar-empty invar-update*)
subgoal *4* **for** *vs*
using *3* **by**(*induction vs*)
(*auto simp add*: *map-update map-empty dom-def*)
subgoal *5* **for** *es*
by(*induction es*)
(*auto intro*: *Map-not-blocked.invar-empty Map-not-blocked.invar-update*)
subgoal *6* **for** *vs*
using *5* **by**(*induction vs*)
(*auto simp add*: *Map-not-blocked.map-update Map-not-blocked.map-empty dom-def*)
subgoal *7* **for** *vs*
using *5* **by**(*induction vs*)
(*auto simp add*: *Map-not-blocked.map-update Map-not-blocked.map-empty dom-def*)
done

lemma *orlins-axioms*:
orlins-axioms snd $\mathcal{E}$ *flow-lookup flow-invar bal-lookup bal-invar rep-comp-lookup rep-comp-invar not-blocked-lookup not-blocked-invar* b *get-max fst init-flow*
init-bal init-rep-card init-not-blocked
proof(*rule orlins-axioms.intro*, *goal-cases*)
case *2*
then show *?case*
by (*simp add*: *init-impl-variables*(*1*) *local.init-flow-def*)
next
case *1*

then show *?case*
using *local.get-max* **by** *force*
next
case *4*
then show *?case*
using *invar-b-impl local.init-bal-def* **by** *auto*
next
case *5*
then show *?case*
by (*simp add*: *b-impl-dom local.$\mathcal{E}$-def local.init-bal-def make-pairs-are*)
next
case (*6 x*)
then show *?case*
by (*simp add*: *b-impl-dom domIff local.$\mathcal{E}$-def local.*b*-def local.init-bal-def make-pairs-are*)
next
case *7*
then show *?case*
using *init-impl-variables*(*3*) *local.init-rep-card-def* **by** *auto*
next
case *8*
then show *?case*
using *init-impl-variables*(*4*) *local.init-rep-card-def vs-is-V* **by**(*auto simp add*: *make-pairs-are*)
next
case *9*
then show *?case*
by (*simp add*: *init-impl-variables*(*5*) *local.init-not-blocked-def*)
next
case *10*
then show *?case*
using *$\mathcal{E}$-impl-invar init-impl-variables*(*6*) *local.algo.$\mathcal{E}$-impl-meaning*(*1*) *local.ees-def local.init-not-blocked-def local.to-list*(*1*) **by** *force*
next
case (*11 e*)
then show *?case*
by (*simp add*: *init-impl-variables*(*7*) *local.init-not-blocked-def*)
next
case *3*
thus *?case*
by(*simp add*: *local.$\mathcal{E}$-def init-flow-def init-impl-variables*(*2*) *ees-def*
$\mathcal{E}$-impl-invar local.to-list(*1*))
qed

interpretation *orlins*:
Orlins.orlins snd create-edge u $\mathcal{E}$ c *edge-map-update vset-empty vset-delete vset-insert*
vset-inv isin filter are-all set-invar to-set lookup t-set sel adj-inv flow-empty
flow-update flow-delete flow-lookup flow-invar bal-empty bal-update bal-delete bal-lookup

bal-invar rep-comp-empty rep-comp-update rep-comp-delete rep-comp-lookup rep-comp-invar
conv-empty conv-update conv-delete conv-lookup conv-invar not-blocked-update not-blocked-empty
not-blocked-delete not-blocked-lookup not-blocked-invar rep-comp-upd-all flow-update-all
not-blocked-upd-all b *get-max* ε *N get-from-set map-empty* $\mathcal{E}$*-impl get-path fst*
get-source-target-path-a get-source-target-path-b get-source get-target
test-all-vertices-zero-balance init-flow init-bal init-rep-card init-not-blocked
by(*auto intro*!: *orlins.intro*
simp add: *maintain-forest.maintain-forest-axioms*
send-flow.send-flow-axioms send-flow
maintain-forest.maintain-forest-spec-axioms orlins-spec-def
orlins-axioms)

definition *orlins-initial* = *orlins.initial*
definition *maintain-forest-loop-impl* = *maintain-forest.maintain-forest-impl*
definition *send-flow-loop-impl* = *send-flow-spec.send-flow-impl*
definition *orlins-loop-impl* = *orlins.orlins-impl*
definition *final-state* = *orlins-loop-impl* (*send-flow-loop-impl orlins-initial*)
definition *final-flow-impl* = *current-flow final-state*

corollary *correctness-of-implementation*:
return final-state = *success* $\Longrightarrow$ *cost-flow-network.is-Opt* b (*abstract-flow-map final-flow-impl*)
return final-state = *infeasible* $\Longrightarrow$ $\nexists$ *f*. *cost-flow-network.isbflow f* b
return final-state = *notyetterm* $\Longrightarrow$ *False*
using *orlins.initial-state-orlins-dom-and-results*[*OF refl*]
by(*auto simp add*: *final-state-def send-flow-loop-impl-def orlins-loop-impl-def*
orlins-initial-def final-flow-impl-def)

end
end

definition *no-cycle-cond fst snd* c*-impl* $\mathcal{E}$*-impl c-lookup* =
($\neg$ *has-neg-cycle* (*multigraph-spec.make-pair fst snd*)
(*function-generation*.$\mathcal{E}$ $\mathcal{E}$*-impl to-set*) (*function-generation*.c c*-impl c-lookup*))
for *fst snd*

lemma *no-cycle-condI*:
($\bigwedge$ *D*. ⟦*closed-w* ((*multigraph-spec.make-pair fst snd*) ‘ (*function-generation*.$\mathcal{E}$ $\mathcal{E}$*-impl to-set*))
(*map* (*multigraph-spec.make-pair fst snd*) *D*);
foldr (λ*e*. (+) ((*function-generation*.c c*-impl c-lookup*) *e*)) *D 0* < *0* ;
set D $\subseteq$ (*function-generation*.$\mathcal{E}$ $\mathcal{E}$*-impl to-set*)⟧ $\Longrightarrow$ *False*)
$\Longrightarrow$ *no-cycle-cond fst snd* c*-impl* $\mathcal{E}$*-impl c-lookup* **for** *fst snd*
by(*auto simp add*: *no-cycle-cond-def has-neg-cycle-def*)

term ‹*multigraph-spec.make-pair fst snd*›

thm *function-generation-proof-axioms-def*

lemma *function-generation-proof-axioms*:
⟦*set-invar* $\mathcal{E}$*-impl*; *bal-invar* b-*impl*;
dVs (*multigraph-spec.make-pair fst snd* ' *to-set* $\mathcal{E}$*-impl*) = *dom* (*bal-lookup* b-*impl*);
0 < *function-generation.N* $\mathcal{E}$*-impl to-list fst snd*⟧
⟹ *function-generation-proof-axioms bal-lookup bal-invar*
$\mathcal{E}$*-impl to-list* b-*impl set-invar to-set fst snd get-max* **for** *fst snd*
by(*intro function-generation-proof-axioms.intro*)
(*auto simp add*: *to-list* $\mathcal{E}$*-def* c-*def no-cycle-cond-def*
get-max multigraph-spec.make-pair-def selection-functions.make-pair-def)

interpretation *rep-comp-iterator*: *Map-iterator rep-comp-invar rep-comp-lookup rep-comp-upd-all*
using *Map-iterator-def rep-comp-upd-all* **by** *blast*
lemmas *rep-comp-iterator=rep-comp-iterator.Map-iterator-axioms*

interpretation *flow-iterator*: *Map-iterator flow-invar flow-lookup flow-update-all*
using *Map-iterator-def flow-update-all* **by** *blast*
lemmas *flow-iterator=flow-iterator.Map-iterator-axioms*

interpretation *not-blocked-iterator*:
Map-iterator not-blocked-invar not-blocked-lookup not-blocked-upd-all
using *Map-iterator-def not-blocked-upd-all* **by** *blast*
lemmas *not-blocked-iterator* = *not-blocked-iterator.Map-iterator-axioms*

definition *final-state fst snd create-edge* $\mathcal{E}$*-impl* c-*impl* b-*impl c-lookup* =
orlins-impl fst snd create-edge $\mathcal{E}$*-impl* c-*impl c-lookup*
(*send-flow-impl fst snd create-edge* $\mathcal{E}$*-impl* c-*impl c-lookup*
(*initial fst snd* $\mathcal{E}$*-impl* b-*impl*)) **for** *fst snd*

definition *final-flow-impl fst snd create-edge* $\mathcal{E}$*-impl* c-*impl* b-*impl c-lookup*=
(*current-flow*
(*final-state fst snd create-edge* $\mathcal{E}$*-impl* c-*impl* b-*impl c-lookup*)) **for** *fst snd*

definition *abstract-flow-map* = *algo-spec.abstract-flow-map flow-lookup*

locale *correctness-of-algo* =
fixes *fst snd*::(*'edge-type*::*linorder*) ⇒ (*'a*::*linorder*)
and c-*impl*:: *'c-impl*
and $\mathcal{E}$*-impl*::(*'edge-type*::*linorder*) *list* **and** *create-edge*
and b-*impl*:: ((*'a*::*linorder* × *real*) × *color*) *tree*
and *c-lookup*::*'c-impl* ⇒ *'edge-type* ⇒ *real option*

assumes $\mathcal{E}$*-impl-basic*: *set-invar* $\mathcal{E}$*-impl* *bal-invar* (b-*impl*)
and *Vs-is-bal-dom*: *dVs* (*multigraph-spec.make-pair fst snd* ' *to-set* $\mathcal{E}$*-impl*) = *dom* (*bal-lookup* b-*impl*)
and *at-least-2-verts*: *0* < *function-generation.N* $\mathcal{E}$*-impl to-list fst snd*

and *multigraph*: *multigraph fst snd create-edge* (*function-generation*.$\mathcal{E}$ $\mathcal{E}$-*impl* *to-set*)
begin

interpretation *function-generation-proof*:
function-generation-proof realising-edges-empty realising-edges-update realising-edges-delete realising-edges-lookup realising-edges-invar bal-empty bal-delete bal-lookup bal-invar

flow-empty flow-delete flow-lookup flow-invar not-blocked-empty not-blocked-delete

not-blocked-lookup not-blocked-invar rep-comp-empty rep-comp-delete rep-comp-lookup

rep-comp-invar $\mathcal{E}$-*impl to-list create-edge* c-*impl* b-*impl c-lookup filter are-all set-invar*
get-from-set to-set fst snd rep-comp-update conv-empty conv-delete conv-lookup conv-invar conv-update not-blocked-update flow-update bal-update rep-comp-upd-all

flow-update-all not-blocked-upd-all get-max
using $\mathcal{E}$-*impl-basic at-least-2-verts gt-zero multigraph*
rep-comp-iterator flow-iterator not-blocked-iterator
by(*auto intro*!: *function-generation-proof-axioms function-generation-proof*.*intro*

simp add: *flow-map*.*Map-axioms Map-not-blocked*.*Map-axioms Set-with-predicate* $\mathcal{E}$-*def Adj-Map-Specs2*
Map-rep-comp Map-conv bal-invar-b Vs-is-bal-dom
Map-realising-edges function-generation.*intro bal-map*.*Map-axioms*)

lemmas *function-generation-proof* = *function-generation-proof*.*function-generation-proof-axioms*
context
assumes *no-cycle*: *no-cycle-cond fst snd* c-*impl* $\mathcal{E}$-*impl c-lookup*
begin

lemma *no-cycle-cond*:*function-generation-proof*.*no-cycle-cond*
using *no-cycle*
unfolding *no-cycle-cond-def function-generation-proof*.*no-cycle-cond-def function-generation-proof*.*multigraph*.*make-pair-def selection-functions*.*make-pair-def*
by *simp*

corollary *correctness-of-implementation*:
return (*final-state fst snd create-edge* $\mathcal{E}$-*impl* c-*impl* b-*impl c-lookup*) = *success*
$\Longrightarrow$
cost-flow-spec.*is-Opt fst snd* u ($\mathcal{E}$ $\mathcal{E}$-*impl*) (c c-*impl c-lookup*) (b b-*impl*)
(*abstract-flow-map* (*final-flow-impl fst snd create-edge* $\mathcal{E}$-*impl* c-*impl* b-*impl c-lookup*))
return (*final-state fst snd create-edge* $\mathcal{E}$-*impl* c-*impl* b-*impl c-lookup*) = *infeasible*
$\Longrightarrow$
$\nexists$ *f*. *flow-network-spec*.*isbflow fst snd* ($\mathcal{E}$ $\mathcal{E}$-*impl*) u *f* (b b-*impl*)
return (*final-state fst snd create-edge* $\mathcal{E}$-*impl* c-*impl* b-*impl c-lookup*) = *notyetterm*
$\Longrightarrow$
False

using *function-generation-proof.correctness-of-implementation*[*OF no-cycle-cond*]

by(*auto simp add*: *final-state-def*
function-generation-proof.final-state-def[*OF no-cycle-cond*]
function-generation-proof.orlins-loop-impl-def[*OF no-cycle-cond*]
orlins-impl-def send-flow-impl-def N-def get-source-target-path-a-def
get-source-target-path-b-def get-source-def get-target-def get-path-def
test-all-vertices-zero-balance-def
function-generation-proof.send-flow-loop-impl-def[*OF no-cycle-cond*]
initial-def
function-generation-proof.orlins-initial-def[*OF no-cycle-cond*]
init-flow-def
init-bal-def init-rep-card-def init-not-blocked-def abstract-flow-map-def
final-flow-impl-def
function-generation-proof.final-flow-impl-def[*OF no-cycle-cond*]
$\mathcal{E}$-*def* b-*def* c-*def* u-*def*)

end
end
datatype *'a cost-wrapper* = *cost-container 'a*
end

0.2 Using Orlins Algorithms for Flows in Uncapacitated Simple Graphs

theory *Usage-Pair-Graph*
imports *Instantiation*
begin

definition $\mathcal{E}$-*impl* = [(*1*::*nat*, *2*::*nat*), (*1*,*3*), (*3*,*2*), (*2*,*4*), (*2*,*5*),
(*3*,*5*), (*4*,*6*), (*6*,*5*), (*2*,*6*)]
value $\mathcal{E}$-*impl*

definition *b-list* = [(*1*::*nat*,*128*::*real*), (*2*,*0*), (*3*,*1*), (*4*,−*33*), (*5*,−*32*), (*6*,−*64*)]

definition b-*impl* = *foldr* (λ *xy tree. update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *b-list*
Leaf
value b-*impl*

definition *c-list* = [((*1*::*nat*, *2*::*nat*), *1*::*real*),
((*1*,*3*), *4*), ((*3*,*2*), *2*), ((*2*,*4*), *3*), ((*2*,*5*), *1*),
((*3*,*5*), *5*), ((*4*,*6*), *2*), ((*6*,*5*), *1*), ((*2*,*6*), *9*)]

definition c-*impl* = *foldr* (λ *xy tree. update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *c-list*
Leaf
value c-*impl*

definition *final-state-pair* = *final-state fst snd Pair* $\mathcal{E}$-*impl* c-*impl* b-*impl flow-lookup*

value *final-state-pair*

definition *final-flow-impl-pair* = *final-flow-impl fst snd Pair* $\mathcal{E}$*-impl* c*-impl* b*-impl flow-lookup*
value *final-flow-impl-pair*

definition *final-forest* = ($\mathfrak{F}$ *final-state-pair*)

value *inorder final-flow-impl-pair*
value *map* (λ (*x*, *y*). (*x*, *inorder y*)) (*inorder final-forest*)
value *inorder* (*conv-to-rdg-impl final-state-pair*)
value *inorder* (*not-blocked-impl final-state-pair*)

lemma *no-cycle*: *closed-w* ($\mathcal{E}$ $\mathcal{E}$*-impl*) *C* $\Longrightarrow$ (*set C* $\subseteq$ $\mathcal{E}$ $\mathcal{E}$*-impl*) $\Longrightarrow$
foldr (λ*e acc. acc* + c c*-impl flow-lookup e*) *C 0* < *0* $\Longrightarrow$ *False*
proof(*goal-cases*)
case *1*
have *C-in-E*:*set C* $\subseteq$ *set* $\mathcal{E}$*-impl*
using *1* $\mathcal{E}$*-impl-def*
by (*simp add*: *subset-eq to-set-def selection-functions.*$\mathcal{E}$*-def*)
moreover have *List.filter* (λ *e.* c c*-impl flow-lookup e* > *0*) $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl*
unfolding *selection-functions.*c*-def flow-lookup-def* c*-impl-def update-def* $\mathcal{E}$*-impl-def c-list-def*
by *simp*
moreover hence *e* $\in$ *set* $\mathcal{E}$*-impl* $\Longrightarrow$ c c*-impl flow-lookup e* > *0* **for** *e*
by (*meson filter-id-conv*)
ultimately have *foldr* (λ*e acc. acc* + c c*-impl flow-lookup e*) *C 0* $\geq$ *0*
by(*induction C*)
(*auto simp add*: *order-less-le*)
then show *?case*
using *1* **by** *simp*
qed

lemma $\mathcal{E}$*-impl-basic*: *set-invar* $\mathcal{E}$*-impl* $\exists$ *e. e* $\in$ (*to-set* $\mathcal{E}$*-impl*)
finite ($\mathcal{E}$ $\mathcal{E}$*-impl*)
proof(*goal-cases*)
case *1*
then show *?case*
by(*auto simp add*: $\mathcal{E}$*-impl-def set-invar-def*)
next
case *2*
then show *?case*
by(*auto simp add*: $\mathcal{E}$*-impl-def set-invar-def to-set-def*)
next
case *3*
then show *?case*
by(*auto simp add*: $\mathcal{E}$*-impl-def set-invar-def selection-functions.*$\mathcal{E}$*-def to-set-def*)
qed

lemma *multigraph*: *multigraph fst snd Pair* ($\mathcal{E}$ $\mathcal{E}$*-impl*)
using $\mathcal{E}$*-impl-basic*(*2*,*3*)
by(*auto intro*!: *multigraph.intro simp add*: *selection-functions.*$\mathcal{E}$*-def*)

lemma *Vs-is*: *dVs* (*id* ' *to-set* $\mathcal{E}$*-impl*) = {*1*,*2*,*3*,*4*,*5*,*6*}
unfolding *to-set-def* $\mathcal{E}$*-impl-def* **by** (*auto simp add*: *dVs-def*)

lemma *Vs-is-bal-dom*: *dVs* (*id*' *to-set* $\mathcal{E}$*-impl*) = *dom* (*bal-lookup* b*-impl*)
apply(*rule trans*[*of - {1,2,3,4,5,6}*])
subgoal
unfolding *to-set-def* $\mathcal{E}$*-impl-def* **by** (*auto simp add*: *dVs-def*)
subgoal
unfolding *dom-def bal-lookup-def* b*-impl-def update-def b-list-def*
by *auto*
done

lemma *at-least-2-verts*: *1* < *function-generation.N* $\mathcal{E}$*-impl to-list* (*prod.fst o id*) (*prod.snd o id*)
apply(*subst function-generation.N-def*[*OF selection-functions.function-generation-axioms*])
by(*auto simp add*: *to-list-def* $\mathcal{E}$*-impl-def*)

lemma *no-cycle-cond*: *no-cycle-cond fst snd* c*-impl* $\mathcal{E}$*-impl flow-lookup*
by(*auto intro*!: *not-has-neg-cycleI no-cycle simp add*: $\mathcal{E}$*-def multigraph-spec.make-pair-def map-idI add.commute*[*of - -* c*-impl - -*] c*-def no-cycle-cond-def*)

lemma *correctness-of-algo*:*correctness-of-algo fst snd* $\mathcal{E}$*-impl Pair* b*-impl*
using $\mathcal{E}$*-impl-basic at-least-2-verts gt-zero multigraph Vs-is-bal-dom bal-invar-b*[*of b-list*, *simplified sym*[*OF* b*-impl-def*]]
by(*auto intro*!: *correctness-of-algo.intro simp add*: *bal-invar-b* $\mathcal{E}$*-def multigraph-spec.make-pair-def*)

corollary *correctness-of-implementation*:
return final-state-pair = *success* $\Longrightarrow$
cost-flow-spec.is-Opt fst snd u ($\mathcal{E}$ $\mathcal{E}$*-impl*) (c c*-impl flow-lookup*) (b b*-impl*) (*abstract-flow-map final-flow-impl-pair*)
return final-state-pair = *infeasible* $\Longrightarrow$
$\nexists$ *f*. *flow-network-spec.isbflow fst snd* ($\mathcal{E}$ $\mathcal{E}$*-impl*) u *f* (b b*-impl*)
return final-state-pair = *notyetterm* $\Longrightarrow$
False
using *correctness-of-algo.correctness-of-implementation*[*OF correctness-of-algo no-cycle-cond*]
by(*auto simp add*: *final-state-pair-def final-flow-impl-pair-def*)

lemma *opt-flow-found*: *cost-flow-spec.is-Opt fst snd* u ($\mathcal{E}$ $\mathcal{E}$*-impl*) (c c*-impl flow-lookup*) (b b*-impl*) (*abstract-flow-map final-flow-impl-pair*)
apply(*rule correctness-of-implementation*(*1*))
by *eval*
end

0.2.1 Flows in Multigraphs without Capacities

theory *Usage-Multigraph*
  imports *Instantiation*
begin

datatype *'a edge-type* = *an-edge* (*'a* × *'a*) | *another-edge* (*'a* × *'a*)

definition *create-edge x y* = *an-edge* (*x*,*y*)
fun *fstt* **where**
 fstt (*an-edge e*) = *fst e*|
 fstt (*another-edge e*) = *fst e*

fun *sndd* **where**
 sndd (*an-edge e*) =*snd e*|
 sndd (*another-edge e*) = *snd e*
  instantiation *edge-type*::(*linorder*) *linorder*
begin

fun *less-eq-edge-type* **where**
 less-eq-edge-type (*an-edge* (*x*, *y*)) (*another-edge* (*a*, *b*)) = *True* |
 less-eq-edge-type (*another-edge* (*x*, *y*)) (*an-edge* (*a*, *b*)) = *False* |
 less-eq-edge-type (*an-edge* (*x*, *y*)) (*an-edge* (*a*, *b*)) = ((*x*, *y*) ≤ (*a*, *b*))|
 less-eq-edge-type (*another-edge* (*x*, *y*)) (*another-edge* (*a*, *b*)) = ((*x*, *y*) ≤ (*a*, *b*))

fun *less-edge-type* **where**
 less-edge-type (*an-edge* (*x*, *y*)) (*another-edge* (*a*, *b*)) = *True* |
 less-edge-type (*another-edge* (*x*, *y*)) (*an-edge* (*a*, *b*)) = *False* |
 less-edge-type (*an-edge* (*x*, *y*)) (*an-edge* (*a*, *b*)) = ((*x*, *y*) < (*a*, *b*))|
 less-edge-type (*another-edge* (*x*, *y*)) (*another-edge* (*a*, *b*)) = ((*x*, *y*) < (*a*, *b*))
instance
proof(*intro Orderings.linorder.intro-of-class class.linorder.intro*
  class.order-axioms.intro class.order.intro class.preorder.intro
  class.linorder-axioms.intro, *goal-cases*)
  case (*1 x y*)
  then show *?case*
   apply(*all* ‹*cases x*›, *all* ‹*cases y*›)
   apply *force*
   subgoal for *a b*
   by(*all* ‹*cases a*›, *all* ‹*cases b*›)
    (*auto split*: *if-split simp add*: *less-le-not-le*)
   subgoal for *a b*
   by(*all* ‹*cases a*›, *all* ‹*cases b*›)
    (*auto split*: *if-split simp add*: *less-le-not-le*)
   by *force*
next
  case (*2 x*)
  then show *?case* **by**(*cases x*) *auto*
next
  case (*3 x y z*)

have a: $\llbracket$ *if* $ab \leq aa \wedge \neg\ aa \leq ab$ *then True else if* $ab = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq ab \wedge \neg\ ab \leq aa$ *then True else if* $aa = ab$ *then* $ba \leq bb$ *else False* ;
$x = \textit{an-edge}\ (ab, b)$; $y = \textit{an-edge}\ (aa, ba)$; $z = \textit{an-edge}\ (ab, bb)$ $\rrbracket \Longrightarrow b \leq bb$
for *aa ab ba b bb*
using *order.trans* **by** *metis*
have b: $\llbracket$ *if* $a \leq aa \wedge \neg\ aa \leq a$ *then True else if* $a = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq ab \wedge \neg\ ab \leq aa$ *then True else if* $aa = ab$ *then* $ba \leq bb$ *else False* ;
$x = \textit{an-edge}\ (a, b)$;
$y = \textit{an-edge}\ (aa, ba)$; $z = \textit{an-edge}\ (ab, bb)$; $a \neq ab$ $\rrbracket \Longrightarrow a \leq ab$
for *a aa ab b ba bb*
using *order.trans* **by** *metis*
have c: $\llbracket$ *if* $ab \leq aa \wedge \neg\ aa \leq ab$ *then True else if* $ab = aa$ *then* $b \leq ba$ *else False*
;
if $aa \leq ab \wedge \neg\ ab \leq aa$ *then True else if* $aa = ab$ *then* $ba \leq bb$ *else False* ;
$x = \textit{another-edge}\ (ab, b)$;
$y = \textit{another-edge}\ (aa, ba)$; $z = \textit{another-edge}\ (ab, bb)\rrbracket \Longrightarrow b \leq bb$
for *aa ab b ba bb*
using *order.trans* **by** *metis*
have d: $\llbracket$*if* $a \leq aa \wedge \neg\ aa \leq a$ *then True else if* $a = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq ab \wedge \neg\ ab \leq aa$ *then True else if* $aa = ab$ *then* $ba \leq bb$ *else False* ;
$x = \textit{another-edge}\ (a, b)$;
$y = \textit{another-edge}\ (aa, ba)$; $z = \textit{another-edge}\ (ab, bb)$; $a \neq ab$ $\rrbracket \Longrightarrow a \leq ab$
for *a aa ab b ba bb*
using *order.trans* **by** *metis*
from *3* **show** *?case*
by(*all* ‹*cases x*›, *all* ‹*cases y*›, *all* ‹*cases z*›)
(*auto split*: *if-split simp add*: *less-le-not-le intro*: *a b c d*)
next
case (*4 x y*)
have a: $\llbracket$*if* $a \leq aa \wedge \neg\ aa \leq a$ *then True else if* $a = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq a \wedge \neg\ a \leq aa$ *then True else if* $aa = a$ *then* $ba \leq b$ *else False* ;
$x = \textit{an-edge}\ (a, b)$; $y = \textit{an-edge}\ (aa, ba)\rrbracket \Longrightarrow a = aa$
for *a aa b ba bb*
by *presburger*
have b: $\llbracket$*if* $a \leq aa \wedge \neg\ aa \leq a$ *then True else if* $a = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq a \wedge \neg\ a \leq aa$ *then True else if* $aa = a$ *then* $ba \leq b$ *else False* ;
$x = \textit{an-edge}\ (a, b)$;$y = \textit{an-edge}\ (aa, ba)\rrbracket \Longrightarrow b = ba$
for *a aa b ba*
by (*metis order-antisym-conv*)
have c: $\llbracket$*if* $a \leq aa \wedge \neg\ aa \leq a$ *then True else if* $a = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq a \wedge \neg\ a \leq aa$ *then True else if* $aa = a$ *then* $ba \leq b$ *else False* ;
$x = \textit{another-edge}\ (a, b)$; $y = \textit{another-edge}\ (aa, ba)$ $\rrbracket \Longrightarrow b = ba$
for *a aa b ba*
by (*metis order-antisym-conv*)
have d: $\llbracket$*if* $a \leq aa \wedge \neg\ aa \leq a$ *then True else if* $a = aa$ *then* $b \leq ba$ *else False* ;
if $aa \leq a \wedge \neg\ a \leq aa$ *then True else if* $aa = a$ *then* $ba \leq b$ *else False* ;
$x = \textit{another-edge}\ (a, b)$; $y = \textit{another-edge}\ (aa, ba)\rrbracket \Longrightarrow a = aa$
for *a aa b ba*

by *presburger*
from *4* **show** *?case*
by(*all* ‹*cases x*›, *all* ‹*cases y*›)
(*auto split*: *if-split simp add*: *less-le-not-le intro*: *a b c d*)
next
case (*5 x y*)
then show *?case*
by(*all* ‹*cases x*›, *all* ‹*cases y*›)
(*force intro*: *le-cases3*)+
qed
end

definition $\mathcal{E}$*-impl* = *map an-edge* [(*1*::*nat*, *2*::*nat*), (*1*,*3*), (*3*,*2*), (*2*,*4*), (*2*,*5*), (*3*,*5*), (*4*,*6*), (*6*,*5*), (*2*,*6*)] @[*another-edge* (*1*,*2*)]
value $\mathcal{E}$*-impl*

definition *b-list* = [(*1*::*nat*,*128*::*real*), (*2*,*0*), (*3*,*1*), (*4*,−*33*), (*5*,−*32*), (*6*,−*64*)]

definition b*-impl* = *foldr* (λ *xy tree*. *update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *b-list Leaf*
value b*-impl*

definition *c-list* = [(*an-edge* (*1*::*nat*, *2*::*nat*), *1*::*real*), (*an-edge*(*1*,*3*), *4*), (*an-edge*(*3*,*2*), *2*), (*an-edge*(*2*,*4*), *3*), (*an-edge*(*2*,*5*), *1*), (*an-edge*(*3*,*5*), *5*), (*an-edge*(*4*,*6*), *2*), (*an-edge*(*6*,*5*), *1*), (*an-edge*(*2*,*6*), *9*)]@[(*another-edge* (*1*,*2*), *0.0001*)]

definition c*-impl* = *foldr* (λ *xy tree*. *update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *c-list Leaf*
value c*-impl*

term *initial-impl make-pair*

context
begin
definition *edges* = [(*0*::*nat*, *1*::*nat*), (*0*, *2*), (*2*, *3*), (*2*,*4*), (*2*,*1*), (*1*,*5*), (*5*,*8*), (*8*,*7*), (*7*,*1*),
(*7*,*2*), (*7*,*4*), (*4*,*3*), (*3*,*4*), (*3*,*3*), (*9*, *8*), (*8*, *1*), (*4*,*5*), (*5*,*10*)]

definition *G* = *a-graph edges*

value *edges*
value *G*
value *dfs-initial-state* (*1*::*nat*)
value *dfs-impl G 9* (*dfs-initial-state 0*)
value *vset-diff* (*nbs edges* (*1*::*nat*)) (*nbs edges* (*2*::*nat*))
end

definition *final-state-multi* = *final-state fstt sndd create-edge* $\mathcal{E}$*-impl* c*-impl* b*-impl flow-lookup*
value *final-state-multi*

definition *final-flow-impl-multi* = *final-flow-impl fstt sndd create-edge* $\mathcal{E}$*-impl* c*-impl* b*-impl flow-lookup*
value *final-flow-impl-multi*

definition *final-forest* = ($\mathfrak{F}$ *final-state-multi*)

value *inorder final-flow-impl-multi*
value *map* (λ (x, y). (x, *inorder* y)) (*inorder final-forest*)
value *inorder* (*conv-to-rdg final-state-multi*)
value *inorder* (*not-blocked final-state-multi*)

lemma *no-cycle*: *closed-w* (*make-pair fstt sndd* ‘ $\mathcal{E}$ $\mathcal{E}$*-impl*) (*map* (*make-pair fstt sndd*) *C*)
$\Longrightarrow$ (*set* $C \subseteq \mathcal{E}$ $\mathcal{E}$*-impl*) $\Longrightarrow$
foldr (λ*e acc. acc* + c c*-impl flow-lookup e*) *C 0* < *0* $\Longrightarrow$ *False*
proof(*goal-cases*)
case *1*
have *C-in-E*:*set* $C \subseteq$ *set* $\mathcal{E}$*-impl*
using *1* $\mathcal{E}$*-impl-def*
by (*simp add*: *subset-eq to-set-def selection-functions.*$\mathcal{E}$*-def*)
moreover have *List.filter* (λ *e*. c c*-impl flow-lookup e* > *0*) $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl*
unfolding *selection-functions.*c*-def flow-lookup-def* c*-impl-def update-def* $\mathcal{E}$*-impl-def c-list-def*
by *simp*
moreover hence $e \in$ *set* $\mathcal{E}$*-impl* $\Longrightarrow$ c c*-impl flow-lookup e* > *0* **for** *e*
by (*meson filter-id-conv*)
ultimately have *foldr* (λ*e acc. acc* + c c*-impl flow-lookup e*) *C 0* $\geq$ *0*
by(*induction C*)
(*auto simp add*: *order-less-le*)
then show *?case*
using *1* **by** *simp*
qed

lemma $\mathcal{E}$*-impl-basic*: *set-invar* $\mathcal{E}$*-impl* $\exists$ *e*. $e \in$ (*to-set* $\mathcal{E}$*-impl*)
finite ($\mathcal{E}$ $\mathcal{E}$*-impl*)
proof(*goal-cases*)
case *1*
then show *?case*
by(*auto simp add*: $\mathcal{E}$*-impl-def set-invar-def*)
next
case *2*
then show *?case*
by(*auto simp add*: $\mathcal{E}$*-impl-def set-invar-def to-set-def*)
next
case *3*

then show *?case*
by(*auto simp add*: $\mathcal{E}$*-impl-def set-invar-def selection-functions.*$\mathcal{E}$*-def to-set-def*)
qed

lemma *multigraph*: *multigraph fstt sndd create-edge* ($\mathcal{E}$ $\mathcal{E}$*-impl*)
using $\mathcal{E}$*-impl-basic*(*2*,*3*)
by(*auto intro*!: *multigraph.intro simp add*: *create-edge-def selection-functions.*$\mathcal{E}$*-def*)

lemma *Vs-is*: *dVs* (*make-pair fstt sndd ‘ to-set* $\mathcal{E}$*-impl*) = {*1*,*2*,*3*,*4*,*5*,*6*}
unfolding *to-set-def* $\mathcal{E}$*-impl-def*
by (*auto simp add*: *dVs-def make-pair-def multigraph-spec.make-pair-def*)

lemma *Vs-is-bal-dom*: *dVs* (*make-pair fstt sndd‘ to-set* $\mathcal{E}$*-impl*) = *dom* (*bal-lookup* b*-impl*)
apply(*rule trans*[*OF Vs-is*])
by(*auto simp add*: *dom-def bal-lookup-def* b*-impl-def update-def b-list-def*)

lemma *at-least-2-verts*: *1* < *function-generation.N* $\mathcal{E}$*-impl to-list fstt sndd*
apply(*subst function-generation.N-def*[*OF selection-functions.function-generation-axioms*])
by(*auto simp add*: *to-list-def* $\mathcal{E}$*-impl-def*)

lemma *no-cycle-cond*: *no-cycle-cond fstt sndd* c*-impl* $\mathcal{E}$*-impl flow-lookup*
using *no-cycle*
by(*auto intro*!: *no-cycle-condI elim*!: *has-neg-cycleE*
simp add: *no-cycle-cond-def* c*-def make-pair-def* $\mathcal{E}$*-def add.commute*[*of* - - c*-impl* - -])

lemma *correctness-of-algo*:*correctness-of-algo fstt sndd* $\mathcal{E}$*-impl create-edge* b*-impl*
using $\mathcal{E}$*-impl-basic at-least-2-verts gt-zero multigraph Vs-is-bal-dom*
by (*auto intro*!: *correctness-of-algo.intro*
simp add: b*-impl-def bal-invar-b Vs-is-bal-dom* $\mathcal{E}$*-def make-pair-def*)

corollary *correctness-of-implementation*:
return final-state-multi = *success* $\Longrightarrow$
cost-flow-spec.is-Opt fstt sndd u ($\mathcal{E}$ $\mathcal{E}$*-impl*) (c c*-impl flow-lookup*) (b b*-impl*)

(*abstract-flow-map final-flow-impl-multi*)
return final-state-multi = *infeasible* $\Longrightarrow$
$\nexists$ *f*. *flow-network-spec.isbflow fstt sndd* ($\mathcal{E}$ $\mathcal{E}$*-impl*) u *f* (b b*-impl*)
return final-state-multi = *notyetterm* $\Longrightarrow$
False
using *correctness-of-algo.correctness-of-implementation*[*OF correctness-of-algo no-cycle-cond*]
by(*auto simp add*: *final-state-multi-def final-flow-impl-multi-def*)

lemma *opt-flow-found*: *cost-flow-spec.is-Opt fstt sndd* u ($\mathcal{E}$ $\mathcal{E}$*-impl*) (c c*-impl flow-lookup*) (b b*-impl*) (*abstract-flow-map final-flow-impl-multi*)
apply(*rule correctness-of-implementation*(*1*))
by *eval*

end

0.2.2 Flows in Multigraphs with Capacities

theory *Usage-Capacitated*
  imports *Instantiation*
          Flow-Theory.Hitchcock-Reduction Flow-Theory.STFlow
begin

  instantiation *hitchcock-wrapper*::(*linorder*, *linorder*) *linorder*
begin

fun *less-eq-hitchcock-wrapper* **where**
less-eq-hitchcock-wrapper (*edge e*) (*vertex v*) = *True*|
less-eq-hitchcock-wrapper (*edge e*) (*edge d*) = ($e \geq d$)|
less-eq-hitchcock-wrapper (*vertex u*) (*vertex v*) = ($u \geq v$)|
less-eq-hitchcock-wrapper (*vertex v*) (*edge e*) = *False*

fun *less-hitchcock-wrapper* **where**
less-hitchcock-wrapper (*edge e*) (*vertex v*) = *True*|
less-hitchcock-wrapper (*edge e*) (*edge d*) = ($e > d$)|
less-hitchcock-wrapper (*vertex u*) (*vertex v*) = ($u > v$)|
less-hitchcock-wrapper (*vertex v*) (*edge e*) = *False*
instance
  apply(*intro Orderings.linorder.intro-of-class class.linorder.intro*
      class.order-axioms.intro class.order.intro class.preorder.intro
      class.linorder-axioms.intro)
  subgoal for *x y*
    by(*all* ‹*cases x*›, *all* ‹*cases y*›) *force+*
  subgoal for *x*
    by(*cases x*) *auto*
  subgoal for *x y z*
      by(*all* ‹*cases x*›, *all* ‹*cases y*›, *all* ‹*cases z*›)(*auto split*: *if-split simp add*: *less-le-not-le*)
  subgoal for *a b*
    by(*all* ‹*cases a*›, *all* ‹*cases b*›)
      (*auto split*: *if-split simp add*: *less-le-not-le*)
  subgoal for *x y*
    by(*all* ‹*cases x*›, *all* ‹*cases y*›)
      (*auto split*: *if-split simp add*: *less-le-not-le*)
  done
end

 instantiation *hitchcock-edge*::(*linorder*, *linorder*) *linorder*
begin

fun *less-eq-hitchcock-edge*::(*'a*, *'b*) *hitchcock-edge* ⇒ (*'a*, *'b*) *hitchcock-edge* ⇒ *bool*
where
less-eq-hitchcock-edge (*outedge e*) (*outedge d*) = ($e \leq d$)|

```
less-eq-hitchcock-edge (inedge e) (inedge d) = (e ≤ d)|
less-eq-hitchcock-edge (vtovedge e) (vtovedge d) = (e ≤ d)|
less-eq-hitchcock-edge (dummy x y) (dummy a b) = ((x, y) ≤ (a, b))|
less-eq-hitchcock-edge (outedge e) - = False|
less-eq-hitchcock-edge (inedge e) (outedge d) = True|
less-eq-hitchcock-edge (inedge e) - = False|
less-eq-hitchcock-edge (vtovedge e) (dummy x y) = False|
less-eq-hitchcock-edge (vtovedge e)- = True|
less-eq-hitchcock-edge (dummy x y) - = True

fun less-hitchcock-edge::('a, 'b) hitchcock-edge ⇒ ('a, 'b) hitchcock-edge ⇒ bool
where
less-hitchcock-edge (outedge e) (outedge d) = (e < d)|
less-hitchcock-edge (inedge e) (inedge d) = (e < d)|
less-hitchcock-edge (vtovedge e) (vtovedge d) = (e < d)|
less-hitchcock-edge (dummy x y) (dummy a b) = ((x, y) < (a, b))|
less-hitchcock-edge (outedge e) - = False|
less-hitchcock-edge (inedge e) (outedge d) = True|
less-hitchcock-edge (inedge e) - = False|
less-hitchcock-edge (vtovedge e) (dummy x y) = False|
less-hitchcock-edge (vtovedge e)- = True|
less-hitchcock-edge (dummy x y) - = True

instance
proof(intro Orderings.linorder.intro-of-class  class.linorder.intro
       class.order-axioms.intro class.order.intro class.preorder.intro
       class.linorder-axioms.intro, goal-cases)
  case (1 x y)
  then show ?case
    by(all ‹cases x›, all ‹cases y›) force+
next
  case (2 x)
  then show ?case
    by(cases x) auto
next
  case (3 x y z)
  then show ?case
    apply(all ‹cases x›, all ‹cases y›, all ‹cases z›)
    by(auto split: if-split simp add: less-le-not-le ) (metis order.trans)+
next
  case (4 x y)
  then show ?case
    apply(all ‹cases x›, all ‹cases y›)
    by(auto split: if-split simp add: less-le-not-le) (metis nle-le)+
next
  case (5 x y)
  then show ?case
   by(all ‹cases x›, all ‹cases y›)
     (auto split: if-split simp add: less-le-not-le)
```

qed
end

locale *with-capacity* =
fixes $fst::('edge\text{-}type::linorder) \Rightarrow ('a::linorder)$
and $snd::('edge\text{-}type::linorder) \Rightarrow ('a::linorder)$
and $create\text{-}edge::'a \Rightarrow 'a \Rightarrow 'edge\text{-}type$
and $\mathcal{E}\text{-}impl::'edge\text{-}type\ list$
and $\mathrm{c}\text{-}impl::\ 'c\text{-}type$
and $\mathrm{u}\text{-}impl::\ (('edge\text{-}type::linorder \times ereal) \times color)\ tree$
and $\mathrm{b}\text{-}impl::\ (('a::linorder \times real) \times color)\ tree$
and $c\text{-}lookup::'c\text{-}type \Rightarrow 'edge\text{-}type \Rightarrow real\ option$
begin

definition $\mathcal{E}\text{-}impl\text{-}infty = (filter\ (\lambda\ e.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e) = PInfty)\ \mathcal{E}\text{-}impl)$

definition $\mathcal{E}\text{-}impl\text{-}finite = (filter\ (\lambda\ e.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e) < PInfty)\ \mathcal{E}\text{-}impl)$

definition $\mathcal{E}1\text{-}impl = map\ inedge\ \mathcal{E}\text{-}impl\text{-}finite$
definition $\mathcal{E}2\text{-}impl = map\ outedge\ \mathcal{E}\text{-}impl\text{-}finite$
definition $\mathcal{E}3\text{-}impl = map\ (vtovedge::'edge\text{-}type \Rightarrow ('a,\ 'edge\text{-}type)\ hitchcock\text{-}edge)$
$\mathcal{E}\text{-}impl\text{-}infty$
definition $\mathcal{E}'\text{-}impl = \mathcal{E}1\text{-}impl@\mathcal{E}2\text{-}impl@\mathcal{E}3\text{-}impl$

definition $\mathrm{c}'\text{-}impl = \mathrm{c}\text{-}impl$

definition $c\text{-}lookup'\ c\ e = (case\ e\ of\ inedge\ d \Rightarrow Some\ 0\ |$
$outedge\ d \Rightarrow c\text{-}lookup\ c\ d\ |$
$vtovedge\ d \Rightarrow c\text{-}lookup\ c\ d\ |$
$dummy\ \text{-}\ \text{-} \Rightarrow None)$

definition $b\text{-}lifted = foldr\ (\lambda\ x\ tree.\ bal\text{-}update\ ((vertex::'a \Rightarrow ('a,\ 'edge\text{-}type)$
$hitchcock\text{-}wrapper)\ x)\ (the\ (bal\text{-}lookup\ \mathrm{b}\text{-}impl\ x))\ tree)$
$(vs\ fst\ snd\ \mathcal{E}\text{-}impl)\ Leaf$

definition $vertices\text{-}done = foldr\ (\lambda\ xy\ tree.\ let\ u = the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ xy)\ in$
$bal\text{-}update\ (vertex\ (fst\ xy))$
$((the\ (bal\text{-}lookup\ tree\ (vertex\ (fst\ xy))\ )) - real\text{-}of\text{-}ereal$
$u)\ tree)$
$\mathcal{E}\text{-}impl\text{-}finite\ b\text{-}lifted$

definition $\mathrm{b}'\text{-}impl = foldr\ (\lambda\ e\ tree.$
$bal\text{-}update\ ((edge::'edge\text{-}type \Rightarrow ('a,\ 'edge\text{-}type)\ hitchcock\text{-}wrapper)$
$e)$
$(real\text{-}of\text{-}ereal\ (the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e)))\ tree)\ \mathcal{E}\text{-}impl\text{-}finite$
$vertices\text{-}done$

definition $final\text{-}state\text{-}cap = final\text{-}state\ (new\text{-}fstv\text{-}gen\ fst)\ (new\text{-}sndv\text{-}gen\ fst\ snd)$

(*new-create-edge-gen*) $\mathcal{E}'$*-impl* c$'$*-impl* b$'$*-impl c-lookup$'$*

definition *final-flow-impl-cap* = *final-flow-impl* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*)
(*new-create-edge-gen*) $\mathcal{E}'$*-impl* c$'$*-impl* b$'$*-impl c-lookup$'$*

definition *final-flow-impl-original* =
(*let finite-flow* = *foldr*
(λ *e tree. flow-update e* (*the-default 0* (*flow-lookup final-flow-impl-cap* (*outedge e*))) *tree*)
$\mathcal{E}$*-impl-finite flow-empty*
in *foldr* (λ *e tree. flow-update e* (*the-default 0* (*flow-lookup final-flow-impl-cap* (*vtovedge e*))) *tree*)
$\mathcal{E}$*-impl-infty finite-flow*)

lemma *dom-final-flow-impl-original*:*dom* (*flow-lookup final-flow-impl-original*) = *set* $\mathcal{E}$*-impl*
unfolding *final-flow-impl-original-def Let-def*
apply(*subst dom-fold*)
apply(*simp add*: *flow-invar-fold flow-map.invar-update flow-map.invar-empty*)
apply(*subst dom-fold*)
by (*auto simp add*: *flow-map.map-empty dom-def* $\mathcal{E}$*-impl-finite-def* $\mathcal{E}$*-impl-infty-def*
flow-invar-fold flow-map.invar-update flow-map.invar-empty)

end

lemma *flow-lookup-fold*: *flow-invar T* $\Longrightarrow$ *flow-lookup* (*foldr* (λ*e. flow-update e* (*f e*))*AS T*) *e*
= (*if e* $\in$ *set AS then Some* (*f e*) *else flow-lookup T e*)
by(*induction AS*)
(*auto simp add*: *flow-map.map-update flow-invar-fold flow-map.invar-update*)

lemma *b$'$impl-lookup-general*:
bal-invar T $\Longrightarrow$ *bal-lookup*
(*foldr* (λ*e. bal-update* (*edge e*) (*f e*)) *ES T*)
x = (*case x of edge e* $\Rightarrow$ *if e* $\in$ *set ES then Some* (*f e*) *else bal-lookup T x*
|- $\Rightarrow$ *bal-lookup T x*)
by(*induction ES*)
(*auto split*: *hitchcock-wrapper.split simp add*: *bal-invar-fold bal-map.map-update*)

lemma *bal-lookup-fold*:
bal-invar T $\Longrightarrow$ *bal-lookup*
(*foldr* (λ*e. bal-update e* (*f e*)) *ES T*)
e = (*if e* $\in$ *set ES then Some* (*f e*) *else bal-lookup T e*)
by(*induction ES*)
(*auto split*: *hitchcock-wrapper.split simp add*: *bal-invar-fold bal-map.map-update*)

locale *with-capacity-proofs* =

with-capacity **where** *fst* = *fst*::′*edge-type*::*linorder* ⇒ ′*a*::*linorder*
and *create-edge* = *create-edge*
and $\mathcal{E}$-*impl* = $\mathcal{E}$-*impl*
and u-*impl* = u-*impl* +

cost-flow-network **where** *fst* = *fst*
and *snd* = *snd*
and *create-edge* = *create-edge*
and $\mathcal{E}$ = $\mathcal{E}$
and u = u
and c = c

for *fst create-edge* $\mathcal{E}$-*impl* u-*impl* $\mathcal{E}$ u c+
fixes b
assumes *c-domain*: $\mathcal{E}$ ⊆ *dom* (*c-lookup* c-*impl*)
and *u-domain*: *dom* (*flow-lookup* u-*impl*) = $\mathcal{E}$
and *b-domain*: *dom* (*bal-lookup* b-*impl*) = $\mathcal{V}$
and *set-invar-E*: *set-invar* $\mathcal{E}$-*impl*
and *bal-invar-b*: *bal-invar* b-*impl*
and *Es-are*: $\mathcal{E}$ = *to-set* $\mathcal{E}$-*impl*
and *cs-are*: c = *the o* (*c-lookup* c-*impl*)
and *us-are*: u = *the-default PInfty o* (*flow-lookup* u-*impl*)
and *bs-are*:b = *the-default 0 o* (*bal-lookup* b-*impl*)
begin

lemma *infty-edges-are*:*to-set* $\mathcal{E}$-*impl-infty* = *infty-edges*
using *u-domain*
unfolding $\mathcal{E}$-*impl-infty-def infty-edges-def*
by(*force simp add*: *infty-edges-def to-set-def Es-are us-are the-default-def dom-def*)

lemma *infty-edges-invar*: *set-invar* $\mathcal{E}$-*impl-infty*
using *invar-filter set-invar-E* **by** (*auto simp add*: $\mathcal{E}$-*impl-infty-def*)

lemma *finite-edges-are*:*to-set* $\mathcal{E}$-*impl-finite* = $\mathcal{E}$ − *infty-edges*
using *u-domain*
unfolding $\mathcal{E}$-*impl-finite-def infty-edges-def*
by(*force simp add*: *infty-edges-def to-set-def Es-are us-are the-default-def dom-def*)

lemma *finite-edges-invar*: *set-invar* $\mathcal{E}$-*impl-finite*
using *invar-filter set-invar-E* **by** (*auto simp add*: $\mathcal{E}$-*impl-finite-def*)

lemma *E1-impl-are*: *to-set* $\mathcal{E}$*1-impl* = *new*-$\mathcal{E}$*1-gen* $\mathcal{E}$ u
using *finite-edges-are*
by(*auto simp add*: *to-set-def* $\mathcal{E}$*1-impl-def new*-$\mathcal{E}$*1-gen-def*)

lemma *E2-impl-are*: *to-set* $\mathcal{E}$*2-impl* = *new*-$\mathcal{E}$*2-gen* $\mathcal{E}$ u
using *finite-edges-are*
by(*auto simp add*: *to-set-def* $\mathcal{E}$*2-impl-def new*-$\mathcal{E}$*2-gen-def*)

lemma *E3-impl-are*: *to-set* $\mathcal{E}$*3-impl* = *new-*$\mathcal{E}$*3-gen* $\mathcal{E}$ u
  using *infty-edges-are*
  by(*auto simp add*: *to-set-def* $\mathcal{E}$*3-impl-def new-*$\mathcal{E}$*3-gen-def*)

lemma *correctness-of-algo*:*correctness-of-algo fst snd* $\mathcal{E}$*-impl create-edge* b-*impl*
 using *Es-are b-domain E-not-empty multigraph-axioms*
  by(*auto intro*!: *correctness-of-algo.intro*
    simp add: *to-set-def to-list-def function-generation.*$\mathcal{E}$*-def*[*OF selection-functions.function-generation-axioms*]
      function-generation.N-def[*OF selection-functions.function-generation-axioms*]
        set-invar-E bal-invar-b domD make-pair-def)

lemmas *vs-and-es* = *function-generation-proof.vs-and-es*[*OF correctness-of-algo.function-generation-proof*,
    OF correctness-of-algo]

lemmas *es-def* = *function-generation.es-def*[*OF selection-functions.function-generation-axioms*]

lemma *vs-Are*:*set* (*vs fst snd* $\mathcal{E}$*-impl*) = $\mathcal{V}$
  apply(*simp add*: *vs-def vs-and-es*(*2*) *es-def dVs-def*)
  by(*auto intro*!: *cong*[*of image vertex* - $\bigcup$ - $\bigcup$ -, *OF refl*] *cong*[*of* $\bigcup$, *OF refl*]
   simp add: *Es-are to-set-def to-list-def selection-functions.make-pair-def make-pair-def*)

lemma *dom-b-listed*: *dom* (*bal-lookup b-lifted*) = *vertex* ‘ $\mathcal{V}$
  unfolding *b-lifted-def bal-lookup-def bal-update-def*
  apply(*subst dom-fold*[*simplified flow-lookup-def flow-update-def*])
  using *flow-map.invar-empty*
  by(*auto simp add*: *RBT-Set.empty-def flow-empty-def vs-Are*)

lemma *pre-b-lifted-lookup*:*bal-invar T* $\Longrightarrow$ *bal-lookup* (*foldr* (λx. *bal-update* (*vertex x*) (*the* (*bal-lookup* b-*impl x*))) *xs T*) *x* =
  (*case x of edge edge-type* $\Rightarrow$ *bal-lookup T x* | *vertex y* $\Rightarrow$ *if* $y \in$ *set xs then Some* (*the* (*bal-lookup* b-*impl y*))
   else bal-lookup T x)
  apply(*induction xs*)
  subgoal
   by(*auto split*: *hitchcock-wrapper.split*)
  apply *simp*
  apply(*subst bal-map.map-update*)
  by(*auto intro*!: *flow-invar-fold*[*simplified flow-invar-def flow-update-def*]
      flow-map.invar-update[*simplified flow-invar-def flow-update-def*]
    split: *hitchcock-wrapper.split*
    simp add: *bal-lookup-def bal-invar-def bal-update-def*)

lemma *b-lifted-lookup*: *bal-lookup b-lifted x* =
    (*case x of vertex y* $\Rightarrow$ *if* $y \in \mathcal{V}$ *then Some* (*the* (*bal-lookup* b-*impl y*))
      else None |
    - $\Rightarrow$ *None*)
  unfolding *b-lifted-def*

apply(*subst pre-b-lifted-lookup*)
using *bal-map.invar-empty*[*simplified RBT-Set.empty-def bal-empty-def*] *vs-Are*
by(*auto split*: *hitchcock-wrapper.split*
simp add: *cong*[*OF bal-map.map-empty*[*simplified RBT-Set.empty-def bal-empty-def*] *refl*])

lemma *vertices-done-general-lookup*:
$x \in$ *dom* (*bal-lookup bs*) $\Longrightarrow$ *bal-invar bs* $\Longrightarrow$ *distinct ES* $\Longrightarrow$ *bal-lookup* (*foldr*
(λ*xy tree*.
let u = *the* (*flow-lookup* u-*impl xy*)
in bal-update (*vertex* (*fst xy*))
(*the* (*bal-lookup tree* (*vertex* (*fst xy*))) − *real-of-ereal u*) *tree*)
ES bs) *x* =
(*case x of vertex u* ⇒ *Some* (
the (*bal-lookup bs* (*vertex u*))
− *sum* (λ *e*. *real-of-ereal* (*the* (*flow-lookup* u-*impl e*))) {*e* | *e*. *e* ∈ *set ES* ∧ *u* = *fst e*})
| - ⇒ *bal-lookup bs x*)
proof(*induction ES*)
case *Nil*
then show *?case*
by(*auto split*: *hitchcock-wrapper.split*)
next
case (*Cons a ES*)
then show *?case*
apply *simp*
apply(*subst bal-map.map-update*)
subgoal
by(*auto intro*: *bal-invar-fold*)
by(*auto split*: *hitchcock-wrapper.split*)
(((*subst sym*[*OF minus-distr*], *subst add.commute*, *subst sym*[*OF sum.insert*]);
(*force intro*!: *cong*[*of uminus*, *OF refl*] *cong*[*of sum* -, *OF refl*] *simp add*:)+),
metis)
qed

lemma *bal-invar-b-lifted*: *bal-invar b-lifted*
using *bal-map.invar-empty*
by(*auto intro*: *bal-invar-fold simp add*:*b-lifted-def RBT-Set.empty-def bal-empty-def*)

lemma *flow-network2*: *flow-network fst snd create-edge*
(*the-default PInfty* ∘ *flow-lookup* u-*impl*) $\mathcal{E}$
using *flow-network-axioms us-are* **by** *auto*

lemma *bal-lookup-vertices-done*:$x \in \mathcal{V} \Longrightarrow$ *bal-lookup vertices-done* (*vertex x*) =
Some (b *x* − *sum* (*real-of-ereal o* u)
((*delta-plus x*) − (*delta-plus-infty x*)))
unfolding *vertices-done-def*
apply(*subst vertices-done-general-lookup*)
using *dom-b-listed bal-invar-b-lifted finite-edges-invar*

apply(*auto simp add*: *set-invar-def*)[*3*]
using *u-domain b-domain*
by(*simp add*: *b-lifted-lookup bs-are us-are*, *unfold delta-plus-def flow-network-spec.delta-plus-infty-def the-default-def*)
(*cases bal-lookup* b-*impl x*, *blast*, *simp*,*intro sum-cong-extensive*,
(*force simp add*: *Es-are* $\mathcal{E}$*-impl-finite-def to-set-def delta-plus-def*
dom-def the-default-def)+)

lemma *dom-vertices-done*:*dom* (*bal-lookup vertices-done*) = *vertex* ' $\mathcal{V}$
using *fst-E-V*
by (*auto simp add*: *vertices-done-def bal-dom-fold bal-invar-b-lifted dom-b-listed* $\mathcal{E}$*-impl-finite-def Es-are to-set-def*)

lemma *bal-invar-vertices-done*: *bal-invar vertices-done*
by(*auto intro*: *bal-invar-fold simp add*: *bal-invar-b-lifted vertices-done-def*)

lemma *b′-impl-dom*:*dom* (*bal-lookup* b′*-impl*) = *vertex* ' $\mathcal{V}$ ∪ *edge* ' ($\mathcal{E}$ − *infty-edges*)
unfolding b′*-impl-def*
apply(*subst bal-dom-fold*, *simp add*: *bal-invar-vertices-done*)
using *u-domain*
unfolding $\mathcal{E}$*-impl-finite-def infty-edges-def*
by(*subst dom-vertices-done*)(*force simp add*: *us-are Es-are to-set-def dom-def the-default-def*)

lemma *bal-invar-b′-impl*: *bal-invar* b′*-impl*
by (*simp add*: b′*-impl-def bal-invar-fold bal-invar-vertices-done*)

lemma *b′-impl-lookup*:*x* ∈ *vertex* ' $\mathcal{V}$ ∪ *edge* ' ($\mathcal{E}$ − *infty-edges*) ⟹
the (*bal-lookup* b′*-impl x*) = *new*-b-*gen fst* $\mathcal{E}$ u b *x*
using *finite-edges-are u-domain*
by(*auto split*: *hitchcock-wrapper.split*
simp add: *to-set-def us-are bal-lookup-vertices-done bal-invar-vertices-done*
b′impl-lookup-general b′*-impl-def new*-b-*gen-def dom-def the-default-def*)

lemma *old-f-gen-final-flow-impl-original-cong*:*e* ∈ $\mathcal{E}$ ⟹
old-f-gen $\mathcal{E}$ u (*abstract-flow-map final-flow-impl-cap*) *e* = *abstract-flow-map final-flow-impl-original e*
unfolding *old-f-gen-def final-flow-impl-original-def Let-def abstract-flow-map-def the-default-def abstract-real-map-def*
apply(*subst flow-lookup-fold*, *simp add*: *flow-invar-fold flow-map.invar-empty flow-map.invar-update*)+
by (*auto simp add*:*sym*[*OF infty-edges-are*, *simplified to-set-def*] *flow-map.map-empty finite-edges-are*[*simplified sym*[*OF infty-edges-are*] *to-set-def*])

lemma *set-invar-E′*:*set-invar* $\mathcal{E}$′*-impl*
using *set-invar-E*
by (*auto intro*!: *distinct-map-filter distinct-filter simp add*: *distinct-map inj-on-def*

set-invar-def
$\mathcal{E}'$-impl-def $\mathcal{E}$1-impl-def $\mathcal{E}$2-impl-def $\mathcal{E}$3-impl-def $\mathcal{E}$-impl-finite-def $\mathcal{E}$-impl-infty-def)

lemma *V-new-graph*:*dVs* (*multigraph-spec.make-pair* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*) ‘ *to-set* $\mathcal{E}'$*-impl*)
$= vertex$ ‘ $\mathcal{V} \cup edge$ ‘ $(\mathcal{E} - \mathcal{E}_\infty)$
proof−
have *1*:$x \notin edge$ ‘
(*set* $\mathcal{E}$*-impl* − {$e \in$ *set* $\mathcal{E}$*-impl*. *the-default* ∞ (*flow-lookup* u-*impl e*) = ∞}) $\Longrightarrow$
$x \in dVs$ ((λx. (*edge x*, *vertex* (*fst x*))) ‘
{$x \in$ *set* $\mathcal{E}$*-impl*. *the* (*flow-lookup* u-*impl x*) $\neq \infty$}) $\Longrightarrow$
$x \in$ *vertex* ‘ *dVs* ((λx. (*fst x*, *snd x*)) ‘ *set* $\mathcal{E}$*-impl*) **for** *x*
proof(*goal-cases*)
case (*1*)
then obtain *e* **where** $x = edge\ e \lor x = vertex\ (fst\ e)$ $e \in$ *set* $\mathcal{E}$*-impl*
the (*flow-lookup* u-*impl e*) $\neq \infty$ **by**(*auto simp add*: *dVs-def*)
moreover hence $x \neq edge\ e$ **using** *u-domain 1*(*1*)
by(*force simp add*: *dom-def the-default-def Es-are case-simp*(*1*) *to-set-def*)
ultimately show *?case*
unfolding *dVs-def*
by(*fastforce intro*!: *imageI intro*: *exI*[*of* - {*fst e*, *snd e*}] *simp add*: *dVs-def*)
qed
moreover have *2*:$x \notin edge$ ‘
(*set* $\mathcal{E}$*-impl* − {$e \in$ *set* $\mathcal{E}$*-impl*. *the-default* ∞ (*flow-lookup* u-*impl e*) = ∞}) $\Longrightarrow$
$x \in dVs$ ((λx. (*edge x*, *vertex* (*snd x*))) ‘
{$x \in$ *set* $\mathcal{E}$*-impl*. *the* (*flow-lookup* u-*impl x*) $\neq \infty$}) $\Longrightarrow$
$x \in$ *vertex* ‘ *dVs* ((λx. (*fst x*, *snd x*)) ‘ *set* $\mathcal{E}$*-impl*) **for** *x*
proof(*goal-cases*)
case *1*
note *2* = *this*
then obtain *e* **where** $x = edge\ e \lor x = vertex\ (snd\ e)$ $e \in$ *set* $\mathcal{E}$*-impl*
the (*flow-lookup* u-*impl e*) $\neq \infty$ **by**(*auto simp add*: *dVs-def*)
moreover hence $x \neq edge\ e$ **using** *u-domain 2*(*1*)
by(*force simp add*: *dom-def the-default-def Es-are case-simp*(*1*) *to-set-def*)
ultimately show *?case*
unfolding *dVs-def*
by(*fastforce intro*!: *imageI intro*: *exI*[*of* - {*fst e*, *snd e*}] *simp add*: *dVs-def*)
qed
moreover have *3*:$x \notin edge$ ‘
(*set* $\mathcal{E}$*-impl* − {$e \in$ *set* $\mathcal{E}$*-impl*. *the-default* ∞ (*flow-lookup* u-*impl e*) = ∞}) $\Longrightarrow$
$x \in dVs$ ((λx. (*vertex* (*fst x*), *vertex* (*snd x*))) ‘
{$x \in$ *set* $\mathcal{E}$*-impl*. *the* (*flow-lookup* u-*impl x*) = ∞}) $\Longrightarrow$
$x \in$ *vertex* ‘ *dVs* ((λx. (*fst x*, *snd x*)) ‘ *set* $\mathcal{E}$*-impl*) **for** *x*
proof(*goal-cases*)
case *1*

note *3=1*
then obtain *e* **where** $x = vertex\ (fst\ e) \lor x = vertex\ (snd\ e)$ $e \in set\ \mathcal{E}\text{-}impl$
$the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e) = \infty$ **by**(*auto simp add*: *dVs-def*)
thus *?case*
unfolding *dVs-def*
by(*fastforce intro*!: *imageI intro*: *exI*[*of* - {*fst e*, *snd e*}] *simp add*: *dVs-def*)
qed
moreover have *4*:*vertex xa*
$\notin dVs\ ((\lambda x.\ (edge\ x,\ vertex\ (fst\ x)))$ ‘
$\{x \in set\ \mathcal{E}\text{-}impl.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ x) \neq \infty\}) \Longrightarrow$
vertex xa
$\notin dVs\ ((\lambda x.\ (vertex\ (fst\ x),\ vertex\ (snd\ x)))$ ‘
$\{x \in set\ \mathcal{E}\text{-}impl.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ x) = \infty\}) \Longrightarrow$
$xa \in dVs\ ((\lambda x.\ (fst\ x,\ snd\ x))$ ‘ $set\ \mathcal{E}\text{-}impl) \Longrightarrow$
vertex xa
$\in dVs\ ((\lambda x.\ (edge\ x,\ vertex\ (snd\ x)))$ ‘
$\{x \in set\ \mathcal{E}\text{-}impl.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ x) \neq \infty\})$ **for** *xa*
proof(*goal-cases*)
case *1*
note *4 = 1*
obtain *e* **where** *e-prop*:$xa = fst\ e \lor xa = snd\ e$ $e \in set\ \mathcal{E}\text{-}impl$
using *4*(*3*) **by** (*auto simp add*: *dVs-def make-pair*)
show *?case*
proof(*rule disjE*[*OF e-prop*(*1*)], *goal-cases*)
case *1*
hence $the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e) = \infty$
using *4*(*1*) *e-prop*(*2*)
by(*auto simp add*:*dVs-def*)
moreover have $the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e) \neq \infty$
using *4*(*2*) *e-prop*(*2*) *1*
by(*auto simp add*:*dVs-def*)
ultimately show *?case* **by** *simp*
next
case *2*
have $the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ e) \neq \infty$
using *4*(*2*) *e-prop*(*2*) *2*
by(*auto simp add*:*dVs-def*)
then show *?case*
using *2 e-prop*(*2*) **by** *auto*
qed
qed
moreover have *5*:*edge e*
$\notin dVs\ ((\lambda x.\ (edge\ x,\ vertex\ (fst\ x)))$ ‘
$\{x \in set\ \mathcal{E}\text{-}impl.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ x) \neq \infty\}) \Longrightarrow$
edge e
$\notin dVs\ ((\lambda x.\ (vertex\ (fst\ x),\ vertex\ (snd\ x)))$ ‘
$\{x \in set\ \mathcal{E}\text{-}impl.\ the\ (flow\text{-}lookup\ \mathrm{u}\text{-}impl\ x) = \infty\}) \Longrightarrow$
$e \in set\ \mathcal{E}\text{-}impl \Longrightarrow$
edge e

$\notin$ *dVs* ((λ*x*. (*edge x*, *vertex* (*snd x*))) ‘
{*x* ∈ *set* $\mathcal{E}$-*impl*. *the* (*flow-lookup* u-*impl x*) ≠ ∞}) ⟹
the-default ∞ (*flow-lookup* u-*impl e*) = ∞ **for** *e*
proof(*goal-cases*)
case *1*
note *5* = *1*
have *the* (*flow-lookup* u-*impl e*) = ∞
using *5*(*1*) *5*(*3*) **by**(*auto simp add*:*dVs-def*)
moreover have *e* ∈ *dom*(*flow-lookup* u-*impl*)
using *u-domain Es-are 5*(*3*)
by(*auto simp add*:*the-default-def to-set-def dom-def*)
ultimately show *?case*
by(*auto simp add*: *dom-def the-default-def*)
qed
show *?thesis*
by(*subst infty-edges-def*)
(*auto simp add*: $\mathcal{E}'$-*impl-def* $\mathcal{E}$*1-impl-def* $\mathcal{E}$*2-impl-def* $\mathcal{E}$-*impl-finite-def* $\mathcal{E}$-*impl-infty-def*
$\mathcal{E}$*3-impl-def*
to-set-def new-fstv-gen-def new-sndv-gen-def multigraph-spec.make-pair-def
image-Un image-comp Es-are us-are intro: *1 2 3 4 5*)
qed

lemma *filter-neg-filter-empty*:*filter P xs* = *ys* ⟹ *filter* (λ *x*. ¬ *P x*) *xs* = *zs*
⟹ *ys* = [] ⟹ *zs* = [] ⟹ *xs* = []
by(*induction ys*, *all* ‹*induction xs*›, *auto*)
(*meson list.discI*)

lemma *E'-non-empt*:*to-list* $\mathcal{E}'$-*impl* ≠ []
using *E-not-empty filter-neg-filter-empty*
by(*auto simp add*: *to-list-def* $\mathcal{E}'$-*impl-def* $\mathcal{E}$*1-impl-def* $\mathcal{E}$*2-impl-def* $\mathcal{E}$*3-impl-def*
Es-are
$\mathcal{E}$-*impl-infty-def* $\mathcal{E}$-*impl-finite-def to-set-def*)

lemma *finite-E'*:*finite* (*set* $\mathcal{E}'$-*impl*)
by(*auto simp add*: *to-list-def* $\mathcal{E}'$-*impl-def* $\mathcal{E}$*1-impl-def* $\mathcal{E}$*2-impl-def* $\mathcal{E}$*3-impl-def*
Es-are
$\mathcal{E}$-*impl-infty-def* $\mathcal{E}$-*impl-finite-def to-set-def*)

lemma *multigraph'*:*multigraph* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*)
new-create-edge-gen
(*function-generation*.$\mathcal{E}$ $\mathcal{E}'$-*impl to-set*)
using *finite-E' E'-non-empt*
by(*auto intro*: *multigraph.intro*
simp add: *new-create-edge-gen-def new-fstv-gen-def new-sndv-gen-def*
to-set-def to-list-def function-generation.$\mathcal{E}$-*def*[*OF function-generation*])

lemma *collapse-union-ofE1E2E3*:*to-set* $\mathcal{E}$*1-impl* ∪ *to-set* $\mathcal{E}$*2-impl* ∪ *to-set* $\mathcal{E}$*3-impl*
= *to-set* $\mathcal{E}'$-*impl*

by (*simp add*: *Un-assoc* $\mathcal{E}'$*-impl-def to-set-def*)

lemma *E1-are*: *to-set* $\mathcal{E}$*1-impl* = *inedge* ' ($\mathcal{E}$ − $\mathcal{E}_\infty$)
  using *u-domain infty-edges-def dom-def*
  by(*fastforce split*: *option.split*
      simp add: $\mathcal{E}$*1-impl-def Es-are* $\mathcal{E}$*-impl-finite-def to-set-def us-are the-default-def*)

lemma *E2-are*: *to-set* $\mathcal{E}$*2-impl* = *outedge* ' ($\mathcal{E}$ − $\mathcal{E}_\infty$)
  using *u-domain infty-edges-def dom-def*
  by(*fastforce split*: *option.split*
      simp add: $\mathcal{E}$*2-impl-def Es-are* $\mathcal{E}$*-impl-finite-def to-set-def us-are the-default-def*)

lemma *E3-are*: *to-set* $\mathcal{E}$*3-impl* = *vtovedge* ' ($\mathcal{E}_\infty$)
  using *u-domain infty-edges-def dom-def*
  by(*fastforce split*: *option.split*
      simp add: $\mathcal{E}$*3-impl-def Es-are* $\mathcal{E}$*-impl-infty-def to-set-def us-are the-default-def*)

interpretation *correctness-of-algo-red*: *correctness-of-algo*
  where *fst* =*new-fstv-gen fst*
and *snd* =*new-sndv-gen fst snd*
and c*-impl* = c′*-impl*
and $\mathcal{E}$*-impl* = $\mathcal{E}'$*-impl*
and *create-edge* = *new-create-edge-gen*
and b*-impl* = b′*-impl*
and *c-lookup* = *c-lookup*′
  using *set-invar-E*′ *bal-invar-b*′*-impl b*′*-impl-dom V-new-graph E*′*-non-empt multigraph*′
  by(*intro correctness-of-algo.intro*)
    (*auto simp add*: *function-generation.N-def*[*OF function-generation*])

lemma *E*′*-impl-in-cost*′*-dom*:*e* ∈ *set* $\mathcal{E}'$*-impl* ⟹ *e* ∈ *dom* (*c-lookup*′ c′*-impl*)
  using *c-domain u-domain*
  by(*force simp add*: $\mathcal{E}'$*-impl-def* $\mathcal{E}$*1-impl-def* $\mathcal{E}$*2-impl-def* $\mathcal{E}$*3-impl-def* c′*-impl-def Let-def c-lookup*′*-def* $\mathcal{E}$*-impl-finite-def dom-def* $\mathcal{E}$*-impl-infty-def Es-are to-set-def image-def*)

lemma *c*′*-dom-is*: *dom* (*c-lookup*′ c′*-impl*) =
      inedge ' *UNIV* ∪ *vtovedge* ' *dom* (*c-lookup* c*-impl*) ∪ *outedge* ' *dom* (*c-lookup* c*-impl*)
proof(*rule*, *all* ‹*rule*›, *goal-cases*)
  case (*1 x*)
  show *?case*
  proof(*cases x*)
    case (*outedge x1*)
    hence *x1* ∈ *dom* (*c-lookup* c*-impl*)
      using *1* **by**(*auto simp add*: *c-lookup*′*-def* c′*-impl-def*)

then show *?thesis*
using *outedge c-domain* **by** *simp*
next
case (*inedge x2*)
then show *?thesis* **by** *simp*
next
case (*vtovedge x3*)
hence *x3* $\in$ *dom* (*c-lookup* c-*impl*)
using *1* **by**(*auto simp add*: *c-lookup*′*-def* c′*-impl-def*)
then show *?thesis*
using *vtovedge c-domain* **by** *simp*
next
case (*dummy x41 x42*)
then show *?thesis*
using *1* **by**(*auto simp add*: *c-lookup*′*-def dom-def*)
qed
next
case (*2 x*)
then show *?case*
using *c-domain*
by(*force simp add*: $\mathcal{E}$′*-impl-def* $\mathcal{E}$*1-impl-def* $\mathcal{E}$*2-impl-def* $\mathcal{E}$*3-impl-def* c′*-impl-def Let-def c-lookup*′*-def* $\mathcal{E}$*-impl-finite-def*
dom-def $\mathcal{E}$*-impl-infty-def Es-are to-set-def image-def*)
qed

lemma *c*′*-impl-lookup*:*x* $\in$ *set* $\mathcal{E}$′*-impl* $\Longrightarrow$ *the* (*c-lookup*′ c′*-impl x*) = *new*-c-*gen D fst* $\mathcal{E}$ u c *x*
by(*auto split*: *hitchcock-edge.split*
simp add: $\mathcal{E}$′*-impl-def* $\mathcal{E}$*3-impl-def* $\mathcal{E}$*2-impl-def* $\mathcal{E}$*1-impl-def to-set-def cs-are*
new-c-*gen-def new-fstv-gen-def sym*[*OF E1-impl-are*] *sym*[*OF E2-impl-are*] *sym*[*OF E3-impl-are*]
c′*-impl-def c-lookup*′*-def*)+

lemma *new-gen-c-unfold*:*new*-c-*gen* (*dom* (*c-lookup* c-*impl*)) *fst* $\mathcal{E}$ u c = *Instantiation*.c c′*-impl c-lookup*′
unfolding *selection-functions*.c*-def*
apply(*rule ext*)
subgoal for *e*
apply(*cases e* $\in$ *set* $\mathcal{E}$′*-impl*)
subgoal
using *E*′*-impl-in-cost*′*-dom*[*of e*] *c*′*-impl-lookup*[*of e* (*dom* (*c-lookup* c-*impl*)), *symmetric*]
by (*fastforce intro*: *option-Some-theE*[*of - the* (*c-lookup*′ c′*-impl e*)])
subgoal
using *c-domain*
by(*auto split*: *hitchcock-edge.split simp add*: *c-lookup*′*-def* c′*-impl-def dom-def cs-are*
sym[*OF E1-impl-are*] *sym*[*OF E2-impl-are*] *sym*[*OF E3-impl-are*]
sym[*OF collapse-union-ofE1E2E3*, *simplified to-set-def*]

*to-set-def new-*c*-gen-def*)
done
done

lemma *new-b-domain-cong*: $x \in$ *vertex* ' $\mathcal{V} \cup$ *edge* ' $(\mathcal{E} - \mathcal{E}_\infty) \Longrightarrow$ *new-*b*-gen fst* $\mathcal{E}$ u b x = *selection-functions.*b b'*-impl* x
by(*auto simp add*: *selection-functions.*b*-def new-*b*-gen-def new-*b*-gen-def b'-impl-lookup b'-impl-dom*[*simplified dom-def*, *symmetric*])

lemma *cost-flow-network3*: *cost-flow-network* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*)
new-create-edge-gen ($\lambda e.$ *PInfty*) (*to-set* $\mathcal{E}'$*-impl*)
apply(*rule cost-flow-network.intro*)
apply(*rule flow-network.intro*)
subgoal
using *multigraph'*
by(*auto split*: *hitchcock-edge.split*
simp add: *function-generation.*$\mathcal{E}$*-def*[*OF function-generation*] *comp-def*
new-fstv-gen-def new-sndv-gen-def)
by(*auto intro*: *flow-network-axioms.intro*)

context
assumes *no-infinite-cycle*: ¬ *has-neg-infty-cycle make-pair* $\mathcal{E}$ c u
begin

lemma *no-cycle-in-reduction*:*no-cycle-cond* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*) c'*-impl* $\mathcal{E}'$*-impl c-lookup'*
proof(*rule no-cycle-condI*, *goal-cases*)
case (*1 C*)
hence *has-neg-cycle* (*multigraph-spec.make-pair* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*)) (*to-set* $\mathcal{E}'$*-impl*)
(*function-generation.*c c'*-impl c-lookup'*)
by(*auto intro*!: *has-neg-cycleI*[*of - - C*]
simp add: *function-generation.*$\mathcal{E}$*-def*[*OF function-generation*]
add.commute[*of - -* c'*-impl c-lookup' -*])
hence *has-neg-infty-cycle local.make-pair* $\mathcal{E}$ c u
using *sym*[*OF reduction-of-mincost-flow-to-hitchcock-general*(*4*)[*OF flow-network-axioms*, *of* (*dom* (*c-lookup* c*-impl*)) c]]
unfolding *sym*[*OF E1-impl-are*] *sym*[*OF E2-impl-are*] *sym*[*OF E3-impl-are*]
collapse-union-ofE1E2E3 function-generation.$\mathcal{E}$*-def*[*OF function-generation*]
new-gen-c-unfold
by(*auto simp add*: *Es-are cs-are* c*-def*)
thus *False*
using *no-infinite-cycle* **by** *simp*
qed

corollary *correctness-of-implementation-success*:
return (*final-state-cap*) = *success* $\Longrightarrow$

is-Opt b (*abstract-flow-map* (*final-flow-impl-original*))
apply(*rule is-Opt-cong*[*of old-f-gen* $\mathcal{E}$ u (*abstract-flow-map final-flow-impl-cap*)
, *OF old-f-gen-final-flow-impl-original-cong refl*], *simp*)
apply(*rule reduction-of-mincost-flow-to-hitchcock-general*(*5*)[*OF flow-network-axioms refl*, *of* (*dom* (*c-lookup* c-*impl*)) c b])
apply(*unfold final-flow-impl-cap-def sym*[*OF E1-impl-are*] *sym*[*OF E2-impl-are*] *sym*[*OF E3-impl-are*]
collapse-union-ofE1E2E3 u-*def function-generation*.u-*def*[*OF function-generation*])
apply(*unfold new-gen-c-unfold*)
using *V-new-graph no-cycle-in-reduction*
by(*fastforce simp add*: *final-state-cap-def*
intro!: *cost-flow-spec.is-Opt-cong*[*OF refl sym*[*OF new-b-domain-cong*]]
correctness-of-algo.correctness-of-implementation(*1*)
[*OF correctness-of-algo-red.correctness-of-algo-axioms*, *of* c′-*impl*,
simplified u-*def function-generation*.u-*def*[*OF function-generation*]
$\mathcal{E}$-*def function-generation*.$\mathcal{E}$-*def*[*OF function-generation*]])

corollary *correctness-of-implementation-infeasible*:
return (*final-state-cap*) = *infeasible* $\Longrightarrow$
$\nexists$ *f*. *isbflow f* b
proof(*rule nexistsI*, *goal-cases*)
case (*1 f*)
have *flow-network-spec.isbflow* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*) (*to-set* $\mathcal{E}$′-*impl*) (λ*e*. *PInfty*)
(*new-f-gen fst* $\mathcal{E}$ u *f*)
(*selection-functions*.b b′-*impl*)
apply(*rule cost-flow-spec.isbflow-cong*[*OF refl*])
using *V-new-graph conjunct1*[*OF reduction-of-mincost-flow-to-hitchcock-general*(*2*)[*OF flow-network-axioms*
1(*2*) *refl*, *of* (λ -. *0*)]]
by(*auto intro*: *new-b-domain-cong*
simp add: *sym*[*OF E1-impl-are*] *sym*[*OF E2-impl-are*] *sym*[*OF E3-impl-are*]
collapse-union-ofE1E2E3)
moreover have $\nexists$*f*. *flow-network-spec.isbflow* (*new-fstv-gen fst*) (*new-sndv-gen fst snd*) (*to-set* $\mathcal{E}$′-*impl*) (λ*e*. *PInfty*) *f*
(*selection-functions*.b b′-*impl*)
using *no-cycle-in-reduction 1*(*1*)
by(*intro correctness-of-algo.correctness-of-implementation*(*2*)
[*OF correctness-of-algo-red.correctness-of-algo-axioms*, *of* c′-*impl*,
simplified u-*def function-generation*.u-*def*[*OF function-generation*]
$\mathcal{E}$-*def function-generation*.$\mathcal{E}$-*def*[*OF function-generation*]])
(*auto simp add*: *final-state-cap-def*)
ultimately show *?case* **by** *simp*
qed

corollary *correctness-of-implementation-excluded-case*:
return final-state-cap = *notyetterm* $\Longrightarrow$ *False*

using *no-cycle-in-reduction*
by(*auto intro*: *correctness-of-algo.correctness-of-implementation*(*3*)
[*OF correctness-of-algo-red.correctness-of-algo-axioms*, *of* c′*-impl*] *simp add*: *final-state-cap-def*)

lemmas *correctness-of-implementation* = *correctness-of-implementation-success*
correctness-of-implementation-infeasible
correctness-of-implementation-excluded-case

end
definition *make-pair-capacity* = *make-pair*
end
lemmas *make-pair-capacity-def*[*code*] = *multigraph-spec.make-pair-def*
global-interpretation *flow-with-capacity*: *with-capacity*
where *fst* = *fst*
and *snd* = *snd*
and *create-edge* = *create-edge*
and $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl*
and c*-impl* = c*-impl*
and u*-impl* = u*-impl*
and b*-impl* = b*-impl*
and *c-lookup* = *c-lookup*
for *fst snd create-edge* $\mathcal{E}$*-impl* c*-impl* u*-impl* b*-impl* *c-lookup*
defines *final-flow-impl-cap* = *flow-with-capacity.final-flow-impl-cap*
and *final-state-cap*=*flow-with-capacity.final-state-cap*
and *final-flow-impl-original* = *flow-with-capacity.final-flow-impl-original*

done

definition $\mathcal{E}$*-impl* = [(*1*::*nat*, *2*::*nat*), (*1*,*3*), (*3*,*2*), (*2*,*4*), (*2*,*5*), (*3*,*5*), (*4*,*6*), (*6*,*5*), (*2*,*6*)]
value $\mathcal{E}$*-impl*

definition *b-list* = [(*1*::*nat*,*128*::*real*), (*2*,*0*), (*3*,*1*), (*4*,−*33*), (*5*,−*32*), (*6*,−*64*)]

definition b*-impl* = *foldr* (λ *xy tree. update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *b-list Leaf*
value b*-impl*

definition *c-list* = [((*1*::*nat*, *2*::*nat*), *1*::*real*),
((*1*,*3*), *4*), ((*3*,*2*), *2*), ((*2*,*4*), *3*), ((*2*,*5*), *1*),
((*3*,*5*), *5*), ((*4*,*6*), *2*), ((*6*,*5*), *1*), ((*2*,*6*), *9*)]

definition c*-impl* = *foldr* (λ *xy tree. update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *c-list Leaf*
value c*-impl*

definition *u-list* = [((*1*::*nat*, *2*::*nat*), *20*),
((*1*,*3*), *108*), ((*3*,*2*), *PInfty*), ((*2*,*4*), *PInfty*), ((*2*,*5*), *PInfty*),

((*3*,*5*), *PInfty*), ((*4*,*6*), *45*), ((*6*,*5*), *PInfty*), ((*2*,*6*), *PInfty*)]

definition u-*impl* = *foldr* (λ *xy tree*. *update* (*prod.fst xy*) (*prod.snd xy*) *tree*) *u-list* *Leaf*
value u-*impl*

value *final-state-cap fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* b-*impl flow-lookup*
value *final-flow-impl-cap fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* b-*impl flow-lookup*
value *final-flow-impl-original fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* b-*impl flow-lookup*
value *inorder* (*final-flow-impl-original fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* b-*impl flow-lookup*)

instantiation *edge-wrapper*::(*linorder*) *linorder*
begin

fun *less-eq-edge-wrapper*::*'a edge-wrapper* ⇒ *'a edge-wrapper* ⇒ *bool* **where**
less-eq-edge-wrapper (*old-edge e*) (*old-edge d*) = (*e* ≤ *d*)|
less-eq-edge-wrapper (*new-edge e*) (*new-edge d*) = (*e* ≤ *d*)|
less-eq-edge-wrapper (*new-edge e*) (*old-edge d*) = *False*|
less-eq-edge-wrapper (*old-edge e*) (*new-edge d*) = *True*

fun *less-edge-wrapper*::*'a edge-wrapper* ⇒ *'a edge-wrapper* ⇒ *bool* **where**
less-edge-wrapper (*old-edge e*) (*old-edge d*) = (*e* < *d*)|
less-edge-wrapper (*new-edge e*) (*new-edge d*) = (*e* < *d*)|
less-edge-wrapper (*new-edge e*) (*old-edge d*) = *False*|
less-edge-wrapper (*old-edge e*) (*new-edge d*) = *True*

instance
  apply(*intro Orderings.linorder.intro-of-class class.linorder.intro*
            class.order-axioms.intro class.order.intro class.preorder.intro
            class.linorder-axioms.intro)
  subgoal for *x y*
    by(*all* ‹*cases x*›, *all* ‹*cases y*›) *force+*
  subgoal for *x*
    by(*cases x*) *auto*
   subgoal for *x y z*
     by(*all* ‹*cases x*›, *all* ‹*cases y*›, *all* ‹*cases z*›)
       (*auto split*: *if-split simp add*: *less-le-not-le*)
  subgoal for *a b*
    by(*all* ‹*cases a*›, *all* ‹*cases b*›)
     (*auto split*: *if-split simp add*: *less-le-not-le*)
  subgoal for *a b*
   by(*all* ‹*cases a*›, *all* ‹*cases b*›)
     (*auto split*: *if-split simp add*: *less-le-not-le*)
  done
end

datatype *cost-dummy* = *cost-dummy*

locale *solve-maxflow* =

fixes *fst*::(′*edge-type*::*linorder*) ⇒ (′*a*::*linorder*)
and *snd*::(′*edge-type*::*linorder*) ⇒ (′*a*::*linorder*)
and *create-edge*::′*a* ⇒ ′*a* ⇒ ′*edge-type*
and *ℰ-impl*::′*edge-type list*
and u-*impl*:: ((′*edge-type*::*linorder* × *ereal*) × *color*) *tree*
and *s*::′*a*
and *t*::′*a*
begin

definition *ℰ-impl′* = *map old-edge ℰ-impl* @ [*new-edge* (*create-edge t s*)]

definition c-*impl′* = *cost-dummy*

definition *c-lookup′ c* (*e*::′*edge-type edge-wrapper*) = (*case e of old-edge* - ⇒ *Some* (*0*::*real*) |
new-edge - ⇒ *Some* (−*1*))

definition b-*impl′* = *foldr* (λ *x tree. bal-update x 0 tree*) (*vs fst snd ℰ-impl*) *Leaf*

definition *u-sum* = *foldr* (λ *e acc. acc* + *the* (*flow-lookup* u-*impl e*)) *ℰ-impl 0*

definition u-*impl′* = *flow-update* (*new-edge* (*create-edge t s*)) *u-sum*
(*foldr* (λ *e tree. flow-update* (*old-edge e*) (*the* (*flow-lookup* u-*impl* *e*)) *tree*) *ℰ-impl Leaf*)

definition *final-state-maxflow* = *final-state-cap*
(λ *e. case e of old-edge e* ⇒ *fst e* | *new-edge e* ⇒ *fst e*)
(λ *e. case e of old-edge e* ⇒ *snd e* | *new-edge e* ⇒ *snd e*)
ℰ-impl′ c-*impl′* u-*impl′* b-*impl′* *c-lookup′*

definition *final-flow-impl-maxflow* = *final-flow-impl-original*
(λ *e. case e of old-edge e* ⇒ *fst e* | *new-edge e* ⇒ *fst e*)
(λ *e. case e of old-edge e* ⇒ *snd e* | *new-edge e* ⇒ *snd e*)
ℰ-impl′ c-*impl′* u-*impl′* b-*impl′* *c-lookup′*

definition *final-flow-impl-maxflow-original* =
(*foldr* (λ *e tree. flow-update e*
(*the-default 0* (*flow-lookup final-flow-impl-maxflow* (*old-edge* *e*))) *tree*)
ℰ-impl flow-empty)
end

global-interpretation *solve-maxflow-by-orlins*: *solve-maxflow* **where**
fst = *fst*
and *snd* = *snd*
and *create-edge* = *create-edge*
and *ℰ-impl* = *ℰ-impl*
and u-*impl* = u-*impl*
and *s* = *s*

and $t = t$
for *fst snd create-edge* $\mathcal{E}$*-impl* u-*impl s t*
defines *final-state-maxflow* = *solve-maxflow.final-state-maxflow*
and *final-flow-impl-maxflow* = *solve-maxflow.final-flow-impl-maxflow*
and *final-flow-impl-maxflow-original* = *solve-maxflow.final-flow-impl-maxflow-original*
  done

lemma *capacity-Opt-cong*:
  fixes *fst snd make-pair u c E b f create-edge*
  assumes *cost-flow-network1*: *cost-flow-network fst snd create-edge u E*
    and *cost-flow-network2*: *cost-flow-network fst snd create-edge u' E*
    and $\bigwedge$ *e*. $e \in E \Longrightarrow u\ e = u'\ e$
    and *cost-flow-spec.is-Opt fst snd u E c b f*
  shows *cost-flow-spec.is-Opt fst snd u' E c b f*
  using *assms*(*3*,*4*)
  by(*simp add*: *cost-flow-spec.is-Opt-def flow-network-spec.isbflow-def*
        flow-network-spec.isuflow-def)

lemma *capacity-bflow-cong*:
  fixes *fst snd make-pair u c E b f create-edge*
  assumes *cost-flow-network1*: *flow-network fst snd create-edge u E*
    and *cost-flow-network2*: *flow-network fst snd create-edge u' E*
    and $\bigwedge$ *e*. $e \in E \Longrightarrow u\ e = u'\ e$
    and *flow-network-spec.isbflow fst snd E u b f*
  shows *flow-network-spec.isbflow fst snd E u' b f*
  using *assms*(*3*,*4*)
  by(*simp add*: *flow-network-spec.isbflow-def flow-network-spec.isuflow-def*)

locale *solve-maxflow-proofs* =
solve-maxflow **where** *fst* = *fst*::'*edge-type*::*linorder* ⇒ '*a*::*linorder*
and *snd* = *snd*::'*edge-type*::*linorder* ⇒ '*a*::*linorder*
and *create-edge* = *create-edge*
and $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl*
and u-*impl* = u-*impl* +

flow-network **where** *fst* = *fst*
and *snd* = *snd*
and *create-edge* = *create-edge*
and $\mathcal{E} = \mathcal{E}$
and u = u

for *fst snd create-edge* $\mathcal{E}$*-impl* u-*impl* $\mathcal{E}$ u+
assumes *u-domain*: *dom* (*flow-lookup* u-*impl*) = $\mathcal{E}$
and *set-invar-E*: *set-invar* $\mathcal{E}$*-impl*
and *Es-are*: $\mathcal{E}$ = *to-set* $\mathcal{E}$*-impl*
and *us-are*: u = *the-default PInfty o* (*flow-lookup* u-*impl*)
assumes *s-in-V*: $s \in \mathcal{V}$
assumes *t-in-V*: $t \in \mathcal{V}$
assumes *s-neq-t*: $s \neq t$

begin

definition c′ = *the o* (*c-lookup′ cost-dummy*)
definition u′ = *the-default PInfty o* (*flow-lookup* u-*impl′*)

lemma *in-E-same-cap*:*e* ∈ *set* $\mathcal{E}$*-impl* ⟹ *flow-lookup* u-*impl′* (*old-edge e*) = *flow-lookup* u-*impl e*
unfolding u-*impl′-def Es-are*
apply(*subst foldr-map*[*of* (λ*e. flow-update e* (*the* (*flow-lookup* u-*impl* (*get-old-edge e*))))
old-edge, *simplified comp-def*, *simplified*, *symmetric*])
apply(*subst flow-map.map-update*)
using *u-domain*
by(*force intro*: *flow-invar-fold*[*OF flow-invar-Leaf*]
simp add: *flow-map.invar-update dom-def Es-are to-set-def flow-lookup-fold*[*OF flow-invar-Leaf*])+

lemma *dom-final-flow-impl-maxflow*:*dom* (*flow-lookup final-flow-impl-maxflow*) = *set* $\mathcal{E}$*-impl′*
by(*simp add*: *final-flow-impl-maxflow-def flow-with-capacity.dom-final-flow-impl-original*)

lemma *abstract-flows-are*:*abstract-flow-map final-flow-impl-maxflow-original* = (λ*e. abstract-flow-map final-flow-impl-maxflow* (*old-edge e*))
using *dom-final-flow-impl-maxflow*
by (*fastforce simp add*: *flow-lookup-fold flow-map.invar-empty the-default-def*
flow-map.map-empty $\mathcal{E}$*-impl′-def dom-def abstract-real-map-def*
final-flow-impl-maxflow-original-def abstract-flow-map-def)

lemma *multigraph′*: *multigraph* (*prod.fst* ∘ *make-pair′*) (*prod.snd* ∘ *make-pair′*) *create-edge′* (*set* $\mathcal{E}$*-impl′*)
by(*auto intro*!: *multigraph.intro simp add*: *finite-E fst-create-edge snd-create-edge* $\mathcal{E}$*-impl′-def*)

lemma *flow-network-axioms′*: *flow-network-axioms* (λ*e. case flow-lookup* u-*impl′ e of None* ⇒ *PInfty* |
Some - ⇒ *case e of old-edge e* ⇒ u *e*
| *new-edge b* ⇒ *sum* u $\mathcal{E}$)
using *u-sum-pos u-non-neg*
by(*auto intro*!: *flow-network-axioms.intro split*: *edge-wrapper.split option.split*)

lemma *dom-u′-impl*: *dom* (*flow-lookup* u-*impl′*) = *set* $\mathcal{E}$*-impl′*
unfolding u-*impl′-def* $\mathcal{E}$*-impl′-def*
apply(*subst dom-update-insert*[*simplified sym*[*OF flow-lookup-def*] *sym*[*OF flow-update-def*]])
by(*auto intro*!: *conjunct1*[*OF flow-invar-fold*[*simplified flow-invar-def*]]
flow-map.invar-update[*simplified flow-invar-def*]
simp add: *flow-invar-Leaf*[*simplified flow-invar-def*] *dom-fold flow-invar-Leaf flow-map.map-empty* [*simplified RBT-Set.empty-def flow-empty-def*])

lemma *dom-b′-impl*: *dom* (*bal-lookup* b-*impl′*) = $\mathcal{V}$

by(*force simp add*: *dVs-eq dVs-swap Es-are to-set-def*
vs-def function-generation.vs-def[*OF function-generation*]
function-generation.es-def[*OF function-generation*] *to-list-def*
bal-map.map-specs(*1*)[*simplified RBT-Set.empty-def bal-empty-def*]
bal-map.invar-empty[*simplified RBT-Set.empty-def bal-empty-def*]
bal-dom-fold b-*impl′-def*
image-comp make-pair″(*3*) *selection-functions.make-pair-def*
make-pair″(*2*) *image-iff*)

lemma *set-invar′*:*set-invar* $\mathcal{E}$-*impl′*
using *set-invar-E*
by(*auto simp add*: *distinct-map inj-on-def set-invar-def* $\mathcal{E}$-*impl′-def*)

lemma *bal-invar′*:*bal-invar* b-*impl′*
by(*auto intro*: *bal-invar-fold simp add*: b-*impl′-def bal-map.invar-empty*[*simplified RBT-Set.empty-def bal-empty-def*])

lemma *u-impl′-same-u*:*flow-lookup* u-*impl′* (*old-edge e*) = *Some u* $\Longrightarrow$ u *e* = *u*
unfolding u-*impl′-def*
apply(*subst* (*asm*) *flow-map.map-update*)
apply (*simp add*: *flow-invar-Leaf flow-invar-fold flow-map.invar-update*, *simp*)
apply(*subst* (*asm*) *foldr-map*[*of* (*λe. flow-update e* (*the* (*flow-lookup* u-*impl* (*get-old-edge e*)))) *old-edge*,
simplified comp-def, *simplified*, *symmetric*])
apply(*subst* (*asm*) *flow-lookup-fold*)
apply (*simp add*: *flow-invar-Leaf*)
using *u-domain*
by (*cases old-edge e* ∈ *old-edge* ‘ *set* $\mathcal{E}$-*impl*)
(*force simp add*: *flow-map.map-empty*[*simplified RBT-Set.empty-def flow-empty-def*]
us-are the-default-def Es-are to-set-def dom-def)+

lemma *u-sum-is*: *u-sum* = *sum* u (*set* $\mathcal{E}$-*impl*)
unfolding *u-sum-def*
using *set-invar-E u-domain us-are*
by(*subst distinct-sum*)(*force intro*: *foldr-cong simp add*: *Es-are to-set-def the-default-def set-invar-def*)+

lemma *u-impl′-sum*:*flow-lookup* u-*impl′* (*new-edge e*) = *Some u* $\Longrightarrow$ *sum* u (*set* $\mathcal{E}$-*impl*) = *u*
unfolding u-*impl′-def*
apply(*subst* (*asm*) *flow-map.map-update*)
apply (*simp add*: *flow-invar-Leaf flow-invar-fold flow-map.invar-update*, *simp*)
apply(*subst* (*asm*) *foldr-map*[*of* (*λe. flow-update e* (*the* (*flow-lookup* u-*impl* (*get-old-edge e*)))) *old-edge*,
simplified comp-def, *simplified*, *symmetric*])
apply(*subst* (*asm*) *flow-lookup-fold*)
apply (*simp add*: *flow-invar-Leaf*)
apply(*subst* (*asm*) *flow-map.map-empty*[*simplified RBT-Set.empty-def flow-empty-def*])
by(*cases e* = *create-edge t s*)(*auto simp add*: *u-sum-is image-iff*)

lemma *with-capacity-proofs-axioms*:
with-capacity-proofs-axioms (*prod.snd o make-pair′*) c-*impl′* b-*impl′* *c-lookup′* (*prod.fst o make-pair′*) $\mathcal{E}$-*impl′* u-*impl′* (*set* $\mathcal{E}$-*impl′*)
(λ*e*. *case flow-lookup* u-*impl′* *e of None* ⇒ *PInfty* |
Some - ⇒ *case e of old-edge e* ⇒ u *e*
| *new-edge b* ⇒ *sum* u $\mathcal{E}$)
(λ*e*. *case e of old-edge x* ⇒ *0* | *new-edge b* ⇒ − *1*)
(*the-default 0* ∘ *bal-lookup* b-*impl′*)
using *dom-u′-impl same-Vs-s-t*[*OF s-in-V t-in-V s-neq-t*] *dom-b′-impl set-invar′ bal-invar′ u-impl′-same-u u-impl′-sum*
by(*auto intro*!: *with-capacity-proofs-axioms.intro split*: *edge-wrapper.split option.split*

simp add: *the-default-def comp-def to-set-def* $\mathcal{E}$-*impl′-def Es-are c-lookup′-def*
make-pair-def multigraph-spec.make-pair)

lemma *with-capacity-proofs*:*with-capacity-proofs snd′* c-*impl′* b-*impl′*
c-lookup′ fst′ create-edge′ $\mathcal{E}$-*impl′* u-*impl′* (*set* $\mathcal{E}$-*impl′*)
(λ*e*. *case flow-lookup* u-*impl′* *e of None* ⇒ *PInfty* |
Some - ⇒ *case e of old-edge e* ⇒ u *e*
| *new-edge b* ⇒ *sum* u $\mathcal{E}$)
(*case-edge-wrapper* (λ*x*. *0*) (λ*b*. − *1*))
(*the-default 0* ∘ *bal-lookup* b-*impl′*)
using *multigraph′ flow-network-axioms′ with-capacity-proofs-axioms*
by(*auto intro*!: *with-capacity-proofs.intro cost-flow-network.intro flow-network.intro*
simp add: *fst′-def snd′-def*)

lemma *cost-flow-network1*: *cost-flow-network fst′ snd′ create-edge′* (*case-edge-wrapper* u (λ*b*. *sum* u $\mathcal{E}$)) (*set* $\mathcal{E}$-*impl′*)
using *multigraph′ flow-network-axioms′ u-sum-pos u-non-neg*
by(*auto intro*!: *cost-flow-network.intro flow-network.intro flow-network-axioms.intro split*: *edge-wrapper.split option.split*
simp add: *fst′-def snd′-def*)

lemma *cost-flow-network2*: *cost-flow-network fst′ snd′ create-edge′*
(λ*e*. *case flow-lookup* u-*impl′* *e of None* ⇒ *PInfty* |
Some - ⇒ *case e of old-edge e* ⇒ u *e*
| *new-edge b* ⇒ *sum* u $\mathcal{E}$) (*set* $\mathcal{E}$-*impl′*)
using *multigraph′ flow-network-axioms′ u-sum-pos u-non-neg*
by(*auto intro*!: *cost-flow-network.intro flow-network.intro flow-network-axioms.intro split*: *edge-wrapper.split option.split*
simp add: *fst′-def snd′-def*)

lemma *capacity-cong*:*e* ∈ (*set* $\mathcal{E}$-*impl′*) ⟹
(*case flow-lookup* u-*impl′* *e of None* ⇒ *PInfty* | *Some x* ⇒ *case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$) =
(*case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$)

using *dom-u′-impl in-E-same-cap u-domain*
by(*auto split*: *edge-wrapper.split option.split simp add*: *ℰ-impl′-def Es-are to-set-def*)

lemma *E′-are*: (*ℰ′ s t*) = *set ℰ-impl′*
unfolding *ℰ′-def*[*OF s-in-V t-in-V s-neq-t*]
by(*simp add*: *ℰ-impl′-def Es-are to-set-def*)

lemma *b-impl′-0-cong*: *v* ∈ *dVs* (*make-pair′* ‘ *ℰ′ s t*) ⟹ (*the-default 0* ∘ *bal-lookup* b-*impl′*) *v* = *0*
unfolding *same-Vs*[*OF s-in-V t-in-V s-neq-t*] b-*impl′-def o-apply*
apply(*subst bal-lookup-fold*)
using *bal-map.invar-empty*[*simplified RBT-Set.empty-def bal-empty-def*]
by(*auto simp add*: *vs-def function-generation.vs-def*[*OF function-generation*]
bal-lookup-fold function-generation.es-def[*OF function-generation*]
dVs-eq to-list-def Es-are to-set-def image-Un image-comp the-default-def
selection-functions.make-pair-def make-pair-def bal-lookup-def lookup.simps(*1*))

lemma *capacity-aux-rewrite*:*the-default PInfty* (*flow-lookup* u-*impl′ e*) =(*case flow-lookup* u-*impl′ e of None* ⇒ *PInfty*
| *Some x* ⇒ *case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u *ℰ*)
using *in-E-same-cap dom-u′-impl u-impl′-sum*
by(*fastforce split*: *option.split edge-wrapper.split*
simp add: *Es-are to-set-def ℰ-impl′-def us-are the-default-def*)

context
assumes *no-infty-path*:¬ *has-infty-st-path make-pair ℰ* u *s t*
begin

lemma *no-infinite-cycle*: ¬ *has-neg-infty-cycle make-pair′* (*set ℰ-impl′*) c′ u′
proof(*rule not-has-neg-infty-cycleI*, *goal-cases*)
case (*1 D*)
have *top*: *set D* ⊆ *set ℰ-impl′*
foldr (*λe.* (+) (c′ *e*)) *D 0* < *0*
closed-w (*make-pair′* ‘ *set ℰ-impl′*) (*map make-pair′ D*) (∀ *e*∈*set D*. u′ *e* = ∞)
using *1* **by** *auto*
have *new-edge* (*create-edge t s*) ∈ *set D*
using *top*(*1*,*2*)
by(*induction D*)(*auto simp add*: *ℰ-impl′-def* c′-*def c-lookup′-def*)
then obtain *D1 D2* **where** *D-prop*:*D* = *D1*@[*new-edge* (*create-edge t s*)]@*D2*
new-edge (*create-edge t s*) ∉ *set D1*
by (*metis single-in-append split-list-first*)
then obtain *u* **where** *u-prop*: *awalk* (*make-pair′* ‘ *set ℰ-impl′*) *u*
(*map make-pair′* (*D1*@[*new-edge* (*create-edge t s*)]@*D2*)) *u*
0 < *length* (*map make-pair′* (*D1*@[*new-edge* (*create-edge t s*)]@*D2*))
using *top*(*3*) **by**(*auto simp add*: *closed-w-def*)
hence *awalk-u-t*:*awalk* (*make-pair′* ‘ *set ℰ-impl′*) *u* (*map make-pair′ D1*) *t*

by (*auto simp add*: *awalk-Cons-iff create-edge'*(*1*))
obtain *D21 D22* **where** *D2-prop*:[*new-edge* (*create-edge t s*)]@*D2* = *D21*@[*new-edge* (*create-edge t s*)]@*D22*
new-edge (*create-edge t s*) $\notin$ *set D22*
by (*metis append.left-neutral append-Cons split-list-last*)
hence *awalk-s-u*:*awalk* (*make-pair'* ' *set* $\mathcal{E}$*-impl'*) *s* (*map make-pair' D22*) *u*
using *u-prop*(*1*) **by**(*auto simp add*: *awalk-Cons-iff create-edge'*(*2*))
hence *awalk-s-t*:*awalk* (*make-pair'* ' *set* $\mathcal{E}$*-impl'*) *s* (*map make-pair'* (*D22*@*D1*)) *t*
using *awalk-u-t* **by** *auto*
have *in-E*:*set* (*D22* @ *D1*) $\subseteq$ *old-edge* ' $\mathcal{E}$
proof(*rule*, *goal-cases*)
case (*1 e*)
hence *e* $\in$ *set* $\mathcal{E}$*-impl' e* $\neq$ *new-edge* (*create-edge t s*)
using *D-prop D2-prop top*(*1*) **by** *auto*
thus *?case*
by(*simp add*: *Es-are to-set-def* $\mathcal{E}$*-impl'-def*)
qed
have *same-path*:*map make-pair* (*map get-old-edge* (*D22* @ *D1*)) = *map make-pair'* (*D22*@*D1*)
using *map-make-pair'-is-make-pair-of-get-old-edge*[*OF in-E*] **by** *simp*
have *not-nil*:*D22*@*D1* $\neq$ *Nil*
using *awalk-s-t s-neq-t* **by** *auto*
have *awalk* (*make-pair* ' $\mathcal{E}$) *s* (*map make-pair* (*map get-old-edge* (*D22*@*D1*))) *t*
using *not-nil in-E*
by (*subst same-path*)(*fastforce intro*: *subset-mono-awalk'*[*OF awalk-s-t*] *simp add*: *make-pair*)
moreover have *path-set-in-E*:*set* (*map get-old-edge* (*D22*@*D1*)) $\subseteq \mathcal{E}$
using *in-E* **by** *auto*
moreover have *e* $\in$ *set* (*map get-old-edge* (*D22*@*D1*)) $\Longrightarrow$ u *e* = *PInfty* **for** *e*
proof(*goal-cases*)
case *1*
hence *old-edge e* $\in$ *set D*
using *in-E D-prop*(*1*) *D2-prop*(*1*) **by** *auto*
hence u' (*old-edge e*) = ∞
using *top*(*4*) **by** *auto*
moreover have *e* $\in$ *set* $\mathcal{E}$*-impl*
using *1 Es-are path-set-in-E* **by**(*auto simp add*: *to-set-def*)
ultimately show *?case*
using *in-E-same-cap*[*of e*]
by(*simp add*: u'*-def the-default-def us-are* u*-def Es-are to-set-def comp-def*)
qed
ultimately have *has-infty-st-path local.make-pair* $\mathcal{E}$ u *s t*
using *not-nil*
by(*fastforce intro*!: *has-infty-st-pathI*[*of - - - map get-old-edge* (*D22*@*D1*)])
thus *?case*
using *no-infty-path* **by** *simp*
qed

lemma u′ = (*λe. case flow-lookup* u-*impl′ e of None* ⇒ *PInfty*
| *Some x* ⇒ *case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$)
using u′-*def capacity-aux-rewrite* **by** *auto*

lemma *correctness-of-implementation-success*:
return final-state-maxflow = *success* ⟹ *is-max-flow s t* (*abstract-flow-map final-flow-impl-maxflow-original*)
apply(*rule maxflow-to-mincost-flow-reduction*(*4*)[*OF s-in-V t-in-V s-neq-t - abstract-flows-are*])+
apply(*subst E′-are*)
apply(*rule capacity-Opt-cong*[*OF cost-flow-network2 cost-flow-network1 capacity-cong*], *simp*)
apply(*rule cost-flow-spec.is-Opt-cong*[*OF refl*, *of - - - the-default 0 o bal-lookup* b-*impl′*])
apply(*rule b-impl′-0-cong*)
apply(*simp add*: *E′-are make-pair′-is*(*1*))
unfolding *final-flow-impl-maxflow-def fst′-def2*(*2*) *snd′-def2*(*2*) *final-flow-impl-original-def*
apply(*rule with-capacity-proofs.correctness-of-implementation-success*[*OF with-capacity-proofs*])
using *no-infinite-cycle*
by(*auto simp add*: *final-state-maxflow-def final-state-cap-def* $\mathcal{E}$-*impl′-def fst′-def2*(*1*)

snd′-def2(*1*) c′-*def* c-*impl′-def* u′-*def with-capacity-proofs.cs-are*[*OF with-capacity-proofs*]
with-capacity-proofs.us-are[*OF with-capacity-proofs*]
capacity-aux-rewrite make-pair′-is(*2*))

notation *is-s-t-flow* (- *is* - −− - *flow*)

lemma *correctness-of-implementation-infeasible*:
return final-state-maxflow = *infeasible* ⟹ *False*
proof(*rule ccontr*, *goal-cases*)
case *1*
have *f-prop*: (*λ x. 0*) *is s* −− *t flow*
using *s-in-V t-in-V s-neq-t u-non-neg*
by(*auto simp add*: *is-s-t-flow-def isuflow-def ex-def zero-ereal-def*)
have *no-flow*:∄ *f. flow-network-spec.isbflow fst′ snd′*
(*set* $\mathcal{E}$-*impl′*) (*λe. case flow-lookup* u-*impl′ e of None* ⇒ *PInfty* |
Some x ⇒ *case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$)*f*
(*the-default 0* ∘ *bal-lookup* b-*impl′*)
proof(*rule with-capacity-proofs.correctness-of-implementation-infeasible*[*OF with-capacity-proofs*], *goal-cases*)
case *1*
then show *?case*
using *no-infinite-cycle*
by(*simp add*: c′-*def* c-*impl′-def* u′-*def make-pair′-is*(*1*)
with-capacity-proofs.cs-are[*OF with-capacity-proofs*]
with-capacity-proofs.us-are[*OF with-capacity-proofs*])
next
case *2*
thus *?case*

using *1*(*1*)
by(*simp add*: *final-state-cap-def fst′-def2*(*1*) *local.final-state-maxflow-def snd′-def2*(*1*))
qed
have *a*:*flow-network-spec.isbflow fst′ snd′*
(*set* $\mathcal{E}$*-impl′*) (*λe. case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$)
(*λe. case e of old-edge e* ⇒ (*λ x. 0*) *e* | *new-edge b* ⇒ *ex* (*λ x. 0*) *t*) (*λe. 0*)
using *maxflow-to-mincost-flow-reduction*(*1*)[*OF s-in-V t-in-V s-neq-t f-prop refl*] *E′-are* **by** *auto*
have *b*:*flow-network-spec.isbflow fst′ snd′* (*set* $\mathcal{E}$*-impl′*)
(*λe. case flow-lookup* u*-impl′ e of None* ⇒ *PInfty* |
Some x ⇒ *case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$)
(*λe. case e of old-edge e* ⇒ (*λ x. 0*) *e* | *new-edge b* ⇒ *ex* (*λ x. 0*) *t*) (*λe. 0*)
using *capacity-aux-rewrite capacity-cong cost-flow-network.axioms*[*OF cost-flow-network2*]
cost-flow-network.axioms[*OF cost-flow-network1*]
by (*force intro*: *capacity-bflow-cong*[*OF - - - a*])
have *flow-network-spec.isbflow fst′ snd′* (*set* $\mathcal{E}$*-impl′*)
(*λe. case flow-lookup* u*-impl′ e of None* ⇒ *PInfty*
| *Some x* ⇒ *case e of old-edge e* ⇒ u *e* | *new-edge b* ⇒ *sum* u $\mathcal{E}$)
(*λe. case e of old-edge e* ⇒ (*λ x. 0*) *e* | *new-edge b* ⇒ *ex* (*λ x. 0*) *t*) (*the-default 0* ∘ *bal-lookup* b*-impl′*)
using *b-impl′-0-cong E′-are*
by (*force intro*!: *cost-flow-spec.isbflow-cong*[*OF - - b*] *simp add*: *make-pair′-is*(*1*))
thus *?case*
using *no-flow*
by (*simp add*: *fst′-def snd′-def*)
qed

lemma *correctness-of-implementation-excluded-case*:
return final-state-maxflow = *notyetterm* ⟹ *False*
using *no-infinite-cycle*[*simplified* c*′-def o-apply c-lookup′-def* u*′-def edge-wrapper.case-distrib*[*of the*]
option.sel $\mathcal{E}$*-impl′-def*] *make-pair′-is*(*1*)
no-infinite-cycle
with-capacity-proofs.correctness-of-implementation-excluded-case[*OF with-capacity-proofs*]
with-capacity-proofs.cs-are[*OF with-capacity-proofs*]
with-capacity-proofs.us-are[*OF with-capacity-proofs*]
by (*intro with-capacity-proofs.correctness-of-implementation-excluded-case*[*of snd′* c*-impl′* b*-impl′ c-lookup′ fst′*
create-edge′ $\mathcal{E}$*-impl′* u*-impl′ - - - the-default 0* ∘ *bal-lookup* b*-impl′*])
(*auto simp add*: *final-state-cap-def* c*′-def* c*-impl′-def* u*′-def fst′-def2*(*2*)
final-state-maxflow-def snd′-def2(*2*))

lemmas *correctness-of-implementation* = *correctness-of-implementation-success*
correctness-of-implementation-infeasible
correctness-of-implementation-excluded-case

end
end

value *final-state-maxflow fst snd Pair ℰ-impl* u-*impl 1 3*
value *final-flow-impl-maxflow fst snd Pair ℰ-impl* u-*impl 1 3*
value *final-flow-impl-maxflow-original fst snd Pair ℰ-impl* u-*impl 1 3*
value *inorder* (*final-flow-impl-maxflow-original fst snd Pair ℰ-impl* u-*impl 1 3*)
end

0.3 Characterising the Existence of Optimum Flows

theory *Existence-Optflows*
  imports *Usage-Capacitated*
begin
hide-const *ℰ-impl es* c-*impl* b-*impl* u-*impl* b

locale *cost-flow-network-flow-existence*
= *cost-flow-network*
where *fst* = *fst* **for** *fst*:: (*′edge-type*::*linorder*) ⇒ (*′a*::*linorder*)
begin

lemma *es-exist*: ∃ *es*. *set es* = ℰ ∧ *distinct es*
  using *finite-E*
 by(*induction* ℰ *rule*: *finite-induct*)(*auto intro*: *exI*[*of* - - # -])

definition *ℰ-impl* = (*SOME es*. *set es* = ℰ ∧ *distinct es*)

lemma *ℰ-impl-prop*: *set ℰ-impl* = ℰ *distinct ℰ-impl*
  using *es-exist*[*simplified sym*[*OF some-eq-ex*]]
  by (*auto simp add*: *ℰ-impl-def*)

definition c-*impl* = *cost-dummy*
definition *c-lookup* - *x* = *Some* (c *x*)

lemma *u-impl-exists*: ∃ *u-impl*. *dom* (*flow-lookup u-impl*) = ℰ ∧ (∀ *e* ∈ ℰ. *flow-lookup u-impl e* = *Some* (u *e*))
                              ∧ *flow-invar u-impl*
  using *finite-E*
proof(*induction rule*: *finite-induct*)
  case *empty*
  then show *?case*
   by (*auto intro*: *exI*[*of* - *flow-empty*] *simp add*: *flow-map.invar-empty flow-map.map-empty*)
next
  case (*insert e F*)
  then obtain *u-impl* **where** *u-impl-prop*:*dom* (*flow-lookup u-impl*) = *F* (∀ *e*∈*F*. *flow-lookup u-impl e* = *Some* (u *e*))
                    flow-invar u-impl **by** *auto*
  show *?case*
   using *flow-map.map-update*[*OF u-impl-prop*(*3*)] *u-impl-prop*
   by(*auto intro*!: *exI*[*of* - *flow-update e* (u *e*) *u-impl*] *domI flow-map.invar-update*) *force*+

qed

definition u-*impl* = (*SOME u-impl. dom* (*flow-lookup u-impl*) = $\mathcal{E}$ ∧ (∀ *e* ∈ $\mathcal{E}$. *flow-lookup u-impl e* = *Some* (u *e*))
∧ *flow-invar u-impl*)

lemma u-*impl-props*: *dom* (*flow-lookup* u-*impl*) = $\mathcal{E}$ (∀ *e* ∈ $\mathcal{E}$. *flow-lookup* u-*impl e* = *Some* (u *e*))
flow-invar u-*impl*
using *u-impl-exists*[*simplified sym*[*OF some-eq-ex*]]
by (*auto simp add*: u-*impl-def*)

thm *with-capacity-proofs.correctness-of-implementation*[*of snd - - - fst create-edge - -* $\mathcal{E}$ u c]

lemma *cost-flow-network-impl*:*cost-flow-network fst snd create-edge* (*the-default PInfty* ∘ *flow-lookup* u-*impl*) $\mathcal{E}$
using *cost-flow-network-axioms* u-*impl-props*(*1*,*2*)
by(*force split*: *option.split simp add*: *cost-flow-network-def flow-network-def flow-network-axioms-def the-default-def*
dom-def)

lemmas *cost-flow-network2* = *flow-network-axioms*

lemma *b-impl-exists*: ∃ *b-impl. dom* (*bal-lookup b-impl*) = $\mathcal{V}$ ∧
(∀ *v* ∈ $\mathcal{V}$. *bal-lookup b-impl v* = *Some* (*b v*)) ∧ *bal-invar b-impl*
using $\mathcal{V}$-*finite*
proof(*induction rule*: *finite-induct*)
case *empty*
then show *?case*
by (*auto intro*: *exI*[*of - bal-empty*] *simp add*: *bal-map.invar-empty bal-map.map-empty*)
next
case (*insert u V*)
then obtain *b-impl* **where** *b-impl-prop*:*dom* (*bal-lookup b-impl*) = *V*
(∀ *v* ∈ *V*. *bal-lookup b-impl v* = *Some* (*b v*))
bal-invar b-impl **by** *auto*
show *?case*
using *bal-map.map-update*[*OF b-impl-prop*(*3*)] *b-impl-prop*
by(*auto intro*!: *exI*[*of - bal-update u* (*b u*) *b-impl*] *domI bal-map.invar-update*) *force*+
qed

definition *b-impl b* = (*SOME b-impl. dom* (*bal-lookup b-impl*) = $\mathcal{V}$ ∧
(∀ *v* ∈ $\mathcal{V}$. *bal-lookup b-impl v* = *Some* (*b v*)) ∧ *bal-invar b-impl*)

lemma *b-impl-props*: *dom* (*bal-lookup* (*b-impl b*)) = $\mathcal{V}$ (∀ *v* ∈ $\mathcal{V}$. *bal-lookup* (*b-impl b*) *v* = *Some* (*b v*))
bal-invar (*b-impl b*)
using *b-impl-exists*[*simplified sym*[*OF some-eq-ex*]]

by (*auto simp add*: *b-impl-def*) *force*

lemma *with-capacity-proofs*: *with-capacity-proofs snd* c-*impl* (*b-impl b*) *c-lookup fst create-edge* $\mathcal{E}$*-impl* u-*impl* $\mathcal{E}$
(*the-default PInfty* ∘ *flow-lookup* u-*impl*) c (*the-default 0* ∘ *bal-lookup* (*b-impl b*))
apply(*rule with-capacity-proofs.intro*[*OF cost-flow-network-impl*], *rule with-capacity-proofs-axioms.intro*)
using *b-impl-props*[*of b*]
by (*auto simp add*: $\mathcal{E}$*-impl-prop* u-*impl-props set-invar-def to-set-def c-lookup-def*)

interpretation *algo-locale*: *with-capacity-proofs*
where c-*impl* = c-*impl* **and** b-*impl* = (*b-impl b*)
and *c-lookup* = *c-lookup* **and** *fst* = *fst* **and** *snd* = *snd* **and** *create-edge* = *create-edge*
and $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl* **and** u-*impl* = u-*impl* **and** $\mathcal{E}$ = $\mathcal{E}$
and u = *the-default PInfty* ∘ *flow-lookup* u-*impl* **and** c = c
and b = *the-default 0* ∘ *bal-lookup* (*b-impl b*)
using *with-capacity-proofs* **by** *simp*

lemma *algo-locale-isbflow-def*:*algo-locale.isbflow f b* = *flow-network-spec.isbflow fst snd* $\mathcal{E}$ (*the-default PInfty* ∘ *flow-lookup* u-*impl*) *f b*
by *auto*
thm *algo-locale.correctness-of-implementation*

lemma *existence-of-optimum-flow*:
(∃ *f*. *is-Opt b f*) ⟷ ((∃ *f*. *f is b flow*) ∧ ¬ *has-neg-infty-cycle make-pair* $\mathcal{E}$ c u)
proof(*rule*, *goal-cases*)
case *1*
then obtain *f* **where** *isopt*: *is-Opt b f* **by** *auto*
hence *fbflow*:*f is b flow*
using *is-Opt-def* **by** *blast*
moreover have ¬ *has-neg-infty-cycle make-pair* $\mathcal{E}$ c u
unfolding *has-neg-infty-cycle-def*
proof(*rule nexistsI*, *goal-cases*)
case (*1 D*)
then obtain *u* **where** *u-prop*: *awalk* (*make-pair* ' $\mathcal{E}$) *u* (*map make-pair D*) *u* *0* < *length* (*map make-pair D*)
by (*auto simp add*: *closed-w-def*)
have *rcap*:*0* < *Rcap f* (*set* (*map F D*))
using *1*(*1*)
by (*auto simp add*: *Min-gr-iff Rcap-def*)
have *same-path*:(*map* (*to-vertex-pair* ∘ *F*) *D*) = (*map make-pair D*)
by (*simp add*: *make-pair-def Instantiation.make-pair-def*)
have *fstv-is*: *fstv* (*hd* (*map F D*)) = *u*
using *u-prop*(*2*) *awalk-hd*[*OF u-prop*(*1*)]
by(*cases D*)(*auto simp add*: *make-pair''*)
have *sndv-is*: *sndv* (*last* (*map F D*)) = *u*
using *u-prop*(*2*) *awalk-last*[*OF u-prop*(*1*)]
by(*cases D rule*: *rev-cases*)(*auto simp add*: *make-pair''*)

have *augpath*:*augpath f* (*map F D*)
using *1*(*1*) *u-prop*(*1*,*2*) *rcap*
by(*auto simp add*: *same-path fstv-is sndv-is augpath-def prepath-def closed-w-def intro*: *subset-mono-awalk*)
have *rescost-neg*: *foldr* ($\lambda e.$ (+) ($\mathfrak{c}$ *e*)) (*map F D*) *0* = *foldr* ($\lambda e.$ (+) (c *e*)) *D 0*
by(*induction D*) *auto*
have *D-EE*: *set* (*map F D*) $\subseteq \mathfrak{E}$
using *1*(*1*)
by(*force simp add*: $\mathfrak{E}$*-def*)
obtain *C* **where** *C-prop*:*augcycle f C*
apply(*rule augcycle-from-non-distinct-cycle*[*OF augpath*])
using *D-EE rescost-neg 1*(*1*)
by (*auto simp add*: *fstv-is sndv-is*)
have *rcap2*:*Rcap f* (*set C*) > *0*
using *C-prop augcycle-def augpath-rcap* **by** *blast*
hence *g-gtr-0*:*real-of-ereal* (*min 1* (*Rcap f* (*set C*))) > *0*
by(*cases Rcap f* (*set C*)) (*auto simp add*: *min-def*)
have *g-less-rcap*: *ereal* (*real-of-ereal* (*min 1* (*Rcap f* (*set C*)))) $\leq$ *Rcap f* (*set C*)
using *rcap2* **by**(*cases Rcap f* (*set C*)) (*auto simp add*: *min-def*)
have *in-EE*: *set C* $\subseteq \mathfrak{E}$
using *C-prop augcycle-def* **by** *blast*
have *augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set C*)))) *C is b flow*
using *C-prop 1*(*1*) *g-less-rcap*
by(*auto simp add*: $\mathfrak{E}$*-def zero-ereal-def augcycle-def*
intro!: *aug-cycle-pres-b*[*OF fbflow order.strict-implies-order*[*OF g-gtr-0*]])
moreover have $\mathcal{C}$ (*augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set C*)))) *C*) < $\mathcal{C}$ *f*
using *C-prop 1*(*1*) *in-EE g-gtr-0*
by(*subst cost-change-aug*)(*auto intro*!: *mult-pos-neg simp add*: *augcycle-def* $\mathfrak{C}$*-def*)
ultimately show *?case*
using *isopt* **by**(*auto simp add*: *is-Opt-def*)
qed
ultimately show *?case* **by** *auto*
next
case *2*
then obtain *f* **where** *f-prop*: *f is b flow* **by** *auto*
have *no-neg-cycle*:($\neg$($\exists$ *D*. *closed-w* (*make-pair* ‘ $\mathcal{E}$) (*map make-pair D*) $\wedge$
foldr ($\lambda e.$ (+) (c *e*)) *D 0* < *0* $\wedge$ *set D* $\subseteq \mathcal{E}$ $\wedge$ ($\forall$ *e*$\in$*set D*. u *e* = *PInfty*)))
using *2 has-neg-infty-cycleI* **by** *blast*
have *a1*:*f is* $\lambda x.$ *the-default 0* (*bal-lookup* (*b-impl b*) *x*) *flow*
using *b-impl-props*[*of b*]
by(*auto intro*: *isbflow-cong*[*OF - - f-prop*] *simp add*: *the-default-def make-pair''*)
have *a-flow*:*algo-locale.isbflow f* (*the-default 0* $\circ$ *bal-lookup* (*b-impl b*))
using u*-impl-props a1*
by(*intro capacity-bflow-cong*[*OF cost-flow-network2 algo-locale.flow-network-axioms*])
(*auto simp add*: *make-pair'' comp-def the-default-def*)

have *no-neg-cycle'*:$\nexists$ *D. closed-w* (*make-pair* ‘ $\mathcal{E}$) (*map make-pair D*) $\wedge$
foldr (λ*e.* (+) (c *e*)) *D 0* < *0* $\wedge$
set D $\subseteq$ $\mathcal{E}$ $\wedge$ ($\forall$ *e*$\in$*set D.* (*the-default PInfty* $\circ$ *flow-lookup* u-*impl*) *e* = *PInfty*)
using *no-neg-cycle* u-*impl-props*(*1*,*2*)
by(*force simp add*: *the-default-def*)
hence *no-neg-cycle''*:$\neg$ *has-neg-infty-cycle local.make-pair* $\mathcal{E}$ c (*the-default PInfty* $\circ$ *flow-lookup* u-*impl*)
by(*auto intro*!: *not-has-neg-infty-cycleI*)
have *an-opt*:*algo-locale.is-Opt* (*the-default 0* $\circ$ *bal-lookup* (*b-impl b*))
(*abstract-flow-map* (*with-capacity.final-flow-impl-original fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* (*b-impl b*) *c-lookup*))
using *algo-locale.correctness-of-implementation*[*OF no-neg-cycle''*] *a-flow return.exhaust* **by** *blast*
have *another-opt*:*is-Opt* (*the-default 0* $\circ$ *bal-lookup* (*b-impl b*))
(*abstract-flow-map* (*with-capacity.final-flow-impl-original fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* (*b-impl b*) *c-lookup*))
using *cost-flow-network-axioms* u-*impl-props*
by(*subst comp-def*)
(*force intro*!: *capacity-Opt-cong*[*OF cost-flow-network-impl - - an-opt*, *of* u, *simplified comp-def make-pair''*]
simp add: *the-default-def*)
have *is-Opt b*
(*abstract-flow-map* (*with-capacity.final-flow-impl-original fst snd* $\mathcal{E}$-*impl* c-*impl* u-*impl* (*b-impl b*) *c-lookup*))
using *b-impl-props*(*1*)
by(*auto intro*!: *is-Opt-cong*[*OF refl - another-opt*] *simp add*: *the-default-def b-impl-props*(*2*) *split*: *option.split*)
then show *?case*
by *auto*
qed

end

locale *flow-network-max-flow-existence*
= *flow-network*
where *fst* = *fst* **for** *fst*:: (*'edge-type*::*linorder*) $\Rightarrow$ (*'a*::*linorder*)
begin

context
fixes *s t*
assumes *s-in-V*: $s \in \mathcal{V}$
assumes *t-in-V*: $t \in \mathcal{V}$
assumes *s-neq-t*: $s \neq t$
begin

lemma *es-exist*: $\exists$ *es. set es* = $\mathcal{E}$ $\wedge$ *distinct es*
using *finite-E*
by(*induction* $\mathcal{E}$ *rule*: *finite-induct*)(*auto intro*: *exI*[*of - - # -*])

definition $\mathcal{E}$*-impl* = (*SOME es. set es* = $\mathcal{E}$ ∧ *distinct es*)

lemma $\mathcal{E}$*-impl-prop*: *set* $\mathcal{E}$*-impl* = $\mathcal{E}$ *distinct* $\mathcal{E}$*-impl*
  using *es-exist*[*simplified sym*[*OF some-eq-ex*]]
  by (*auto simp add*: $\mathcal{E}$*-impl-def*)

lemma *u-impl-exists*: ∃ *u-impl. dom* (*flow-lookup u-impl*) = $\mathcal{E}$ ∧ (∀ *e* ∈ $\mathcal{E}$. *flow-lookup u-impl e* = *Some* (u *e*))
∧ *flow-invar u-impl*
  using *finite-E*
proof(*induction rule*: *finite-induct*)
  case *empty*
  then show *?case*
   by (*auto intro*: *exI*[*of - flow-empty*] *simp add*: *flow-map.invar-empty flow-map.map-empty*)
next
  case (*insert e F*)
  then obtain *u-impl* **where** *u-impl-prop*:*dom* (*flow-lookup u-impl*) = *F* (∀ *e*∈*F*. *flow-lookup u-impl e* = *Some* (u *e*))
flow-invar u-impl **by** *auto*
  show *?case*
   using *flow-map.map-update*[*OF u-impl-prop*(*3*)] *u-impl-prop*
   by(*auto intro*!: *exI*[*of - flow-update e* (u *e*) *u-impl*] *domI flow-map.invar-update*) *force+*
qed

definition u*-impl* = (*SOME u-impl. dom* (*flow-lookup u-impl*) = $\mathcal{E}$ ∧ (∀ *e* ∈ $\mathcal{E}$. *flow-lookup u-impl e* = *Some* (u *e*))
∧ *flow-invar u-impl*)

lemma u*-impl-props*: *dom* (*flow-lookup* u*-impl*) = $\mathcal{E}$ (∀ *e* ∈ $\mathcal{E}$. *flow-lookup* u*-impl e* = *Some* (u *e*))
flow-invar u*-impl*
  using *u-impl-exists*[*simplified sym*[*OF some-eq-ex*]]
  by (*auto simp add*: u*-impl-def*)

lemma *flow-network-impl*:*flow-network fst snd create-edge* (*the-default PInfty* ∘ *flow-lookup* u*-impl*) $\mathcal{E}$
 using *flow-network-axioms* u*-impl-props*(*1*,*2*)
 by(*force split*: *option.split simp add*: *flow-network-def flow-network-axioms-def the-default-def dom-def*)

lemma *flow-network2*:*flow-network fst snd create-edge* u $\mathcal{E}$
  using *finite-E*
  by(*auto intro*!: *flow-network.intro multigraph.intro flow-network-axioms.intro*
    simp add: *create-edge′ E-not-empty u-non-neg*)

lemma *b-impl-exists*: ∃ *b-impl. dom* (*bal-lookup b-impl*) = $\mathcal{V}$ ∧
(∀ *v* ∈ $\mathcal{V}$. *bal-lookup b-impl v* = *Some* (*b v*)) ∧ *bal-invar b-impl*
   using $\mathcal{V}$*-finite*

proof(*induction rule*: *finite-induct*)
  case *empty*
  then show *?case*
   by (*auto intro*: *exI*[*of* - *bal-empty*] *simp add*: *bal-map.invar-empty bal-map.map-empty*)
next
  case (*insert u V*)
  then obtain *b-impl* **where** *b-impl-prop*:*dom* (*bal-lookup b-impl*) = *V*
                          ($\forall$ *v* $\in$ *V*. *bal-lookup b-impl v* = *Some* (*b v*))
                           bal-invar b-impl **by** *auto*
  show *?case*
   using *bal-map.map-update*[*OF b-impl-prop*(*3*)] *b-impl-prop*
    by(*auto intro*!: *exI*[*of* - *bal-update u* (*b u*) *b-impl*] *domI bal-map.invar-update*)
force+
qed

definition *b-impl b* = (*SOME b-impl*. *dom* (*bal-lookup b-impl*) = $\mathcal{V}$ $\wedge$
                      ($\forall$ *v* $\in$ $\mathcal{V}$. *bal-lookup b-impl v* = *Some* (*b v*)) $\wedge$ *bal-invar b-impl*)

lemma *b-impl-props*: *dom* (*bal-lookup* (*b-impl b*)) = $\mathcal{V}$ ($\forall$ *v* $\in$ $\mathcal{V}$. *bal-lookup* (*b-impl*
b) *v* = *Some* (*b v*))
                              bal-invar (*b-impl b*)
  using *b-impl-exists*[*simplified sym*[*OF some-eq-ex*]]
  by (*auto simp add*: *b-impl-def*) *force*

lemma *solve-maxflow-proofs*: *solve-maxflow-proofs s t fst snd create-edge* $\mathcal{E}$*-impl*
u-*impl* $\mathcal{E}$ (*the-default PInfty* $\circ$ *flow-lookup* u-*impl*)
 apply(*rule solve-maxflow-proofs.intro*[*OF flow-network-impl*], *rule solve-maxflow-proofs-axioms.intro*)
  by(*auto simp add*: $\mathcal{E}$*-impl-prop* u-*impl-props set-invar-def to-set-def s-in-V t-in-V*
s-neq-t)

interpretation *algo-locale*: *solve-maxflow-proofs*
  where *fst* = *fst* **and** *snd* = *snd* **and** *create-edge* = *create-edge*
  and $\mathcal{E}$*-impl* = $\mathcal{E}$*-impl* **and** u-*impl* = u-*impl* **and** $\mathcal{E}$ = $\mathcal{E}$
  and u = *the-default PInfty* $\circ$ *flow-lookup* u-*impl*
  using *solve-maxflow-proofs* **by** *simp*

lemma *algo-locale-isbflow-def*:*algo-locale.isbflow f b* =
                  flow-network-spec.isbflow fst snd $\mathcal{E}$ (*the-default PInfty* $\circ$ *flow-lookup*
u-*impl*) *f b*
  by *auto*

lemma *to-maxflow-from-algo*: *algo-locale.is-s-t-flow f s t* $\Longrightarrow$ *f is s*−−*t flow*
proof(*goal-cases*)
  case *1*
  hence *all-props*:*algo-locale.isuflow f algo-locale.ex f s* $\leq$ *0*
        s $\in$ $\mathcal{V}$ *t* $\in$ $\mathcal{V}$ *s* $\neq$ *t*
        ($\bigwedge$ *x*. *x*$\in\mathcal{V}$ $\Longrightarrow$ *x* $\neq$ *s* $\Longrightarrow$ *x* $\neq$ *t* $\Longrightarrow$ *algo-locale.ex f x* = *0*)
    using *algo-locale.is-s-t-flow-def*[*of f s t*] **by** *auto*

have *isuflow f*
using *all-props*(*1*) u-*impl-props*(*2*)
by(*subst* (*asm*) *algo-locale.isuflow-def*)(*auto simp add*: *isuflow-def the-default-def*)
moreover have *ex f s* $\leq$ *0*
using *all-props*(*2*) u-*impl-props*(*2*)
by (*auto simp add*: *algo-locale.ex-def delta-minus-def delta-plus-def ex-def delta-plus-def delta-minus-def*)
moreover have ($\bigwedge$ *x*. *x*$\in\mathcal{V}$ $\Longrightarrow$ *x* $\neq$ *s* $\Longrightarrow$ *x* $\neq$ *t* $\Longrightarrow$ *ex f x* = *0*)
using *all-props*(*6*)
by (*auto simp add*: *algo-locale.ex-def delta-minus-def delta-plus-def ex-def delta-plus-def delta-minus-def*)
ultimately show *?case*
using *s-in-V t-in-V s-neq-t* **by**(*auto intro*!: *is-s-t-flowI*)
qed

lemma *to-alog-max-flow*: *f is s*−−*t flow* $\Longrightarrow$ *algo-locale.is-s-t-flow f s t*
proof(*goal-cases*)
case *1*
hence *all-props*:*isuflow f ex f s* $\leq$ *0 s* $\in\mathcal{V}$ *t* $\in\mathcal{V}$ *s* $\neq$ *t*
($\bigwedge$ *x*. *x*$\in\mathcal{V}$ $\Longrightarrow$ *x* $\neq$ *s* $\Longrightarrow$ *x* $\neq$ *t* $\Longrightarrow$*ex f x* = *0*)
by (*auto simp add*: *is-s-t-flow-def*)
have *algo-locale.isuflow f*
using *all-props*(*1*) u-*impl-props*(*2*)
by(*subst algo-locale.isuflow-def*)(*auto simp add*: *isuflow-def the-default-def*)
moreover have *algo-locale.ex f s* $\leq$ *0*
using *all-props*(*2*) u-*impl-props*(*2*)
by (*auto simp add*: *delta-minus-def delta-plus-def ex-def*)
moreover have ($\bigwedge$ *x*. *x*$\in\mathcal{V}$ $\Longrightarrow$ *x* $\neq$ *s* $\Longrightarrow$ *x* $\neq$ *t* $\Longrightarrow$ *algo-locale.ex f x* = *0*)
using *all-props*(*6*)
by (*auto simp add*: *ex-def delta-plus-def delta-minus-def*)
ultimately show *?case*
using *s-in-V t-in-V s-neq-t flow-network-impl*
by(*auto intro*!: *flow-network-spec.is-s-t-flowI*)
qed
term (λ*x*. (*if* (*x* = *s*) *then 0 else 0*))
lemma *existence-of-maximum-flow*:
($\exists$ *f*. *is-max-flow s t f*) $\longleftrightarrow$ $\neg$ *has-infty-st-path make-pair* $\mathcal{E}$ u *s t*
proof(*rule*, *goal-cases*)
case *1*
then obtain *f* **where** *isopt*: *is-max-flow s t f* **by** *auto*
define *b* **where** *b* = (λ*x*. (*if* (*x* = *s*) *then* (*ex f t*) *else* (*if* (*x* = *t*) *then* (− *ex f t*) *else 0*)))
hence *fbflow*:*f is s*−−*t flow f is b flow*
using *isopt is-max-flow-def s-t-flow-is-ex-bflow* **by** *blast*+
moreover have $\neg$*has-infty-st-path make-pair* $\mathcal{E}$ u *s t*
proof(*rule not-has-infty-st-pathI*, *goal-cases*)
case (*1 D*)
hence *u-prop*: *awalk UNIV s* (*map make-pair D*) *t set D* $\subseteq\mathcal{E}$ ($\forall$ *e*$\in$*set D*. u *e* = *PInfty*)

and *Dlen*: *length D* > *0*
using *s-neq-t* **by**(*auto simp add*: *awalk-def*)
have *rcap*:*0* < *Rcap f* (*set* (*map F D*))
using *1*(*4*)
by (*auto simp add*: *Rcap-def*)
have *same-path*:(*map* (*to-vertex-pair* ∘ *F*) *D*) = (*map make-pair D*)
by (*simp add*: *make-pair-def Instantiation.make-pair-def*)
have *fstv-is*: *fstv* (*hd* (*map F D*)) = *s*
using *Dlen awalk-hd*[*OF u-prop*(*1*)]
by(*cases D*)(*auto simp add*: *make-pair''*)
have *sndv-is*: *sndv* (*last* (*map F D*)) = *t*
using *Dlen awalk-last*[*OF u-prop*(*1*)]
by(*cases D rule*: *rev-cases*)(*auto simp add*: *make-pair''*)
have *augpath*:*augpath f* (*map F D*) *prepath* (*map F D*)
using *1*(*1*) *u-prop*(*1*) *Dlen rcap*
by(*auto simp add*: *same-path fstv-is sndv-is augpath-def prepath-def closed-w-def intro*: *subset-mono-awalk*)
have *D-EE*: *set* (*map F D*) ⊆ 𝔈
using *1*(*3*)
by(*force simp add*: 𝔈-*def*)
obtain *ds* **where** *ds-prop*:*prepath ds distinct ds set ds* ⊆ *set* (*map F D*) *fstv* (*hd* (*map F D*)) = *fstv* (*hd ds*)
sndv (*last* (*map F D*)) = *sndv* (*last ds*) *ds* ≠ []
apply(*cases distinct* (*map F D*))
subgoal
using *D-EE augpath*(*2*) **unfolding** *prepath-def* **by** *blast*
by(*auto intro*: *prepath-drop-cycles*[*OF augpath*(*2*)])
have *rcap2*:*Rcap f* (*set ds*) > *0*
using *ds-prop*(*3*) *u-prop*(*3*) **by**(*auto simp add*: *Rcap-def*)
hence *g-gtr-0*:*real-of-ereal* (*min 1* (*Rcap f* (*set ds*))) > *0*
by(*cases Rcap f* (*set ds*)) (*auto simp add*: *min-def*)
have *augpath-ds*: *augpath f ds*
using *ds-prop*(*1*) *rcap2* **by** (*auto simp add*: *augpath-def*)
have *g-less-rcap*: *ereal* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) ≤ *Rcap f* (*set ds*)
using *rcap2* **by**(*cases Rcap f* (*set ds*)) (*auto simp add*: *min-def*)
have *after-augment*: *augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds*
is λ*v*. *if v* = *fstv* (*hd ds*) *then b v* + *real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))
else if v = *sndv* (*last ds*) *then b v* − *real-of-ereal* (*min 1* (*Rcap f* (*set ds*))) *else b v flow*
using *augpath-ds g-less-rcap ds-prop D-EE fstv-is sndv-is s-neq-t g-gtr-0 fbflow*(*2*)
by(*auto intro*!: *augment-path-validness-b-pres-source-target-distinct*)
have *augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds is s* −− *t flow*
proof(*rule is-s-t-flowI*[*OF - - s-in-V t-in-V s-neq-t*], *goal-cases*)
case *1*
then show *?case*
using *after-augment isbflow-def* **by** *blast*

next
case *2*
have *ex* (*augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds*) *s* =
− (*b s* + (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))))
using *after-augment s-in-V ds-prop*(*4*) *fstv-is*
by(*fastforce simp add*: *isbflow-def*)
also have ... ≤ − *b s*
using *g-gtr-0* **by** *argo*
also have ... = *ex f s*
using *b-def fbflow*(*1*) *s-t-flow-excess-s-t* **by** *force*
also have ... ≤ *0*
using *fbflow*(*1*)
by(*simp add*: *is-s-t-flow-def*)
finally show *?case* **by** *simp*
next
case (*3 x*)
have *ex* (*augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds*) *x* =
ex f x
using *after-augment 3 t-in-V ds-prop*(*4*,*5*) *sndv-is s-neq-t fstv-is fbflow*(*2*)
by(*fastforce simp add*: *isbflow-def*)
moreover have ... = *0*
using *3*(*1*) *3*(*2*) *3*(*3*) *fbflow*(*1*) *is-s-t-flow-def* **by** *blast*
ultimately show *?case* **by** *simp*
qed
moreover have *ex* (*augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds*) *t*
> *ex f t*
proof−
have *ex* (*augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds*) *t* =
− (*b t* − (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))))
using *after-augment s-in-V ds-prop*(*4*,*5*) *fstv-is s-neq-t sndv-is t-in-V* **by** (*auto simp add*: *isbflow-def*)
moreover have ... > − *b t*
using *g-gtr-0* **by** *argo*
moreover have − *b t* = *ex f t*
using *b-def fbflow*(*1*) *s-t-flow-excess-s-t* **by** *force*
ultimately show *ex* (*augment-edges f* (*real-of-ereal* (*min 1* (*Rcap f* (*set ds*)))) *ds*) *t* > *ex f t*
by *simp*
qed
ultimately show *?case*
using *isopt* **by**(*auto simp add*: *is-max-flow-def*)
qed
thus *?case* **by** *simp*
next
case *2*
hence *two′*: ¬ *has-infty-st-path local.make-pair* $\mathcal{E}$ (*the-default PInfty* ∘ *flow-lookup* u-*impl*) *s t*
using u-*impl-props*(*2*)

by(*force intro*!: *not-has-infty-st-pathI elim*!: *not-has-infty-st-pathE simp add*: *the-default-def*)
have *success*:*return* (*solve-maxflow.final-state-maxflow fst snd create-edge* $\mathcal{E}$*-impl* u-*impl s t*) = *success*
using *algo-locale.correctness-of-implementation*(*2*,*3*)[*OF two'*] *return.exhaust*
by *blast*
have *max-flow-algo*:*algo-locale.is-max-flow s t*
(*abstract-flow-map* (*solve-maxflow.final-flow-impl-maxflow-original fst snd create-edge* $\mathcal{E}$*-impl* u-*impl s t*))
using *algo-locale.correctness-of-implementation*(*1*)[*OF two' success*] **by** *simp*
have *is-max-flow s t*
(*abstract-flow-map* (*solve-maxflow.final-flow-impl-maxflow-original fst snd create-edge* $\mathcal{E}$*-impl* u-*impl s t*))
using *max-flow-algo to-alog-max-flow*
to-maxflow-from-algo
by(*auto elim*!: *flow-network-spec.is-max-flowE intro*!: *is-max-flowI*)
thus *?case* **by** *auto*
qed
end
end
end