graph-verification

By christine

April 14, 2015

Contents

theory *Connected-Components*
imports *../Graph-Theory/Graph-Theory*
begin

locale *connected-components-locale* =
  fin-digraph +
  fixes *num* :: $'a \Rightarrow nat$
  fixes *parent-edge* :: $'a \Rightarrow 'b$ *option*
  fixes *r* :: $'a$
  assumes *r-assms*: $r \in verts\ G \wedge parent\text{-}edge\ r = None \wedge num\ r = 0$
  assumes *parent-num-assms*:
    $\bigwedge v.\ v \in verts\ G \wedge v \neq r \Longrightarrow$
      $\exists\, e \in arcs\ G.$
        $parent\text{-}edge\ v = Some\ e\ \wedge$
        $head\ G\ e = v\ \wedge$
        $num\ v =\ num\ (tail\ G\ e) + 1$

sublocale *connected-components-locale* $\subseteq$ *fin-digraph G*
  by *auto*

context *connected-components-locale*
begin

lemma *ccl-wellformed*: *wf-digraph G*
  by *unfold-locales*

lemma *num-r-is-min*:
  assumes $v \in verts\ G$
  assumes $v \neq r$
  shows $num\ v > 0$
  using *parent-num-assms assms*
  by *fastforce*

lemma *path-from-root*:
  fixes *v* :: $'a$
  assumes $v \in verts\ G$

shows $r \rightarrow^* v$
using *assms*
proof (*induct num v arbitrary*: *v*)
case *0*
hence $v = r$ **using** *num-r-is-min* **by** *fastforce*
with ‹$v \in$ *verts G*› **show** *?case* **by** *auto*
next
case (*Suc n'*)
hence $v \neq r$ **using** *r-assms* **by** *auto*
then obtain *e* **where** *ee*:
$e \in$ *arcs G*
head G e $= v \wedge$ *num v* $=$ *num* (*tail G e*) $+ 1$
using *Suc parent-num-assms* **by** *blast*
with ‹$v \in$ *verts G*› *Suc*(*1*,*2*) *tail-in-verts*
have $r \rightarrow^*$ (*tail G e*) *tail G e* $\rightarrow v$
by (*auto intro*: *in-arcs-imp-in-arcs-ends*)
then show *?case* **by** (*rule reachable-adj-trans*)
qed

The underlying undirected, simple graph is connected

lemma *connectedG*: *connected G*
proof (*unfold connected-def*, *intro strongly-connectedI*)
show *verts* (*with-proj* (*mk-symmetric G*)) $\neq \{\}$
by (*metis equals0D r-assms reachable-in-vertsE reachable-mk-symmetricI reachable-refl*)
next
let *?SG* = *mk-symmetric G*
interpret *S*: *pair-fin-digraph ?SG* **..**
fix *u v* **assume** *uv-sG*: $u \in$ *verts ?SG* $v \in$ *verts ?SG*
from *uv-sG* **have** $u \in$ *verts G* $v \in$ *verts G* **by** *auto*
then have $u \rightarrow^*_{?SG} r$ $r \rightarrow^*_{?SG} v$
by (*auto intro*: *reachable-mk-symmetricI path-from-root symmetric-reachable symmetric-mk-symmetric simp del*: *pverts-mk-symmetric*)
then show $u \rightarrow^*_{?SG} v$
by (*rule S.reachable-trans*)
qed

theorem *connected-by-path*:
fixes *u v* :: *'a*
assumes $u \in$ *pverts* (*mk-symmetric G*)
assumes $v \in$ *pverts* (*mk-symmetric G*)
shows $u \rightarrow^*_{mk\text{-}symmetric\ G} v$
using *connectedG wellformed-mk-symmetric assms*
unfolding *connected-def strongly-connected-def* **by** *fastforce*
end

corollary (**in** *connected-components-locale*) *connected-graph*:
assumes $u \in$ *verts G* **and** $v \in$ *verts G*
shows $\exists p.$ *vpath p* (*mk-symmetric G*) $\wedge$ *hd p* $= u \wedge$ *last p* $= v$

proof −
  interpret *S*: *pair-fin-digraph mk-symmetric G* **..**
  show *?thesis* **unfolding** *S.reachable-vpath-conv*[*symmetric*]
    using *assms* **by** (*auto intro*: *connected-by-path*)
qed

end
theory *Shortest-Path-Theory*
imports
  Complex
  ../Graph-Theory/Graph-Theory
begin

locale *basic-sp* =
  fin-digraph +
  fixes *dist* :: $'a \Rightarrow$ *ereal*
  fixes *c* :: $'b \Rightarrow$ *real*
  fixes *s* :: $'a$
  assumes *general-source-val*: $dist\ s \leq 0$
  assumes *trian*:
    $\bigwedge e.\ e \in arcs\ G \Longrightarrow$
      $dist\ (head\ G\ e) \leq dist\ (tail\ G\ e) + c\ e$

locale *basic-just-sp* =
  basic-sp +
  fixes *enum* :: $'a \Rightarrow$ *enat*
  assumes *just*:
    $\bigwedge v.\ [\![v \in verts\ G;\ v \neq s;\ enum\ v \neq \infty]\!] \Longrightarrow$
      $\exists\ e \in arcs\ G.\ v = head\ G\ e\ \wedge$
        $dist\ v = dist\ (tail\ G\ e) + c\ e\ \ \wedge$
        $enum\ v = enum\ (tail\ G\ e) + (enat\ 1)$

locale *shortest-path-non-neg-cost* =
  basic-just-sp +
  assumes *s-in-G*: $s \in verts\ G$
  assumes *source-val*: $dist\ s = 0$
  assumes *no-path*: $\bigwedge v.\ v \in verts\ G \Longrightarrow dist\ v = \infty \longleftrightarrow enum\ v = \infty$
  assumes *non-neg-cost*: $\bigwedge e.\ e \in arcs\ G \Longrightarrow 0 \leq c\ e$

locale *basic-just-sp-pred* =
  basic-sp +
  fixes *enum* :: $'a \Rightarrow$ *enat*
  fixes *pred* :: $'a \Rightarrow 'b$ *option*
  assumes *just*:
    $\bigwedge v.\ [\![v \in verts\ G;\ v \neq s;\ enum\ v \neq \infty]\!] \Longrightarrow$
      $\exists\ e \in arcs\ G.$
        $e = the\ (pred\ v)\ \wedge$
        $v = head\ G\ e\ \wedge$

```
      dist v = dist (tail G e) + c e  ∧
      enum v = enum (tail G e) + (enat 1)

sublocale basic-just-sp-pred ⊆ basic-just-sp
using basic-just-sp-pred-axioms
unfolding basic-just-sp-pred-def
  basic-just-sp-pred-axioms-def
by unfold-locales (blast)

locale shortest-path-non-neg-cost-pred =
  basic-just-sp-pred +
  assumes s-in-G: s ∈ verts G
  assumes source-val: dist s = 0
  assumes no-path: ⋀v. v ∈ verts G ⟹ dist v = ∞ ⟷ enum v = ∞
  assumes non-neg-cost: ⋀e. e ∈ arcs G ⟹ 0 ≤ c e

sublocale shortest-path-non-neg-cost-pred ⊆ shortest-path-non-neg-cost
using shortest-path-non-neg-cost-pred-axioms
by unfold-locales
  (auto simp: shortest-path-non-neg-cost-pred-def
  shortest-path-non-neg-cost-pred-axioms-def)

lemma tail-value-helper:
  assumes hd p = last p
  assumes distinct p
  assumes p ≠ []
  shows p = [hd p]
by (metis assms distinct.simps(2) append-butlast-last-id hd-append
  append-self-conv2 distinct-butlast hd-in-set not-distinct-conv-prefix)

lemma (in basic-sp) dist-le-cost:
  fixes v :: 'a
  fixes p :: 'b list
  assumes awalk s p v
  shows dist v ≤ awalk-cost c p
  using assms
  proof (induct length p arbitrary: p v)
  case 0
    hence s = v by auto
    thus ?case using 0(1) general-source-val
      by (metis awalk-cost-Nil length-0-conv zero-ereal-def)
  next
  case (Suc n)
    then obtain p' e where p'e: p = p' @ [e]
      by (cases p rule: rev-cases) auto
    then obtain u where ewu: awalk s p' u ∧ awalk u [e] v
      using awalk-append-iff Suc(3) by simp
    then have du: dist u ≤ ereal (awalk-cost c p')
      using Suc p'e by simp
```

from *ewu* **have** *ust*: $u = tail\ G\ e$ **and** *vta*: $v = head\ G\ e$
  by *auto*
then have $dist\ v \leq dist\ u + c\ e$
  using *ewu du ust trian*[**where** $e{=}e$] **by** *force*
with *du* **have** $dist\ v \leq ereal\ (awalk\text{-}cost\ c\ p') + c\ e$
  by (*metis add-right-mono order-trans*)
thus $dist\ v \leq awalk\text{-}cost\ c\ p$
  using *awalk-cost-append* $p'e$ **by** *simp*
qed

lemma (**in** *fin-digraph*) *witness-path*:
assumes $\mu\ c\ s\ v = ereal\ r$
shows $\exists\ p.\ apath\ s\ p\ v \wedge \mu\ c\ s\ v = awalk\text{-}cost\ c\ p$
proof −
have *sv*: $s \rightarrow^* v$
  using *shortest-path-inf*[*of s v c*] *assms* **by** *fastforce*
{
  fix *p* **assume** *awalk s p v*
  then have *no-neg-cyc*:
  $\neg\ (\exists\, w\ q.\ awalk\ w\ q\ w \wedge w \in set\ (awalk\text{-}verts\ s\ p) \wedge awalk\text{-}cost\ c\ q < 0)$
    using *neg-cycle-imp-inf-*μ *assms* **by** *force*
}
thus *?thesis* **using** *no-neg-cyc-reach-imp-path*[*OF sv*] **by** *presburger*
qed

lemma (**in** *basic-sp*) *dist-le-*μ:
fixes $v :: 'a$
assumes $v \in verts\ G$
shows $dist\ v \leq \mu\ c\ s\ v$
proof (*rule ccontr*)
assume *nt*: $\neg$ *?thesis*
show *False*
proof (*cases* $\mu\ c\ s\ v$)
  show $\bigwedge r.\ \mu\ c\ s\ v = ereal\ r \Longrightarrow False$
  proof −
    fix *r* **assume** *r-asm*: $\mu\ c\ s\ v = ereal\ r$
    hence *sv*: $s \rightarrow^* v$
      using *shortest-path-inf*[**where** $u{=}s$ **and** $v{=}v$ **and** $f{=}c$] **by** *auto*
    obtain *p* **where**
      awalk s p v
      $\mu\ c\ s\ v = awalk\text{-}cost\ c\ p$
      using *witness-path*[*OF r-asm*] **unfolding** *apath-def* **by** *force*
    thus *False* **using** *nt dist-le-cost* **by** *simp*
  qed
next
  show $\mu\ c\ s\ v = \infty \Longrightarrow False$ **using** *nt* **by** *simp*
next
  show $\mu\ c\ s\ v = -\infty \Longrightarrow False$
  proof −

```
    assume asm: μ c s v = − ∞
    let ?C = (λx. ereal (awalk-cost c x)) ‘ {p. awalk s p v}
    have ∃ x∈ ?C. x < dist v
      using Inf-ereal-iff [where y =dist vand X=?C and z= −∞]
      nt asm unfolding μ-def INF-def by simp
    then obtain p where
      awalk s p v
      awalk-cost c p < dist v
      by force
    thus False using dist-le-cost by force
   qed
  qed
qed

lemma (in basic-just-sp) dist-ge-μ:
  fixes v :: 'a
  assumes v ∈ verts G
  assumes enum v ≠ ∞
  assumes dist v ≠ −∞
  assumes μ c s s = ereal 0
  assumes dist s = 0
  assumes ⋀u. u∈verts G ⟹ u≠s ⟹ enum u ≠ enat 0
  shows dist v ≥ μ c s v
proof −
  obtain n where enat n = enum v using assms(2) by force
  thus ?thesis using assms
  proof(induct n arbitrary: v)
  case 0 thus ?case by (cases v=s, auto)
  next
  case (Suc n)
   thus ?case
   proof (cases v=s)
   case False
    obtain e where e-assms:
      e ∈ arcs G
      v = head G e
      dist v = dist (tail G e) + ereal (c e)
      enum v = enum (tail G e) + enat 1
      using just[OF Suc(3) False Suc(4)] by blast
    then have nsinf:enum (tail G e) ≠ ∞
      by (metis Suc(2) enat.simps(3) enat-1 plus-enat-simps(2))
    then have ns:enat n = enum (tail G e)
      using e-assms(4) Suc(2) by force
    have  ds: dist (tail G e) = μ c s (tail G e)
      using Suc(1)[OF ns tail-in-verts[OF e-assms(1)] nsinf]
      Suc(5−8) e-assms(3) dist-le-μ[OF tail-in-verts[OF e-assms(1)]]
      by simp
    have dmuc:dist v = μ c s (tail G e) + ereal (c e)
      using e-assms(3) ds  by auto
```

    thus *?thesis*
    proof (*cases dist v* = ∞)
    case *False*
      have *arc-to-ends G e* = (*tail G e*, *v*)
        unfolding *arc-to-ends-def*
        by (*simp add*: *e-assms*(*2*))
      obtain *r* **where** *μr*: *μ c s* (*tail G e*) = *ereal r*
         using *e-assms*(*3*) *Suc*(*5*) *ds False*
         by (*cases μ c s* (*tail G e*), *auto*)
      obtain *p* **where**
        awalk s p (*tail G e*) **and**
        μs: *μ c s* (*tail G e*) = *ereal* (*awalk-cost c p*)
        using *witness-path*[*OF μr*] **unfolding** *apath-def*
        by *blast*
      then have *pe*: *awalk s* (*p* @ [*e*]) *v*
        using *e-assms*(*1*,*2*) **by** (*auto simp*: *awalk-simps awlast-of-awalk*)
      hence *muc*:*μ c s v* ≤ *μ c s* (*tail G e*) + *ereal* (*c e*)
      using *μs min-cost-le-walk-cost*[*OF pe*] **by** *simp*
      thus *dist v* ≥ *μ c s v* **using** *dmuc* **by** *simp*
    qed *simp*
  qed (*simp add*: *Suc*(*6*,*7*))
 qed
qed

lemma (**in** *shortest-path-non-neg-cost*) *tail-value-check*:
 fixes *u* :: *'a*
 assumes *s* ∈ *verts G*
 shows *μ c s s* = *ereal 0*
proof −
 have ∗: *awalk s* [] *s* **using** *assms* **unfolding** *awalk-def* **by** *simp*
 hence *μ c s s* ≤ *ereal 0* **using** *min-cost-le-walk-cost*[*OF* ∗] **by** *simp*
 moreover
 have (⋀*p*. *awalk s p s* ⟹ *ereal*(*awalk-cost c p*) ≥ *ereal 0*)
  using *non-neg-cost pos-cost-pos-awalk-cost* **by** *auto*
 hence *μ c s s* ≥ *ereal 0*
   unfolding *μ-def* **by** (*blast intro*: *INF-greatest*)
 ultimately
 show *?thesis* **by** *simp*
qed

lemma (**in** *shortest-path-non-neg-cost*) *enum-not0*:
 fixes *v* :: *'a*
 assumes *v* ∈ *verts G*
 assumes *v* ≠ *s*

 shows *enum v* ≠ *enat 0*
proof (*cases enum v* ≠ ∞)
case *True*
 then obtain *ku* **where** *enum v* = *ku* + *enat 1*

```
    using assms just by blast
  thus ?thesis  by (induct ku) auto
qed fast

lemma (in shortest-path-non-neg-cost) dist-ne-ninf:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v ≠ −∞
proof (cases enum v = ∞)
case False
  obtain n where enat n = enum v
    using False by force
  thus ?thesis using assms False
  proof(induct n arbitrary: v)
  case 0 thus ?case
    using enum-not0 source-val by (cases v=s, auto)
  next
  case (Suc n)
    thus ?case
    proof (cases v=s)
    case True
      thus ?thesis using source-val by simp
    next
    case False
      obtain e where e-assms:
        e ∈ arcs G
        dist v = dist (tail G e) + ereal (c e)
        enum v = enum (tail G e) + enat 1
        using just[OF Suc(3) False Suc(4)] by blast
      then have nsinf:enum (tail G e) ≠ ∞
        by (metis Suc(2) enat.simps(3) enat-1 plus-enat-simps(2))
      then have ns:enat n = enum (tail G e)
        using e-assms(3) Suc(2) by force
      have dist (tail G e ) ≠ − ∞
        by (rule Suc(1) [OF ns tail-in-verts[OF e-assms(1)] nsinf])
      thus ?thesis using e-assms(2) by simp
    qed
  qed
next
case True
  thus ?thesis using no-path[OF assms] by simp
qed

theorem (in shortest-path-non-neg-cost) correct-shortest-path:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v = μ c s v
  using no-path[OF assms(1)] dist-le-μ[OF assms(1)]
    dist-ge-μ[OF assms(1) - dist-ne-ninf[OF assms(1)]
```

tail-value-check[*OF s-in-G*] *source-val enum-not0*]
by *fastforce*

corollary (**in** *shortest-path-non-neg-cost-pred*) *correct-shortest-path-pred*:
fixes $v :: 'a$
assumes $v \in verts\ G$
shows $dist\ v = \mu\ c\ s\ v$
using *correct-shortest-path assms* **by** *simp*

end
theory *Shortest-Path-Arbitrary-Edge-Costs*

imports
../Graph-Theory/Graph-Theory
Shortest-Path-Theory
begin

locale *shortest-paths-init* =
fixes $G :: ('a, 'b)$ *pre-digraph* (**structure**)
fixes $s :: 'a$
fixes $c :: 'b \Rightarrow real$
fixes $num :: 'a \Rightarrow nat$
fixes $parent\text{-}edge :: 'a \Rightarrow 'b\ option$
fixes $dist :: 'a \Rightarrow ereal$
assumes *graphG*: *fin-digraph G*

abbreviation (**in** *shortest-paths-init*) $V_f :: 'a\ set$ **where**
$V_f \equiv \{v.\ v \in verts\ G \wedge (\exists r.\ dist\ v = ereal\ r)\}$

abbreviation (**in** *shortest-paths-init*) $V_p :: 'a\ set$ **where**
$V_p \equiv \{v.\ v \in verts\ G \wedge dist\ v = \infty\}$

abbreviation (**in** *shortest-paths-init*) $V_n :: 'a\ set$ **where**
$V_n \equiv \{v.\ v \in verts\ G \wedge dist\ v = -\infty\}$

locale *shortest-paths-reachable* =
shortest-paths-init +
assumes *s-assms*:
$s \in verts\ G$

$num\ s = 0$
assumes *pna*:
$\bigwedge v.\ [\![v \in verts\ G;\ v \neq s;\ v \notin V_p]\!] \Longrightarrow$
$(\exists e \in arcs\ G.\ parent\text{-}edge\ v = Some\ e \wedge$
$head\ G\ e = v \wedge tail\ G\ e \notin V_p \wedge$
$num\ v = num\ (tail\ G\ e) + 1)$

sublocale *shortest-paths-reachable* $\subseteq$ *fin-digraph G*
  using *graphG* **by** *auto*

definition (**in** *shortest-paths-reachable*) *enum* :: $'a \Rightarrow$ *enat* **where**
  enum v = (*if* (*dist v* = ∞ $\vee$ *dist v* = $-\infty$) *then* ∞ *else num v*)

locale *shortest-paths-basic* =
  shortest-paths-reachable +
  basic-just-sp G dist c s enum +
  assumes *source-val*: ($\exists$ *v* $\in$ *verts G*. *enum v* $\neq \infty$) $\Longrightarrow$ *dist s* = *0*

function (**in** *shortest-paths-reachable*) *pwalk* :: $'a \Rightarrow$ $'b$ *list*
where
  pwalk v =
    (*if* (*v* = *s* $\vee$ *dist v* = ∞ $\vee$ *v* $\notin$ *verts G*)
      then []
      else pwalk (*tail G* (*the* (*parent-edge v*))) @ [*the* (*parent-edge v*)]
)
by *auto*
termination (**in** *shortest-paths-reachable*)
  using *pna*
  by (*relation measure num*, *auto*, *fastforce*)

lemma (**in** *shortest-paths-reachable*) *pwalk-simps*:
  v = *s* $\vee$ *dist v* = ∞ $\vee$ *v* $\notin$ *verts G* $\Longrightarrow$ *pwalk v* = []
  v $\neq$ *s* $\Longrightarrow$ *dist v* $\neq \infty$ $\Longrightarrow$ *v* $\in$ *verts G* $\Longrightarrow$
    pwalk v = *pwalk* (*tail G* (*the* (*parent-edge v*))) @ [*the* (*parent-edge v*)]
by *auto*

definition (**in** *shortest-paths-reachable*) *pwalk-verts* :: $'a \Rightarrow$ $'a$ *set* **where**
  pwalk-verts v = {*u*. *u* $\in$ *set* (*awalk-verts s* (*pwalk v*))}

locale *shortest-paths-neg-cyc* =
  shortest-paths-basic +
  fixes *C* :: ($'a$ $\times$($'b$ *awalk*)) *set*
  assumes *C-se*:
    C $\subseteq$ {(*u*, *p*). *dist u* $\neq \infty$ $\wedge$ *awalk u p u* $\wedge$ *awalk-cost c p* < *0*}
  assumes *int-neg-cyc*:
    $\bigwedge v.\ v \in V_n \Longrightarrow$
      (*fst* ‘ *C*) $\cap$ *pwalk-verts v* $\neq$ {}

locale *shortest-paths-basic-pred* =
  shortest-paths-reachable +
  fixes *pred* :: $'a \Rightarrow$ $'b$ *option*
  assumes *bj*: *basic-just-sp-pred G dist c s enum pred*
  assumes *source-val*: ($\exists$ *v* $\in$ *verts G*. *enum v* $\neq \infty$) $\Longrightarrow$ *dist s* = *0*

```
sublocale shortest-paths-basic-pred ⊆ shortest-paths-basic
  using shortest-paths-basic-pred-axioms
  unfolding shortest-paths-basic-pred-def shortest-paths-basic-pred-axioms-def
  shortest-paths-basic-def shortest-paths-basic-axioms-def
  basic-just-sp-pred-def basic-just-sp-pred-axioms-def
  basic-just-sp-def basic-just-sp-axioms-def
  by blast

lemma (in shortest-paths-reachable) num-s-is-min:
  assumes v ∈ verts G
  assumes v ≠ s
  assumes v ∉ V_p
  shows num v > 0
     using pna[OF assms] by fastforce


theorem (in shortest-paths-reachable) path-from-root-Vr-ex:
  fixes v :: 'a
  assumes v ∈ verts G
  assumes v ≠ s
  assumes v ∉ V_p
  shows  ∃ e. s →* tail G e ∧
          e ∈ arcs G ∧ head G e = v ∧ dist (tail G e) ≠ ∞ ∧
          parent-edge v = Some e ∧ num v = num (tail G e) + 1
using assms
proof(induct num v − 1 arbitrary : v)
case 0
  obtain e where ee:
    e ∈ arcs G
    head G e = v
    (tail G e) ∉ V_p
    parent-edge v = Some e
    num v = num (tail G e) + 1
    using pna[OF 0(2−4)]  by fast
  have tail G e = s
    using num-s-is-min[OF tail-in-verts [OF ee(1)] - ee(3)]
    ee(5) 0(1) by auto
  then show ?case using ee by auto
next
case (Suc n')
  obtain e where ee:
    e ∈ arcs G
    head G e = v
    (tail G e) ∉ V_p
    parent-edge v = Some e
    num v = num (tail G e) + 1
```

using *pna*[*OF Suc*(*3*−*5*)] **by** *fast*
then have *ss*: *tail G e* $\neq$ *s*
using *num-s-is-min tail-in-verts ee*
Suc(*2*) *s-assms*(*2*) **by** *force*
have *nst*: $n' = num\ (tail\ G\ e) - 1$
using *ee*(*5*) *Suc*(*2*) **by** *presburger*
obtain e' **where**
reach: $s \to^* tail\ G\ e'$ **and**
e': $e' \in arcs\ G \wedge head\ G\ e' = tail\ G\ e \wedge (tail\ G\ e') \notin V_p$
using *Suc*(*1*)[*OF nst tail-in-verts*[*OF ee*(*1*)] *ss ee*(*3*)] **by** *blast*
from *reach* **also have** $tail\ G\ e' \to tail\ G\ e$ **using** e'
by (*metis in-arcs-imp-in-arcs-ends*)
finally show *?case* **using** e' *ee* **by** *auto*
qed

corollary (**in** *shortest-paths-reachable*) *path-from-root-Vr*:
fixes $v :: 'a$
assumes $v \in verts\ G$
assumes $v \notin V_p$
shows $s \to^* v$
proof(*cases* $v = s$)
case *True* **thus** *?thesis* **using** *assms* **by** *simp*
next
case *False*
obtain *e* **where** $s \to^* tail\ G\ e$ **and** $e \in arcs\ G$ **and** $head\ G\ e = v$
using *path-from-root-Vr-ex*[*OF assms*(*1*) *False assms*(*2*)] **by** *blast*
then have $s \to^* tail\ G\ e$ **and** $tail\ G\ e \to v$
by (*auto intro*: *in-arcs-imp-in-arcs-ends*)
then show *?thesis* **by** (*rule reachable-adj-trans*)
qed

corollary (**in** *shortest-paths-reachable*) *not-Vp-μ-less-inf*:
fixes $v :: 'a$
assumes $v \in verts\ G$
assumes $v \notin V_p$
shows $\mu\ c\ s\ v \neq \infty$
using *assms path-from-root-Vr μ-reach-conv* **by** *force*

lemma (**in** *shortest-paths-basic*) *enum-not0*:
assumes $v \in verts\ G$
assumes $v \neq s$
shows $enum\ v \neq enat\ 0$
using *pna*[*OF assms*(*1*,*2*)] *assms* **unfolding** *enum-def* **by** *auto*

lemma (**in** *shortest-paths-basic*) *dist-Vf-μ*:
fixes $v :: 'a$

```
  assumes vG: v ∈ verts G
  assumes ∃ r. dist v = ereal r
  shows dist v = μ c s v
proof –
  have ds: dist s =  0
    using assms source-val unfolding enum-def by force
  have ews:awalk s [] s
    using s-assms(1) unfolding awalk-def by simp
  have mu: μ c s s = ereal 0
    using min-cost-le-walk-cost[OF ews, where c=c]
    awalk-cost-Nil ds  dist-le-μ[OF s-assms(1)] zero-ereal-def
    by simp
  thus ?thesis
    using ds assms dist-le-μ[OF vG]
    dist-ge-μ[OF vG - - mu ds enum-not0]
    unfolding enum-def by fastforce
qed

lemma (in shortest-paths-reachable) pwalk-awalk:
  fixes v :: 'a
  assumes v ∈ verts G
  assumes dist v ≠ ∞
  shows awalk s (pwalk v) v
proof (cases v=s)
case True
  thus ?thesis
    using assms pwalk.simps[where v=v]
    awalk-Nil-iff by presburger
next
case False
  from assms show ?thesis
  proof (induct rule: pwalk.induct)
    fix v
    let ?e = the (parent-edge v)
    let ?u = tail G ?e
    assume ewu: ¬ (v = s ∨ dist v = ∞ ∨ v ∉ verts G) ⟹
                ?u ∈ verts G ⟹ dist ?u ≠ ∞ ⟹
                awalk s (pwalk ?u) ?u
    assume vG: v ∈ verts G
    assume dv: dist v ≠ ∞
    thus awalk s (pwalk v) v
    proof (cases v = s ∨ dist v = ∞ ∨ v ∉ verts G)
    case True
      thus ?thesis
        using pwalk.simps vG dv
        awalk-Nil-iff by fastforce
    next
    case False
      obtain e  where ee:
```

        $e \in arcs\ G$
        $parent\text{-}edge\ v = Some\ e$
        $head\ G\ e = v$
        $(tail\ G\ e) \notin V_p$
        using *pna False* **by** *blast*
      hence *awalk s (pwalk ?u) ?u*
        using *ewu[OF False] tail-in-verts* **by** *simp*
      hence *awalk s (pwalk (tail G e) @ [e]) v*
        using *ee(1−3) vG*
        by (*auto simp*: *awalk-simps simp del*: *pwalk.simps*)
      thus *?thesis*
         by (*simp only*: *pwalk.simps*[**where** *v=v*, *unfolded ee(2)*, *simplified False if-False option.sel*])
    qed
  qed
qed

lemma (**in** *shortest-paths-neg-cyc*) *Vn-μ-ninf*:
  fixes $v :: 'a$
  assumes $v \in V_n$
  shows $\mu\ c\ s\ v = -\infty$
proof −
  have *awalk s (pwalk v) v*
    using *pwalk-awalk assms* **by** *force*
moreover
  obtain *w* **where** *ww*: $w \in fst\ `\ C \cap pwalk\text{-}verts\ v$
    using *int-neg-cyc[OF assms]* **by** *blast*
  then obtain *q* **where**
    awalk w q w **and**
    $awalk\text{-}cost\ c\ q < 0$
    using *C-se* **by** *auto*
moreover
  have $w \in set\ (awalk\text{-}verts\ s\ (pwalk\ v))$
    using *ww* **unfolding** *pwalk-verts-def* **by** *fast*
ultimately
  show *?thesis* **using** *neg-cycle-imp-inf-μ* **by** *force*
qed

theorem (**in** *shortest-paths-neg-cyc*) *correct-shortest-path*:
  fixes $v :: 'a$
  assumes $v \in verts\ G$
  shows $dist\ v = \mu\ c\ s\ v$
proof(*cases dist v*)
show $\bigwedge r.\ dist\ v = ereal\ r \Longrightarrow dist\ v = \mu\ c\ s\ v$
  using *dist-Vf-μ[OF assms]* **by** *simp*
next
show $dist\ v = \infty \Longrightarrow dist\ v = \mu\ c\ s\ v$
  using *dist-le-μ[OF assms]* **by** *simp*
next

show *dist v* = −∞ ⟹ *dist v* = μ *c s v*
  using *Vn-μ-ninf assms* **by** *simp*
qed

end
theory *Matching*
imports
  Main
  Parity
  ../Graph-Theory/Graph-Theory
begin

type-synonym *label* = *nat*

definition *disjoint-arcs* :: (′*a*, ′*b*) *pre-digraph* => ′*b* ⇒ ′*b* ⇒ *bool* **where**
  disjoint-arcs G e1 e2 = (
    tail G e1 ≠ *tail G e2* ∧ *tail G e1* ≠ *head G e2* ∧
    head G e1 ≠ *tail G e2* ∧ *head G e1* ≠ *head G e2*)

definition *matching* :: (′*a*, ′*b*) *pre-digraph* ⇒ ′*b set* ⇒ *bool* **where**
  matching G M = (*M* ⊆ *arcs G* ∧ (∀ *e1* ∈ *M*. ∀ *e2* ∈ *M*. *e1* ≠ *e2* ⟶ *disjoint-arcs G e1 e2*))

definition *OSC* :: (′*a*, ′*b*) *pre-digraph* ⇒ (′*a* ⇒ *label*) ⇒ *bool* **where**
  OSC G L = (
    ∀ *e* ∈ *arcs G*.
      L (*tail G e*) = *1* ∨ *L* (*head G e*) = *1* ∨
      L (*tail G e*) = *L* (*head G e*) ∧ *L* (*tail G e*) ≥ *2*)

definition *weight*:: *label set* ⇒ (*label* ⇒ *nat*) ⇒ *nat* **where**
  weight LV f ≡ *f 1* + (∑ *i*∈*LV*. (*f i*) *div 2*)

definition *N* :: ′*a set* ⇒ (′*a* ⇒ *label*) ⇒ *label* ⇒ *nat* **where**
  N V L i ≡ *card* {*v* ∈ *V*. *L v* = *i*}

locale *matching-locale* = *digraph* +
  fixes *maxM* :: ′*b set*
  fixes *L* :: ′*a* ⇒ *label*
  assumes *matching*: *matching G maxM*
  assumes *OSC*: *OSC G L*
  assumes *weight*: *card maxM* = *weight* {*i* ∈ *L* ‘ *verts G*. *i* > *1*} (*N* (*verts G*) *L*)

sublocale *matching-locale* ⊆ *digraph* **..**

context *matching-locale* **begin**

definition *degree* :: ′*a* ⇒ *nat* **where**
  degree v ≡ *card* {*e* ∈ *arcs G*. *tail G e* = *v* ∨ *head G e* = *v*}

definition *edge-as-set* :: $'b \Rightarrow 'a$ *set* **where**
edge-as-set $e \equiv \{tail\ G\ e,\ head\ G\ e\}$

definition *matched* :: $'b$ *set* $\Rightarrow 'a \Rightarrow$ *bool* **where**
matched $M\ v \equiv v \in \bigcup\ (edge\text{-}as\text{-}set\ `\ M)$

definition *free* :: $'b$ *set* $\Rightarrow 'a \Rightarrow$ *bool* **where**
free $M\ v \equiv \neg\ matched\ M\ v$

definition *matching-i* :: *nat* $\Rightarrow 'b$ *set* $\Rightarrow 'b$ *set* **where**
matching-i $i\ M \equiv \{e \in M.\ i{=}1 \wedge (L\ (tail\ G\ e) = i \vee L\ (head\ G\ e) = i)$
$\vee\ i{>}1 \wedge L\ (tail\ G\ e) = i \wedge L\ (head\ G\ e) = i\}$

definition *V-i*:: *nat* $\Rightarrow 'b$ *set* $\Rightarrow 'a$ *set* **where**
V-i $i\ M \equiv \bigcup\ (edge\text{-}as\text{-}set\ `\ matching\text{-}i\ i\ M)$

definition *endpoint-inV* :: $'a$ *set* $\Rightarrow 'b \Rightarrow 'a$ **where**
endpoint-inV $V\ e \equiv$ *if tail* $G\ e \in V$ *then tail* $G\ e$ *else head* $G\ e$

definition *relevant-endpoint* :: $'b \Rightarrow 'a$ **where**
relevant-endpoint $e \equiv$ *if* $L\ (tail\ G\ e) = 1$ *then tail* $G\ e$ *else head* $G\ e$

lemma *definition-of-range*:
endpoint-inV $V1\ `\ matching\text{-}i\ 1\ M =$
$\{\ v.\ \exists\ e \in matching\text{-}i\ 1\ M.\ endpoint\text{-}inV\ V1\ e = v\ \}$ **by** *auto*

lemma *matching-i-arcs-as-sets*:
edge-as-set ` *matching-i* $i\ M =$
$\{\ e1.\ \exists\ e \in matching\text{-}i\ i\ M.\ edge\text{-}as\text{-}set\ e = e1\}$ **by** *auto*

lemma *matching-disjointness*:
assumes *matching* $G\ M$
assumes $e1 \in M$
assumes $e2 \in M$
assumes $e1 \neq e2$
shows *edge-as-set* $e1 \cap$ *edge-as-set* $e2 = \{\}$
using *assms*
by (*auto simp add*: *edge-as-set-def disjoint-arcs-def matching-def*)

lemma *expand-set-containment*:
assumes *matching* $G\ M$
assumes $e \in M$
shows $e \in arcs\ G$
using *assms*
by (*auto simp add*:*matching-def*)

```
theorem injectivity:
  assumes is-m: matching G M
  assumes e1-in-M1: e1 ∈ matching-i 1 M
      and e2-in-M1: e2 ∈ matching-i 1 M
  assumes diff: (e1 ≠ e2)
  shows endpoint-inV {v ∈ V. L v = 1} e1 ≠ endpoint-inV {v ∈ V. L v = 1}
e2
proof −
  from e1-in-M1 have e1 ∈ M by (auto simp add: matching-i-def)
  moreover
  from e2-in-M1 have e2 ∈ M by (auto simp add: matching-i-def)
  ultimately
  have disjoint-edge-sets: edge-as-set e1 ∩ edge-as-set e2 = {}
    using diff is-m matching-disjointness by fast
  then show ?thesis by (auto simp add: edge-as-set-def endpoint-inV-def)
qed

lemma card-M1-le-NVL1:
  assumes matching G M
  shows card (matching-i 1 M) ≤ N (verts G) L 1
proof −
  let ?f = endpoint-inV {v ∈ verts G. L v = 1}
  let ?A = matching-i 1 M
  let ?B = {v ∈ verts G. L v = 1}
  have inj-on ?f ?A using assms injectivity
    unfolding inj-on-def by blast
  moreover have ?f ‘ ?A ⊆ ?B
  proof −
    {
      fix e assume e ∈ matching-i 1 M
      hence e ∈ arcs G
        using assms by (auto simp add: matching-def matching-i-def)
      with ‹e ∈ matching-i 1 M›
      have endpoint-inV {v ∈ verts G. L v = 1} e ∈ {v ∈ verts G. L v = 1}
        using assms
          by (auto simp add: endpoint-inV-def matching-i-def intro: tail-in-verts
head-in-verts)
    }
    then show ?thesis using assms definition-of-range by blast
  qed
  moreover have finite ?B by simp
  ultimately show ?thesis unfolding N-def by (rule card-inj-on-le)
qed

lemma edge-as-set-inj-on-Mi:
  assumes matching G M
  shows inj-on edge-as-set (matching-i i M)
  using assms
  unfolding inj-on-def edge-as-set-def matching-def
```

```
    disjoint-arcs-def matching-i-def
  by blast

lemma card-edge-as-set-Mi-twice-card-partitions:
  assumes matching G M ∧ i > 1
  shows 2 ∗ card (edge-as-set‘matching-i i M)
  = card (V-i i M) (is 2 ∗ card ?C = card ?Vi)
proof −
  from assms have 1: finite (⋃ ?C)
    by (auto simp add: matching-def
      matching-i-def edge-as-set-def finite-subset)
  show ?thesis unfolding V-i-def
  proof (rule card-partition)
    show finite ?C using 1 by (rule finite-UnionD)
  next
    show finite (⋃ ?C) using 1 .
  next
    fix c assume c ∈ ?C then show card c = 2
    proof (rule imageE)
      fix x
      assume 2: c = edge-as-set x and 3: x ∈ matching-i i M
      with assms have x ∈ arcs G
        unfolding matching-i-def matching-def by blast
      then have tail G x ≠ head G x using assms 3 by (metis no-loops)
      with 2 show ?thesis by (auto simp add: edge-as-set-def)
    qed
  next
    fix x1 x2
    assume 4: x1 ∈ ?C and 5: x2 ∈ ?C and 6: x1 ≠ x2
    {
      fix e1 e2
      assume 7: x1 = edge-as-set e1 e1 ∈ matching-i i M
        x2 = edge-as-set e2 e2 ∈ matching-i i M
      from assms have matching G M by simp
      moreover
      from 7 assms have e1 ∈ M and e2 ∈ M
        by (simp-all add: matching-i-def)
      moreover from 6 7 have e1 ≠ e2 by blast
      ultimately have x1 ∩ x2 = {} unfolding 7
        by (rule matching-disjointness)
    }
    with 4 5 show x1 ∩ x2 = {} by clarsimp
  qed
qed

lemma card-Mi-twice-card-Vi:
  assumes matching G M ∧ i > 1
  shows 2 ∗ card (matching-i i M) = card (V-i i M)
proof −
```

```
  show ?thesis
    by (metis assms card-edge-as-set-Mi-twice-card-partitions
      edge-as-set-inj-on-Mi card-image)
qed

lemma card-Mi-le-floor-div-2-Vi:
  assumes matching G M ∧ i > 1
  shows card (matching-i i M) ≤ (card (V-i i M)) div 2
  using card-Mi-twice-card-Vi[OF assms]
  by arith

lemma card-Vi-le-NVLi:
  assumes i>1 ∧ matching G M
  shows card (V-i i M) ≤ N (verts G) L i
  unfolding N-def
proof (rule card-mono)
  show finite {v ∈ verts G. L v = i} using assms
    by (simp add: matching-def)
next
  let ?A = edge-as-set ` matching-i i M
  let ?C = {v ∈ verts G. L v = i}
  show V-i i M ⊆ ?C using assms unfolding V-i-def
  proof (intro Union-least)
    fix X assume X ∈ ?A
    with assms have ∃x ∈ matching-i i M. edge-as-set x = X
      by (simp add: matching-i-arcs-as-sets)
    with assms show X ⊆ ?C
      unfolding matching-def
        matching-i-def edge-as-set-def by (blast intro: tail-in-verts head-in-verts)
  qed
qed

lemma card-Mi-le-floor-div-2-NVLi:
  assumes matching G M ∧ i > 1
  shows card (matching-i i M) ≤ (N (verts G) L i) div 2
proof −
  from assms have card (V-i i M) ≤ (N (verts G) L i)
    by (simp add: card-Vi-le-NVLi)
  then have card (V-i i M) div 2 ≤ (N (verts G) L i) div 2
    by simp
  moreover from assms have
    card (matching-i i M) ≤ card (V-i i M) div 2
    by (intro card-Mi-le-floor-div-2-Vi)
  ultimately show ?thesis by auto
qed

lemma card-M-le-sum-card-Mi:
assumes matching G M and OSC G L
shows card M ≤ (∑ i ∈ L`verts G. card (matching-i i M))
```

(**is** *card* - $\leq$ *?CardMi*)
proof −
let *?UnMi* = $\bigcup$ *x* ∈ *L*'*verts G*. *matching-i x M*
from *assms* **have** *1*: *finite ?UnMi*
by (*auto simp add*: *matching-def matching-i-def finite-subset*)
{
fix *e* **assume** *e-inM*: *e* ∈ *M*
let *?v* = *relevant-endpoint e*
have *1*: *e* ∈ *matching-i* (*L ?v*) *M* **using** *assms e-inM*
proof *cases*
assume *L* (*tail G e*) = *1*
thus *?thesis* **using** *assms e-inM*
by (*simp add*: *relevant-endpoint-def matching-i-def*)
next
assume *a*: *L* (*tail G e*) ≠ *1*
have *L* (*tail G e*) = *1* ∨ *L* (*head G e*) = *1*
∨ (*L* (*tail G e*) = *L* (*head G e*) ∧ *L* (*tail G e*) >*1*)
using *assms e-inM* **unfolding** *OSC-def*
by (*auto intro*: *expand-set-containment*)
thus *?thesis* **using** *assms e-inM a*
by (*auto simp add*: *relevant-endpoint-def matching-i-def*)
qed
have *2*: *?v* ∈ *verts G* **using** *assms e-inM*
by (*auto simp add*: *matching-def relevant-endpoint-def intro*: *tail-in-verts head-in-verts*)
then have ∃ *v* ∈ *verts G*. *e* ∈ *matching-i* (*L v*) *M* **using** *assms 1 2*
by (*intro bexI*)
}
with *assms* **have** *M* ⊆ *?UnMi* **by** (*auto*)
with *assms* **and** *1* **have** *card M* $\leq$ *card ?UnMi* **by** (*intro card-mono*)
moreover from *assms* **have** *card ?UnMi* = *?CardMi*
proof (*intro card-UN-disjoint*)
show *finite* (*L*'*verts G*) **by** *simp*
next
show ∀ *i*∈*L*'*verts G*. *finite* (*matching-i i M*) **using** *assms*
using *finite-arcs*
unfolding *matching-def matching-i-def*
by (*blast intro*: *finite-subset finite-arcs*)
next
show ∀ *i* ∈ *L*'*verts G*. ∀ *j* ∈ *L*'*verts G*. *i* ≠ *j* ⟶
matching-i i M ∩ *matching-i j M* = {} **using** *assms*
by (*auto simp add*: *matching-i-def*)
qed
ultimately show *?thesis* **by** *simp*
qed

theorem *card-M-le-weight-NVLi*:
assumes *matching G M* **and** *OSC G L*
shows *card M* $\leq$ *weight* {*i* ∈ *L* ' *verts G*. *i* > *1*} (*N* (*verts G*) *L*) (**is** - $\leq$ *?W*)

proof −
let $?M01 = \sum i|\ i \in L\ `\ verts\ G \wedge (i=1 \vee i=0).\ card\ (matching\text{-}i\ i\ M)$
let $?Mgr1 = \sum i|\ i \in L\ `\ verts\ G \wedge 1 < i.\ card\ (matching\text{-}i\ i\ M)$
let $?Mi = \sum\ i{\in}L\ `\ verts\ G.\ card\ (matching\text{-}i\ i\ M)$
have $card\ M \le ?Mi$ **using** *assms* **by** (*rule card-M-le-sum-card-Mi*)
moreover
have $?Mi \le ?W$
proof −
let $?A = \{i \in L\ `\ verts\ G.\ i = 1 \vee i = 0\}$
let $?B = \{i \in L\ `\ verts\ G.\ 1 < i\}$
let $?g = \lambda\ i.\ card\ (matching\text{-}i\ i\ M)$
let $?set01 = \{\ i.\ i : L\ `\ verts\ G\ \&\ (i = 1\ |\ i = 0)\}$
have a: $L\ `\ verts\ G = ?A \cup ?B$ **using** *assms* **by** *auto*
have b: $setsum\ ?g\ (?A \cup ?B) = setsum\ ?g\ ?A + setsum\ ?g\ ?B$
by (*auto intro*: *setsum.union-disjoint*)
have 1: $?Mi = ?M01 + ?Mgr1$ **using** *assms a b* **by** *simp*
moreover
have 0: $card\ (matching\text{-}i\ 0\ M) = 0$ **using** *assms*
by (*simp add*: *matching-i-def*)
have 2: $?M01 \le N\ (verts\ G)\ L\ 1$
proof *cases*
assume a: $1 \in L\ `\ verts\ G$
have $?M01 = card\ (matching\text{-}i\ 1\ M)$
proof *cases*
assume b: $0 \in L\ `\ verts\ G$
with *a assms* **have** $?set01 = \{0,\ 1\}$ **by** *blast*
thus *?thesis* **using** *assms 0* **by** *simp*
next
assume b: $0 \notin L\ `\ verts\ G$
with *a* **have** $?set01 = \{1\}$ **by** (*auto simp del*:*One-nat-def*)
thus *?thesis* **by** *simp*
qed
thus *?thesis* **using** *assms a*
by (*simp del*: *One-nat-def*, *intro card-M1-le-NVL1*)
next
assume a: $1 \notin L\ `\ verts\ G$
show *?thesis*
proof *cases*
assume b: $0 \in L\ `\ verts\ G$
with *a assms* **have** $?set01 = \{0\}$ **by** (*auto simp del*:*One-nat-def*)
thus *?thesis* **using** *assms 0* **by** *auto*
next
assume b: $0 \notin L\ `\ verts\ G$
with *a* **have** $?set01 = \{\}$ **by** (*auto simp del*:*One-nat-def*)
then have $?M01 = (\sum i{\in}\{\}.\ card\ (matching\text{-}i\ i\ M))$ **by** *auto*
thus *?thesis* **by** *simp*
qed
qed
moreover

```
      have 3: ?Mgr1 ≤ (∑ i|i∈L ‘ verts G ∧ 1 < i. N (verts G) L i div 2)
        using assms
        by (intro setsum-mono card-Mi-le-floor-div-2-NVLi, simp)
    ultimately
    show ?thesis using 1 2 3 assms by (simp add: weight-def)
  qed
  ultimately show ?thesis by simp
qed

theorem maximum-cardinality-matching:
  matching G M' ⟶ card M' ≤ card maxM
  using card-M-le-weight-NVLi OSC matching weight
  by simp

end

end


theory Graph-Checker-Witness-Properties
imports
  Connected-Components
  Shortest-Path-Theory
  Shortest-Path-Arbitrary-Edge-Costs
  Matching

begin

end
```