The Königsberg Bridge Problem and the Friendship Theorem

Wenda Li

April 19, 2020

Abstract

This development provides a formalization of undirected graphs and simple graphs, which are based on Benedikt Nordhoff and Peter Lammich's simple formalization of labelled directed graphs [4] in the archive. Then, with our formalization of graphs, we have shown both necessary and sufficient conditions for Eulerian trails and circuits [2] as well as the fact that the Königsberg Bridge problem does not have a solution. In addition, we have also shown the Friendship Theorem in simple graphs[1, 3].

Contents

1 Undirected Multigraph and undirected trails 2

2 Degrees and related properties 2

3 Connectivity 30

4 Adjacent nodes 45

5 Undirected simple graph 46

6 Definition of Eulerian trails and circuits 49

7 Necessary conditions for Eulerian trails and circuits 49

8 Specific case of the Konigsberg Bridge Problem 51

9 Sufficient conditions for Eulerian trails and circuits 53

10 Common steps 72

theory *MoreGraph* **imports** *Complex-Main Dijkstra-Shortest-Path.Graph*
begin

1 Undirected Multigraph and undirected trails

locale *valid-unMultigraph=valid-graph G* **for** G::$('v,'w)$ *graph*+
assumes *corres*[*simp*]: $(v,w,u') \in$ *edges* $G \longleftrightarrow (u',w,v) \in$ *edges* G
and *no-id*[*simp*]:$(v,w,v) \notin$ *edges* G

fun (**in** *valid-unMultigraph*) *is-trail* :: $'v \Rightarrow ('v,'w)$ *path* $\Rightarrow 'v \Rightarrow$ *bool* **where**
is-trail v [] $v' \longleftrightarrow v=v' \wedge v' \in V$ |
is-trail v $((v1,w,v2)\#ps)$ $v' \longleftrightarrow v=v1 \wedge (v1,w,v2) \in E \wedge$
$(v1,w,v2) \notin$ *set ps* $\wedge (v2,w,v1) \notin$ *set ps* $\wedge$ *is-trail* $v2$ *ps* v'

2 Degrees and related properties

definition *degree* :: $'v \Rightarrow ('v,'w)$ *graph* $\Rightarrow$ *nat* **where**
degree v $g \equiv$ *card*($\{e.\ e \in$ *edges* $g \wedge$ *fst* $e=v\}$)

definition *odd-nodes-set* :: $('v,'w)$ *graph* $\Rightarrow 'v$ *set* **where**
odd-nodes-set $g \equiv \{v.\ v \in$ *nodes* $g \wedge$ *odd*(*degree* v g)$\}$

definition *num-of-odd-nodes* :: $('v, 'w)$ *graph* $\Rightarrow$ *nat* **where**
num-of-odd-nodes $g \equiv$ *card*(*odd-nodes-set* g)

definition *num-of-even-nodes* :: $('v, 'w)$ *graph* $\Rightarrow$ *nat* **where**
num-of-even-nodes $g \equiv$ *card*($\{v.\ v \in$ *nodes* $g \wedge$ *even*(*degree* v g)$\}$)

definition *del-unEdge* **where** *del-unEdge* v e v' $g \equiv$ (|
nodes = *nodes* g, *edges* = *edges* $g - \{(v,e,v'),(v',e,v)\}$ |)

definition *rev-path* :: $('v,'w)$ *path* $\Rightarrow ('v,'w)$ *path* **where**
rev-path ps $\equiv$ *map* $(\lambda(a,b,c).(c,b,a))$ (*rev ps*)

fun *rem-unPath*:: $('v,'w)$ *path* $\Rightarrow ('v,'w)$ *graph* $\Rightarrow ('v,'w)$ *graph* **where**
rem-unPath [] $g= g$|
rem-unPath $((v,w,v')\#ps)$ $g=$
rem-unPath ps (*del-unEdge* v w v' g)

lemma *del-undirected*: *del-unEdge* v e v' g = *delete-edge* v' e v (*delete-edge* v e v' g)
unfolding *del-unEdge-def delete-edge-def* **by** *auto*

lemma *delete-edge-sym*: *del-unEdge* v e v' g = *del-unEdge* v' e v g

unfolding *del-unEdge-def* **by** *auto*

lemma *del-unEdge-valid*[*simp*]: **assumes** *valid-unMultigraph g*
shows *valid-unMultigraph* (*del-unEdge v e v′ g*)
proof −
interpret *valid-unMultigraph g* **by** *fact*
show *?thesis*
unfolding *del-unEdge-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed

lemma *set-compre-diff*:$\{x \in A - B.\ P\ x\}=\{x \in A.\ P\ x\} - \{x \in B\ .\ P\ x\}$ **by** *blast*
lemma *set-compre-subset*: $B \subseteq A \Longrightarrow \{x \in B.\ P\ x\} \subseteq \{x \in A.\ P\ x\}$ **by** *blast*

lemma *del-edge-undirected-degree-plus*: *finite* (*edges g*) $\Longrightarrow$ (v,e,v') ∈ *edges g*
$\Longrightarrow$ (v',e,v) ∈ *edges g* $\Longrightarrow$ *degree v* (*del-unEdge v e v′ g*) + *1*=*degree v g*
proof −
assume *assms*: *finite* (*edges g*) (v,e,v') ∈ *edges g* (v',e,v) ∈ *edges g*
have *degree v* (*del-unEdge v e v′ g*) + *1*
= *card* ({*ea* ∈ *edges g* − {(v, e, v'), (v', e, v)}. *fst ea* = *v*}) + *1*
unfolding *del-unEdge-def degree-def* **by** *simp*
also have ...=*card* ({*ea* ∈ *edges g*. *fst ea* = *v*} − {*ea* ∈ {(v, e, v'), (v', e, v)}.
fst ea = *v*})+*1*
by (*metis set-compre-diff*)
also have ...=*card* ({*ea* ∈ *edges g*. *fst ea* = *v*}) − *card*({*ea* ∈ {(v, e, v'), (v', e, v)}.
fst ea = *v*})+*1*
proof −
have {(v, e, v'), (v', e, v)} ⊆ *edges g* **using** ‹(v,e,v') ∈ *edges g*› ‹(v',e,v) ∈ *edges g*›
by *auto*
hence {*ea* ∈ {(v, e, v'), (v', e, v)}. *fst ea* = *v*} ⊆ {*ea* ∈ *edges g*. *fst ea* = *v*} **by** *auto*
moreover have *finite* {*ea* ∈ {(v, e, v'), (v', e, v)}. *fst ea* = *v*} **by** *auto*
ultimately have *card* ({*ea* ∈ *edges g*. *fst ea* = *v*} − {*ea* ∈ {(v, e, v'), (v', e, v)}.
fst ea = *v*})=*card* {*ea* ∈ *edges g*. *fst ea* = *v*} − *card* {*ea* ∈ {(v, e, v'), (v', e, v)}.
fst ea = *v*}
using *card-Diff-subset* **by** *blast*
thus *?thesis* **by** *auto*
qed
also have ...=*card* ({*ea* ∈ *edges g*. *fst ea* = *v*})
proof −
have {*ea* ∈ {(v, e, v'), (v', e, v)}. *fst ea* = *v*}={(v,e,v')} **by** *auto*
hence *card* {*ea* ∈ {(v, e, v'), (v', e, v)}. *fst ea* = *v*} = *1* **by** *auto*
moreover have *card* {*ea* ∈ *edges g*. *fst ea* = *v*}≠*0*

by (*metis* (*lifting*, *mono-tags*) *Collect-empty-eq assms*(*1*) *assms*(*2*)
card-eq-0-iff fst-conv mem-Collect-eq rev-finite-subset subsetI)
ultimately show *?thesis* **by** *arith*
qed
finally have *degree v* (*del-unEdge v e v′ g*) + *1*=*card* ({*ea* ∈ *edges g*. *fst ea* = *v*}) **.**
thus *?thesis* **unfolding** *degree-def* **.**
qed

lemma *del-edge-undirected-degree-plus′*: *finite* (*edges g*) ⟹ (*v*,*e*,*v′*) ∈ *edges g*
⟹ (*v′*,*e*,*v*) ∈ *edges g* ⟹ *degree v′* (*del-unEdge v e v′ g*) + *1*=*degree v′ g*
by (*metis del-edge-undirected-degree-plus delete-edge-sym*)

lemma *del-edge-undirected-degree-minus*[*simp*]: *finite* (*edges g*) ⟹ (*v*,*e*,*v′*) ∈ *edges g*
⟹ (*v′*,*e*,*v*) ∈ *edges g* ⟹ *degree v* (*del-unEdge v e v′ g*) =*degree v g*− (*1*::*nat*)

using *del-edge-undirected-degree-plus* **by** (*metis add-diff-cancel-left′ add.commute*)

lemma *del-edge-undirected-degree-minus′*[*simp*]: *finite* (*edges g*) ⟹ (*v*,*e*,*v′*) ∈ *edges g*
⟹ (*v′*,*e*,*v*) ∈ *edges g* ⟹ *degree v′* (*del-unEdge v e v′ g*) =*degree v′ g*− (*1*::*nat*)
by (*metis del-edge-undirected-degree-minus delete-edge-sym*)

lemma *del-unEdge-com*: *del-unEdge v w v′* (*del-unEdge n e n′ g*)
= *del-unEdge n e n′* (*del-unEdge v w v′ g*)
unfolding *del-unEdge-def* **by** *auto*

lemma *rem-unPath-com*: *rem-unPath ps* (*del-unEdge v w v′ g*)
= *del-unEdge v w v′* (*rem-unPath ps g*)
proof (*induct ps arbitrary*: *g*)
case *Nil*
thus *?case* **by** (*metis rem-unPath.simps*(*1*))
next
case (*Cons a ps′*)
thus *?case* **using** *del-unEdge-com*
by (*metis prod-cases3 rem-unPath.simps*(*1*) *rem-unPath.simps*(*2*))
qed

lemma *rem-unPath-valid*[*intro*]:
valid-unMultigraph g ⟹ *valid-unMultigraph* (*rem-unPath ps g*)
proof (*induct ps*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
thus *?case*
proof −

have *valid-unMultigraph* (*rem-unPath* (x # *xs*) *g*) = *valid-unMultigraph* (*del-unEdge* (*fst x*) (*fst* (*snd x*)) (*snd* (*snd x*)) (*rem-unPath xs g*))
using *rem-unPath-com* **by** (*metis prod.collapse rem-unPath.simps*(*2*))
also have ...=*valid-unMultigraph* (*rem-unPath xs g*)
by (*metis Cons.hyps Cons.prems del-unEdge-valid*)
also have ...=*True*
using *Cons* **by** *auto*
finally have *?case*=*True* **.**
thus *?case* **by** *simp*
qed
qed

lemma (**in** *valid-unMultigraph*) *degree-frame*:
assumes *finite* (*edges G*) $x \notin \{v,\ v'\}$
shows *degree x* (*del-unEdge v w v' G*) = *degree x G* (**is** *?L*=*?R*)
proof (*cases* $(v,w,v') \in$ *edges G*)
case *True*
have *?L*=*card*({*e*. *e*∈*edges G* − {$(v,w,v'),(v',w,v)$} ∧ *fst e*=*x*})
by (*simp add*:*del-unEdge-def degree-def*)
also have ...=*card*({*e*. *e*∈*edges G* ∧ *fst e*=*x*}−{*e*. *e*∈{$(v,w,v'),(v',w,v)$} ∧ *fst e*=*x*})
by (*metis set-compre-diff*)
also have ...=*card*({*e*. *e*∈*edges G* ∧ *fst e*=*x*}) **using** ‹$x \notin \{v,\ v'\}$›
proof −
have $x \neq v \wedge x \neq v'$ **using** ‹$x \notin \{v,v'\}$›**by** *simp*
hence {*e*. *e*∈{$(v,w,v'),(v',w,v)$} ∧ *fst e*=*x*}={} **by** *auto*
thus *?thesis* **by** (*metis Diff-empty*)
qed
also have ...=*?R* **by** (*simp add*:*degree-def*)
finally show *?thesis* **.**
next
case *False*
moreover hence $(v',w,v) \notin E$ **using** *corres* **by** *auto*
ultimately have $E-\ \{(v,w,v'),(v',w,v)\}=E$ **by** *blast*
hence *del-unEdge v w v' G*=*G* **by** (*auto simp add*:*del-unEdge-def*)
thus *?thesis* **by** *auto*
qed

lemma [*simp*]: *rev-path* [] = [] **unfolding** *rev-path-def* **by** *simp*
lemma *rev-path-append*[*simp*]: *rev-path* (*xs*@*ys*) = (*rev-path ys*) @ (*rev-path xs*)
unfolding *rev-path-def rev-append* **by** *auto*
lemma *rev-path-double*[*simp*]: *rev-path*(*rev-path xs*)=*xs*
unfolding *rev-path-def* **by** (*induct xs*,*auto*)

lemma *del-UnEdge-node*[*simp*]: *v*∈*nodes* (*del-unEdge u e u' G*) ⟷ *v*∈*nodes G*

by (*metis del-unEdge-def select-convs*(*1*))

lemma [*intro*!]: *finite* (*edges* *G*) $\Longrightarrow$ *finite* (*edges* (*del-unEdge* *u* *e* *u'* *G*))
  by (*metis* *del-unEdge-def* *finite-Diff* *select-convs*(*2*))

lemma [*intro*!]: *finite* (*nodes* *G*) $\Longrightarrow$ *finite* (*nodes* (*del-unEdge* *u* *e* *u'* *G*))
  by (*metis* *del-unEdge-def* *select-convs*(*1*))

lemma [*intro*!]: *finite* (*edges* *G*) $\Longrightarrow$ *finite* (*edges* (*rem-unPath* *ps* *G*))
proof (*induct* *ps* *arbitrary*:*G*)
 case *Nil*
 thus *?case* **by** *simp*
next
 case (*Cons* *x* *xs*)
 hence *finite* (*edges* (*rem-unPath* (*x* # *xs*) *G*)) = *finite* (*edges* (*del-unEdge*
    (*fst* *x*) (*fst* (*snd* *x*)) (*snd* (*snd* *x*)) (*rem-unPath* *xs* *G*)))
  by (*metis* *rem-unPath.simps*(*2*) *rem-unPath-com* *surjective-pairing*)
 also have ...=*finite* (*edges* (*rem-unPath* *xs* *G*))
  using *del-unEdge-def*
  by (*metis* *finite.emptyI* *finite-Diff2* *finite-Diff-insert* *select-convs*(*2*))
 also have ...=*True* **using** *Cons* **by** *auto*
 finally have *?case* = *True* **.**
 thus *?case* **by** *simp*
qed

lemma *del-UnEdge-frame*[*intro*]:
 $x \in$*edges* *g* $\Longrightarrow$ $x \neq (v,e,v')$ $\Longrightarrow$ $x \neq (v',e,v)$ $\Longrightarrow$ $x \in$*edges* (*del-unEdge* *v* *e* *v'* *g*)
 unfolding *del-unEdge-def* **by** *auto*

lemma [*intro*!]: *finite* (*nodes* *G*) $\Longrightarrow$ *finite* (*odd-nodes-set* *G*)
  by (*metis* (*lifting*) *mem-Collect-eq* *odd-nodes-set-def* *rev-finite-subset* *subsetI*)

lemma [*simp*]: *nodes* (*del-unEdge* *u* *e* *u'* *G*)=*nodes* *G*
  by (*metis* *del-unEdge-def* *select-convs*(*1*))

lemma [*simp*]: *nodes* (*rem-unPath* *ps* *G*) = *nodes* *G*
proof (*induct* *ps*)
 case *Nil*
 show *?case* **by** *simp*
next
 case (*Cons* *x* *xs*)
 have *nodes* (*rem-unPath* (*x* # *xs*) *G*)=*nodes* (*del-unEdge*
   (*fst* *x*) (*fst* (*snd* *x*)) (*snd* (*snd* *x*)) (*rem-unPath* *xs* *G*))
  by (*metis* *rem-unPath.simps*(*2*) *rem-unPath-com* *surjective-pairing*)
 also have ...=*nodes* (*rem-unPath* *xs* *G*) **by** *auto*
 also have ...=*nodes* *G* **using** *Cons* **by** *auto*
 finally show *?case* **.**
qed

lemma [*intro*!]: *finite* (*nodes* *G*) $\Longrightarrow$ *finite* (*nodes* (*rem-unPath* *ps* *G*)) **by** *auto*

lemma *in-set-rev-path*[*simp*]: (*v* ′,*w*,*v*)∈*set* (*rev-path ps*) ⟷ (*v*,*w*,*v*′)∈*set ps*
proof (*induct ps*)
  case *Nil*
  thus *?case* **unfolding** *rev-path-def* **by** *auto*
next
  case (*Cons x xs*)
  obtain *x1 x2 x3* **where** *x*:*x*=(*x1* ,*x2* ,*x3*) **by** (*metis prod-cases3*)
  have *set* (*rev-path* (*x* # *xs*))=*set* ((*rev-path xs*)@[(*x3* ,*x2* ,*x1*)])
    unfolding *rev-path-def*
    using *x* **by** *auto*
  also have ...=*set* (*rev-path xs*) ∪ {(*x3* ,*x2* ,*x1*)} **by** *auto*
  finally have *set* (*rev-path* (*x* # *xs*)) =*set* (*rev-path xs*) ∪ {(*x3* ,*x2* ,*x1*)} **.**
  moreover have *set* (*x*#*xs*)= *set xs* ∪ {(*x1* ,*x2* ,*x3*)}
    by (*metis List.set-simps*(*2*) *insert-is-Un sup-commute x*)
  ultimately show *?case* **using** *Cons* **by** *auto*
qed

lemma *rem-unPath-edges*:
    edges(*rem-unPath ps G*) = *edges G* − (*set ps* ∪ *set* (*rev-path ps*))
proof (*induct ps*)
  case *Nil*
  show *?case* **unfolding** *rev-path-def* **by** *auto*
next
  case (*Cons x xs*)
  obtain *x1 x2 x3* **where** *x*: *x*=(*x1* ,*x2* ,*x3*) **by** (*metis prod-cases3*)
  hence *edges*(*rem-unPath* (*x*#*xs*) *G*)= *edges*(*del-unEdge x1 x2 x3* (*rem-unPath*
xs G))
    by (*metis rem-unPath.simps*(*2*) *rem-unPath-com*)
  also have ...=*edges*(*rem-unPath xs G*)−{(*x1* ,*x2* ,*x3*),(*x3* ,*x2* ,*x1*)}
    by (*metis del-unEdge-def select-convs*(*2*))
  also have ...= *edges G* − (*set xs* ∪ *set* (*rev-path xs*))−{(*x1* ,*x2* ,*x3*),(*x3* ,*x2* ,*x1*)}
    by (*metis Cons.hyps*)
  also have ...=*edges G* − (*set* (*x*#*xs*) ∪ *set* (*rev-path* (*x*#*xs*)))
    proof −
      have *set* (*rev-path xs*) ∪ {(*x3* ,*x2* ,*x1*)}=*set* ((*rev-path xs*)@[(*x3* ,*x2* ,*x1*)])
        by (*metis List.set-simps*(*2*) *empty-set set-append*)
       also have ...=*set* (*rev-path* (*x*#*xs*)) **unfolding** *rev-path-def* **using** *x* **by**
auto
      finally have *set* (*rev-path xs*) ∪ {(*x3* ,*x2* ,*x1*)}=*set* (*rev-path* (*x*#*xs*)) **.**
      moreover have *set xs* ∪ {(*x1* ,*x2* ,*x3*)}=*set* (*x*#*xs*)
        by (*metis List.set-simps*(*2*) *insert-is-Un sup-commute x*)
    moreover have *edges G* − (*set xs* ∪ *set* (*rev-path xs*))−{(*x1* ,*x2* ,*x3*),(*x3* ,*x2* ,*x1*)}
=
                    edges G − ((*set xs* ∪ {(*x1* ,*x2* ,*x3*)}) ∪ (*set* (*rev-path xs*) ∪
{(*x3* ,*x2* ,*x1*)}))
        by *auto*
      ultimately show *?thesis* **by** *auto*
    qed
  finally show *?case* **.**

qed

lemma *rem-unPath-graph* [*simp*]:
rem-unPath (*rev-path ps*) *G*=*rem-unPath ps G*
proof −
have *nodes*(*rem-unPath* (*rev-path ps*) *G*)=*nodes*(*rem-unPath ps G*)
by *auto*
moreover have *edges*(*rem-unPath* (*rev-path ps*) *G*)=*edges*(*rem-unPath ps G*)
proof −
have *set* (*rev-path ps*) ∪ *set* (*rev-path* (*rev-path ps*)) = *set ps* ∪ *set* (*rev-path ps*)
by *auto*
thus *?thesis* **by** (*metis rem-unPath-edges*)
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *distinct-rev-path*[*simp*]: *distinct* (*rev-path ps*) ⟷*distinct ps*
proof (*induct ps*)
case *Nil*
show *?case* **by** *auto*
next
case (*Cons x xs*)
obtain *x1 x2 x3* **where** *x*: *x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
hence *distinct* (*rev-path* (*x # xs*))=*distinct* ((*rev-path xs*)@[(*x3*,*x2*,*x1*)])
unfolding *rev-path-def* **by** *auto*
also have ...= (*distinct* (*rev-path xs*) ∧ (*x3*,*x2*,*x1*)∉*set* (*rev-path xs*))
by (*metis distinct.simps*(*2*) *distinct1-rotate rotate1.simps*(*2*))
also have ...=*distinct* (*x#xs*)
by (*metis Cons.hyps distinct.simps*(*2*) *in-set-rev-path x*)
finally have *distinct* (*rev-path* (*x # xs*))=*distinct* (*x#xs*) **.**
thus *?case* **.**
qed

lemma (**in** *valid-unMultigraph*) *is-path-rev*: *is-path v′* (*rev-path ps*) *v* ⟷ *is-path v ps v′*
proof (*induct ps arbitrary*: *v*)
case *Nil*
show *?case* **by** *auto*
next
case (*Cons x xs*)
obtain *x1 x2 x3* **where** *x*: *x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
hence *is-path v′* (*rev-path* (*x # xs*)) *v*=*is-path v′* ((*rev-path xs*) @[(*x3*,*x2*,*x1*)]) *v*
unfolding *rev-path-def* **by** *auto*
also have ...=(*is-path v′* (*rev-path xs*) *x3* ∧ (*x3*,*x2*,*x1*)∈*E* ∧ *is-path x1* [] *v*) **by** *auto*
also have ...=(*is-path x3 xs v′* ∧ (*x3*,*x2*,*x1*)∈*E* ∧ *is-path x1* [] *v*) **using** *Cons.hyps*

by *auto*
also have ...=*is-path v* (*x*#*xs*) *v'*
by (*metis corres is-path.simps*(*1*) *is-path.simps*(*2*) *is-path-memb x*)
finally have *is-path v'* (*rev-path* (*x* # *xs*)) *v*=*is-path v* (*x*#*xs*) *v'* **.**
thus *?case* **.**
qed

lemma (**in** *valid-unMultigraph*) *singleton-distinct-path* [*intro*]:
(*v*,*w*,*v'*)∈*E* ⟹ *is-trail v* [(*v*,*w*,*v'*)] *v'*
by (*metis E-validD*(*2*) *all-not-in-conv is-trail.simps set-empty*)

lemma (**in** *valid-unMultigraph*) *is-trail-path*:
is-trail v ps v' ⟷ *is-path v ps v'* ∧ *distinct ps* ∧ (*set ps* ∩ *set* (*rev-path ps*) = {})
proof (*induct ps arbitrary*:*v*)
case *Nil*
show *?case* **by** *auto*
next
case (*Cons x xs*)
obtain *x1 x2 x3* **where** *x*: *x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
hence *is-trail v* (*x*#*xs*) *v'*= (*v*=*x1* ∧ (*x1*,*x2*,*x3*)∈*E* ∧
(*x1*,*x2*,*x3*)∉*set xs* ∧(*x3*,*x2*,*x1*)∉*set xs* ∧ *is-trail x3 xs v'*)
by (*metis is-trail.simps*(*2*))
also have ...=(*v*=*x1* ∧ (*x1*,*x2*,*x3*)∈*E* ∧ (*x1*,*x2*,*x3*)∉*set xs* ∧(*x3*,*x2*,*x1*)∉*set xs* ∧ *is-path x3 xs v'*
∧ *distinct xs* ∧ (*set xs* ∩ *set* (*rev-path xs*)={}))
using *Cons.hyps* **by** *auto*
also have ...=(*is-path v* (*x*#*xs*) *v'* ∧ (*x1*,*x2*,*x3*) ≠ (*x3*,*x2*,*x1*) ∧ (*x1*,*x2*,*x3*)∉*set xs*
∧(*x3*,*x2*,*x1*)∉*set xs* ∧ *distinct xs* ∧ (*set xs* ∩ *set* (*rev-path xs*)={}))
by (*metis append-Nil is-path.simps*(*1*) *is-path-simps*(*2*) *is-path-split' no-id x*)
also have ...=(*is-path v* (*x*#*xs*) *v'* ∧ (*x1*,*x2*,*x3*) ≠ (*x3*,*x2*,*x1*) ∧(*x3*,*x2*,*x1*)∉*set xs*
∧ *distinct* (*x*#*xs*) ∧ (*set xs* ∩ *set* (*rev-path xs*)={}))
by (*metis* (*full-types*) *distinct.simps*(*2*) *x*)
also have ...=(*is-path v* (*x*#*xs*) *v'* ∧ (*x1*,*x2*,*x3*) ≠ (*x3*,*x2*,*x1*) ∧ *distinct* (*x*#*xs*)

∧ (*x3*,*x2*,*x1*)∉*set xs* ∧ *set xs* ∩ *set* (*rev-path* (*x*#*xs*))={})
proof −
have *set* (*rev-path* (*x*#*xs*)) = *set* ((*rev-path xs*)@[(*x3*,*x2*,*x1*)]) **using** *x* **by** *auto*
also have ... = *set* (*rev-path xs*) ∪ {(*x3*,*x2*,*x1*)} **by** *auto*
finally have *set* (*rev-path* (*x*#*xs*))=*set* (*rev-path xs*) ∪ {(*x3*,*x2*,*x1*)} **.**
thus *?thesis* **by** *blast*
qed
also have ...=(*is-path v* (*x*#*xs*) *v'*∧ *distinct* (*x*#*xs*) ∧ (*set* (*x*#*xs*) ∩ *set* (*rev-path* (*x*#*xs*))={}))
proof −

have (x3,x2,x1)∉set xs ⟷ (x1,x2,x3)∉ set (rev-path xs) **using** in-set-rev-path
by auto
moreover have set (rev-path (x#xs))=set (rev-path xs) ∪ {(x3,x2,x1)}
unfolding rev-path-def **using** x **by** auto
ultimately have (x1,x2,x3) ≠ (x3,x2,x1)∧ (x3,x2,x1)∉set xs
⟷ (x1,x2,x3)∉ set (rev-path (x#xs)) **by** blast
thus ?thesis
by (metis (mono-tags) Int-iff Int-insert-left-if0 List.set-simps(2) empty-iff
insertI1 x)
qed
finally have is-trail v (x#xs) v′⟷(is-path v (x#xs) v′∧ distinct (x#xs)
∧ (set (x#xs) ∩ set (rev-path (x#xs))={})) **.**
thus ?case **.**
qed

lemma (**in** valid-unMultigraph) is-trail-rev:
is-trail v′ (rev-path ps) v ⟷ is-trail v ps v′
using rev-path-append is-trail-path is-path-rev distinct-rev-path
by (metis Int-commute distinct-append)

lemma (**in** valid-unMultigraph) is-trail-intro[intro]:
is-trail v′ ps v ⟹ is-path v′ ps v **by** (induct ps arbitrary:v′,auto)

lemma (**in** valid-unMultigraph) is-trail-split:
is-trail v (p1@p2) v′ ⟹ (∃ u. is-trail v p1 u ∧ is-trail u p2 v′)
apply (induct p1 arbitrary: v,auto)
apply (metis is-trail-intro is-path-memb)
done

lemma (**in** valid-unMultigraph) is-trail-split′:is-trail v (p1@(u,w,u′)#p2) v′
⟹ is-trail v p1 u ∧ (u,w,u′)∈E ∧ is-trail u′ p2 v′
by (metis is-trail.simps(2) is-trail-split)

lemma (**in** valid-unMultigraph) distinct-elim[simp]:
assumes is-trail v ((v1,w,v2)#ps) v′
shows (v1,w,v2)∈edges(rem-unPath ps G) ⟷ (v1,w,v2)∈E
proof
assume (v1, w, v2) ∈ edges (rem-unPath ps G)
thus (v1, w, v2) ∈ E **by** (metis assms is-trail.simps(2))
next
assume (v1, w, v2) ∈ E
have (v1,w,v2)∉set ps ∧ (v2,w,v1)∉set ps **by** (metis assms is-trail.simps(2))
hence (v1,w,v2)∉set ps ∧ (v1,w,v2)∉set (rev-path ps) **by** simp
hence (v1,w,v2)∉set ps ∪ set (rev-path ps) **by** simp
hence (v1,w,v2)∈edges G − (set ps ∪ set (rev-path ps))
using ‹(v1, w, v2) ∈ E› **by** auto
thus (v1,w,v2)∈edges(rem-unPath ps G)
by (metis rem-unPath-edges)
qed

lemma *distinct-path-subset*:
assumes *valid-unMultigraph G1 valid-unMultigraph G2 edges G1* ⊆*edges G2 nodes G1* ⊆*nodes G2*
assumes *distinct-G1*:*valid-unMultigraph.is-trail G1 v ps v′*
shows *valid-unMultigraph.is-trail G2 v ps v′* **using** *distinct-G1*
proof (*induct ps arbitrary*:*v*)
case *Nil*
hence *v*=*v′*∧*v′*∈*nodes G1*
by (*metis* (*full-types*) *assms*(*1*) *valid-unMultigraph.is-trail.simps*(*1*))
hence *v*=*v′*∧*v′*∈*nodes G2* **using** ⟨*nodes G1* ⊆ *nodes G2*⟩ **by** *auto*
thus *?case* **by** (*metis assms*(*2*) *valid-unMultigraph.is-trail.simps*(*1*))
next
case (*Cons x xs*)
obtain *x1 x2 x3* **where** *x*:*x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
hence *valid-unMultigraph.is-trail G1 x3 xs v′*
by (*metis Cons.prems assms*(*1*) *valid-unMultigraph.is-trail.simps*(*2*))
hence *valid-unMultigraph.is-trail G2 x3 xs v′* **using** *Cons* **by** *auto*
moreover have *x*∈*edges G1*
by (*metis Cons.prems assms*(*1*) *valid-unMultigraph.is-trail.simps*(*2*) *x*)
hence *x*∈*edges G2* **using** ⟨*edges G1* ⊆ *edges G2*⟩ **by** *auto*
moreover have *v*=*x1*∧(*x1*,*x2*,*x3*)∉*set xs*∧(*x3*,*x2*,*x1*)∉*set xs*
by (*metis Cons.prems assms*(*1*) *valid-unMultigraph.is-trail.simps*(*2*) *x*)
hence *v*=*x1* (*x1*,*x2*,*x3*)∉*set xs* (*x3*,*x2*,*x1*)∉*set xs* **by** *auto*
ultimately show *?case* **by** (*metis assms*(*2*) *valid-unMultigraph.is-trail.simps*(*2*) *x*)
qed

lemma (**in** *valid-unMultigraph*) *distinct-path-intro′*:
assumes *valid-unMultigraph.is-trail* (*rem-unPath p G*) *v ps v′*
shows *is-trail v ps v′*
proof −
have *valid*:*valid-unMultigraph* (*rem-unPath p G*)
using *rem-unPath-valid*[*OF valid-unMultigraph-axioms*,*of p*] **by** *auto*
moreover have *nodes* (*rem-unPath p G*) ⊆ *V* **by** *auto*
moreover have *edges* (*rem-unPath p G*) ⊆ *E*
using *rem-unPath-edges* **by** *auto*
ultimately show *?thesis*
using *distinct-path-subset*[*of rem-unPath p G G*] *valid-unMultigraph-axioms assms*
by *auto*
qed

lemma (**in** *valid-unMultigraph*) *distinct-path-intro*:
assumes *valid-unMultigraph.is-trail* (*del-unEdge x1 x2 x3 G*) *v ps v′*
shows *is-trail v ps v′*
by (*metis* (*full-types*) *assms distinct-path-intro′ rem-unPath.simps*(*1*) *rem-unPath.simps*(*2*))

lemma (**in** *valid-unMultigraph*) *distinct-elim-rev*[*simp*]:
  assumes *is-trail v* ((*v1*,*w*,*v2*)#*ps*) *v'*
  shows (*v2*,*w*,*v1*)∈*edges*(*rem-unPath ps G*) ⟷ (*v2*,*w*,*v1*)∈*E*
proof –
  have *valid-unMultigraph* (*rem-unPath ps G*) **using** *valid-unMultigraph-axioms* **by** *auto*
  hence (*v2*,*w*,*v1*)∈*edges*(*rem-unPath ps G*)⟷(*v1*,*w*,*v2*)∈*edges*(*rem-unPath ps G*)
    by (*metis valid-unMultigraph.corres*)
  moreover have (*v2*,*w*,*v1*)∈*E*⟷(*v1*,*w*,*v2*)∈*E* **using** *corres* **by** *simp*
  ultimately show *?thesis* **using** *distinct-elim* **by** (*metis assms*)
qed

lemma (**in** *valid-unMultigraph*) *del-UnEdge-even*:
  assumes (*v*,*w*,*v'*) ∈ *E finite E*
  shows *v*∈*odd-nodes-set*(*del-unEdge v w v' G*) ⟷ *even* (*degree v G*)
proof –
  have *degree v* (*del-unEdge v w v' G*) + *1*=*degree v G*
    using *del-edge-undirected-degree-plus corres* **by** (*metis assms*)
  from *this* [*symmetric*] **have** *odd* (*degree v* (*del-unEdge v w v' G*)) = *even* (*degree v G*)
    by *simp*
  moreover have *v*∈*nodes* (*del-unEdge v w v' G*) **by** (*metis E-validD*(*1*) *assms*(*1*) *del-UnEdge-node*)
  ultimately show *?thesis* **unfolding** *odd-nodes-set-def* **by** *auto*
qed

lemma (**in** *valid-unMultigraph*) *del-UnEdge-even'*:
  assumes (*v*,*w*,*v'*) ∈ *E finite E*
  shows *v'*∈*odd-nodes-set*(*del-unEdge v w v' G*) ⟷ *even* (*degree v' G*)
proof –
  show *?thesis* **by** (*metis* (*full-types*) *assms corres del-UnEdge-even delete-edge-sym*)

qed

lemma *del-UnEdge-even-even*:
    assumes *valid-unMultigraph G finite*(*edges G*) *finite*(*nodes G*) (*v*, *w*, *v'*)∈*edges G*
    assumes *parity-assms*: *even* (*degree v G*) *even* (*degree v' G*)
    shows *num-of-odd-nodes*(*del-unEdge v w v' G*)=*num-of-odd-nodes G* + *2*
proof –
  interpret *G*:*valid-unMultigraph* **by** *fact*
  have *v*∈*odd-nodes-set*(*del-unEdge v w v' G*)
    by (*metis G.del-UnEdge-even assms*(*2*) *assms*(*4*) *parity-assms*(*1*))
  moreover have *v'*∈*odd-nodes-set*(*del-unEdge v w v' G*)
    by (*metis G.del-UnEdge-even' assms*(*2*) *assms*(*4*) *parity-assms*(*2*))
  ultimately have *extra-odd-nodes*:{*v*,*v'*} ⊆ *odd-nodes-set*(*del-unEdge v w v' G*)
    unfolding *odd-nodes-set-def* **by** *auto*
  moreover have *v* ∉*odd-nodes-set G* **and** *v'*∉*odd-nodes-set G*

using *parity-assms* **unfolding** *odd-nodes-set-def* **by** *auto*
hence *vv′-odd-disjoint*: {*v*,*v′*} ∩ *odd-nodes-set G* = {} **by** *auto*
moreover have *odd-nodes-set*(*del-unEdge v w v′ G*) −{*v*,*v′*}⊆*odd-nodes-set G*
proof
fix *x*
assume *x-odd-set*: *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) − {*v*, *v′*}
hence *degree x* (*del-unEdge v w v′ G*) = *degree x G*
by (*metis Diff-iff G.degree-frame assms*(*2*))
hence *odd*(*degree x G*) **using** *x-odd-set*
unfolding *odd-nodes-set-def* **by** *auto*
moreover have *x* ∈ *nodes G* **using** *x-odd-set* **unfolding** *odd-nodes-set-def*
by *auto*
ultimately show *x* ∈ *odd-nodes-set G* **unfolding** *odd-nodes-set-def* **by** *auto*
qed
moreover have *odd-nodes-set G* ⊆ *odd-nodes-set*(*del-unEdge v w v′ G*)
proof
fix *x*
assume *x-odd-set*: *x* ∈ *odd-nodes-set G*
hence *x*∉{*v*,*v′*} ⟹ *odd*(*degree x* (*del-unEdge v w v′ G*))
by (*metis* (*lifting*) *G.degree-frame assms*(*2*) *mem-Collect-eq odd-nodes-set-def*)
hence *x*∉{*v*,*v′*} ⟹ *x*∈*odd-nodes-set*(*del-unEdge v w v′ G*)
using *x-odd-set del-UnEdge-node* **unfolding** *odd-nodes-set-def* **by** *auto*
moreover have *x*∈{*v*,*v′*} ⟹ *x*∈*odd-nodes-set*(*del-unEdge v w v′ G*)
using *extra-odd-nodes* **by** *auto*
ultimately show *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) **by** *auto*
qed
ultimately have *odd-nodes-set*(*del-unEdge v w v′ G*)=*odd-nodes-set G* ∪ {*v*,*v′*}
by *auto*
thus *num-of-odd-nodes*(*del-unEdge v w v′ G*) = *num-of-odd-nodes G* + *2*
proof −
assume *odd-nodes-set*(*del-unEdge v w v′ G*)=*odd-nodes-set G* ∪ {*v*,*v′*}
moreover have *v*≠*v′* **using** *G.no-id* ‹(*v*,*w*,*v′*)∈*edges G*› **by** *auto*
hence *card*{*v*,*v′*}=*2* **by** *simp*
moreover have *odd-nodes-set G* ∩ {*v*,*v′*} = {}
using *vv′-odd-disjoint* **by** *auto*
moreover have *finite*(*odd-nodes-set G*)
by (*metis* (*lifting*) *assms*(*3*) *mem-Collect-eq odd-nodes-set-def rev-finite-subset subsetI*)
moreover have *finite* {*v*,*v′*} **by** *auto*
ultimately show *?thesis* **unfolding** *num-of-odd-nodes-def* **using** *card-Un-disjoint*
by *metis*
qed
qed

lemma *del-UnEdge-even-odd*:
assumes *valid-unMultigraph G finite*(*edges G*) *finite*(*nodes G*) (*v*, *w*, *v′*)∈*edges G*
assumes *parity-assms*: *even* (*degree v G*) *odd* (*degree v′ G*)
shows *num-of-odd-nodes*(*del-unEdge v w v′ G*)=*num-of-odd-nodes G*

proof −
  interpret *G* : *valid-unMultigraph* **by** *fact*
  have *odd-v*:*v*∈*odd-nodes-set*(*del-unEdge v w v′ G*)
    by (*metis G.del-UnEdge-even assms*(*2*) *assms*(*4*) *parity-assms*(*1*))
  have *not-odd-v′*:*v′*∉*odd-nodes-set*(*del-unEdge v w v′ G*)
    by (*metis G.del-UnEdge-even′ assms*(*2*) *assms*(*4*) *parity-assms*(*2*))
  have *odd-nodes-set*(*del-unEdge v w v′ G*) ∪ {*v′*} ⊆*odd-nodes-set G* ∪ {*v*}
    proof
      fix *x*
      assume *x-prems*: *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v′*}
      have *x*=*v′* ⟹*x*∈*odd-nodes-set G* ∪ {*v*}
        using *parity-assms*
      by (*metis* (*lifting*) *G.E-validD*(*2*) *Un-def assms*(*4*) *mem-Collect-eq odd-nodes-set-def*
)
      moreover have *x*=*v* ⟹ *x*∈*odd-nodes-set G* ∪ {*v*}
        by (*metis insertI1 insert-is-Un sup-commute*)
      moreover have *x*∉{*v*,*v′*} ⟹ *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*)
        using *x-prems* **by** *auto*
      hence *x*∉{*v*,*v′*} ⟹ *x* ∈ *odd-nodes-set G* **unfolding** *odd-nodes-set-def*
        using *G.degree-frame* ‹*finite* (*edges G*)› **by** *auto*
      hence *x*∉{*v*,*v′*} ⟹ *x*∈*odd-nodes-set G* ∪ {*v*} **by** *simp*
      ultimately show *x* ∈ *odd-nodes-set G* ∪ {*v*} **by** *auto*
    qed
  moreover have *odd-nodes-set G* ∪ {*v*} ⊆ *odd-nodes-set*(*del-unEdge v w v′ G*)
∪ {*v′*}
    proof
      fix *x*
      assume *x-prems*: *x* ∈ *odd-nodes-set G* ∪ {*v*}
      have *x*=*v* ⟹ *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v′*}
        by (*metis UnI1 odd-v*)
      moreover have *x*=*v′* ⟹ *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v′*}
        by *auto*
      moreover have *x*∉{*v*,*v′*} ⟹ *x* ∈ *odd-nodes-set G* ∪ {*v*} **using** *x-prems* **by**
auto
        hence *x*∉{*v*,*v′*} ⟹ *x*∈*odd-nodes-set* (*del-unEdge v w v′ G*) **unfolding**
odd-nodes-set-def
        using *G.degree-frame* ‹*finite* (*edges G*)› **by** *auto*
      hence *x*∉{*v*,*v′*} ⟹ *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v′*} **by** *simp*
        ultimately show *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v′*} **by** *auto*
    qed
  ultimately have *odd-nodes-set*(*del-unEdge v w v′ G*) ∪ {*v′*} = *odd-nodes-set G*
∪ {*v*}
    by *auto*
  moreover have *odd-nodes-set G* ∩ {*v*} = {}
    using *parity-assms* **unfolding** *odd-nodes-set-def* **by** *auto*
  moreover have *odd-nodes-set*(*del-unEdge v w v′ G*) ∩ {*v′*}={}
    by (*metis Int-insert-left-if0 inf-bot-left inf-commute not-odd-v′*)
  moreover have *finite* (*odd-nodes-set*(*del-unEdge v w v′ G*))
     using ‹*finite* (*nodes G*)› **by** *auto*

moreover have *finite* (*odd-nodes-set G*) **using** ‹*finite* (*nodes G*)› **by** *auto*
ultimately have *card*(*odd-nodes-set G*) + *card* {*v*} =
card(*odd-nodes-set*(*del-unEdge v w v′ G*)) + *card* {*v′*}
using *card-Un-disjoint*[*of odd-nodes-set* (*del-unEdge v w v′ G*) {*v′*}]
card-Un-disjoint[*of odd-nodes-set G* {*v*}]
by *auto*
thus *?thesis* **unfolding** *num-of-odd-nodes-def* **by** *simp*
qed

lemma *del-UnEdge-odd-even*:
assumes *valid-unMultigraph G finite*(*edges G*) *finite*(*nodes G*) (*v*, *w*, *v′*)∈*edges G*
assumes *parity-assms*: *odd* (*degree v G*) *even* (*degree v′ G*)
shows *num-of-odd-nodes*(*del-unEdge v w v′ G*)=*num-of-odd-nodes G*
by (*metis assms del-UnEdge-even-odd delete-edge-sym parity-assms valid-unMultigraph.corres*)

lemma *del-UnEdge-odd-odd*:
assumes *valid-unMultigraph G finite*(*edges G*) *finite*(*nodes G*) (*v*, *w*, *v′*)∈*edges G*
assumes *parity-assms*: *odd* (*degree v G*) *odd* (*degree v′ G*)
shows *num-of-odd-nodes G*=*num-of-odd-nodes*(*del-unEdge v w v′ G*)+*2*
proof −
interpret *G*:*valid-unMultigraph* **by** *fact*
have *v*∉*odd-nodes-set*(*del-unEdge v w v′ G*)
by (*metis G.del-UnEdge-even assms*(*2*) *assms*(*4*) *parity-assms*(*1*))
moreover have *v′*∉*odd-nodes-set*(*del-unEdge v w v′ G*)
by (*metis G.del-UnEdge-even′ assms*(*2*) *assms*(*4*) *parity-assms*(*2*))
ultimately have *vv′-disjoint*: {*v*,*v′*} ∩ *odd-nodes-set*(*del-unEdge v w v′ G*) = {}
by (*metis* (*full-types*) *Int-insert-left-if0 inf-bot-left*)
moreover have *extra-odd-nodes*:{*v*,*v′*} ⊆ *odd-nodes-set*(*G*)
unfolding *odd-nodes-set-def*
using ‹(*v*,*w*,*v′*)∈*edges G*›
by (*metis* (*lifting*) *G.E-validD empty-subsetI insert-subset mem-Collect-eq parity-assms*)

moreover have *odd-nodes-set G* −{*v*,*v′*}⊆*odd-nodes-set* (*del-unEdge v w v′ G*)

proof
fix *x*
assume *x-odd-set*: *x* ∈ *odd-nodes-set G* − {*v*, *v′*}
hence *degree x G* = *degree x* (*del-unEdge v w v′ G*)
by (*metis Diff-iff G.degree-frame assms*(*2*))
hence *odd*(*degree x* (*del-unEdge v w v′ G*)) **using** *x-odd-set*
unfolding *odd-nodes-set-def* **by** *auto*
moreover have *x* ∈ *nodes* (*del-unEdge v w v′ G*)
using *x-odd-set* **unfolding** *odd-nodes-set-def* **by** *auto*
ultimately show *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*)
unfolding *odd-nodes-set-def* **by** *auto*
qed

moreover have *odd-nodes-set* (*del-unEdge v w v′ G*) ⊆ *odd-nodes-set G*

proof

fix *x*

assume *x-odd-set*: *x* ∈ *odd-nodes-set* (*del-unEdge v w v′ G*)

hence *x*∉{*v*,*v′*} ⟹ *odd*(*degree x G*)

using *assms G.degree-frame* **unfolding** *odd-nodes-set-def*

by *auto*

hence *x*∉{*v*,*v′*} ⟹ *x*∈*odd-nodes-set G*

using *x-odd-set del-UnEdge-node* **unfolding** *odd-nodes-set-def*

by *auto*

moreover have *x*∈{*v*,*v′*} ⟹ *x*∈*odd-nodes-set G*

using *extra-odd-nodes* **by** *auto*

ultimately show *x* ∈ *odd-nodes-set G* **by** *auto*

qed

ultimately have *odd-nodes-set G*=*odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v*,*v′*}

by *auto*

thus *?thesis*

proof −

assume *odd-nodes-set G*=*odd-nodes-set* (*del-unEdge v w v′ G*) ∪ {*v*,*v′*}

moreover have *odd-nodes-set* (*del-unEdge v w v′ G*) ∩ {*v*,*v′*} = {}

using *vv′-disjoint* **by** *auto*

moreover have *finite*(*odd-nodes-set* (*del-unEdge v w v′ G*))

using *assms del-UnEdge-node finite-subset* **unfolding** *odd-nodes-set-def*

by *auto*

moreover have *finite* {*v*,*v′*} **by** *auto*

ultimately have *card*(*odd-nodes-set G*)

= *card*(*odd-nodes-set* (*del-unEdge v w v′ G*)) + *card*{*v*,*v′*}

unfolding *num-of-odd-nodes-def*

using *card-Un-disjoint*

by *metis*

moreover have *v*≠*v′* **using** *G.no-id* ‹(*v*,*w*,*v′*)∈*edges G*› **by** *auto*

hence *card*{*v*,*v′*}=*2* **by** *simp*

ultimately show *?thesis* **unfolding** *num-of-odd-nodes-def* **by** *simp*

qed

qed

lemma (**in** *valid-unMultigraph*) *rem-UnPath-parity-v′*:

assumes *finite E* *is-trail v ps v′*

shows *v*≠*v′* ⟷ (*odd* (*degree v′* (*rem-unPath ps G*)) = *even*(*degree v′ G*)) **using** *assms*

proof (*induct ps arbitrary*:*v*)

case *Nil*

thus *?case* **by** (*metis is-trail.simps*(*1*) *rem-unPath.simps*(*1*))

next

case (*Cons x xs*) **print-cases**

obtain *x1 x2 x3* **where** *x*: *x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)

hence *rem-x*:*odd* (*degree v′* (*rem-unPath* (*x#xs*) *G*)) = *odd*(*degree v′* (*del-unEdge x1 x2 x3* (*rem-unPath xs G*)))

by (*metis rem-unPath.simps*(*2*) *rem-unPath-com*)
have *x3*=*v*′ ⟹ *?case*
proof (*cases v*=*v*′)
case *True*
assume *x3*=*v*′
have *x1*=*v*′ **using** *x* **by** (*metis Cons.prems*(*2*) *True is-trail.simps*(*2*))
thus *?thesis* **using** ‹*x3*=*v*′› **by** (*metis Cons.prems*(*2*) *is-trail.simps*(*2*) *no-id* *x*)
next
case *False*
assume *x3*=*v*′
have *odd* (*degree v*′ (*rem-unPath* (*x* # *xs*) *G*)) =*odd*(*degree v*′ (
del-unEdge x1 x2 x3 (*rem-unPath xs G*))) **using** *rem-x* **.**
also have ...=*odd*(*degree v*′ (*rem-unPath xs G*) − *1*)
proof −
have *finite* (*edges* (*rem-unPath xs G*))
by (*metis* (*full-types*) *assms*(*1*) *finite-Diff rem-unPath-edges*)
moreover have (*x1*,*x2*,*x3*) ∈*edges*(*rem-unPath xs G*)
by (*metis Cons.prems*(*2*) *distinct-elim is-trail.simps*(*2*) *x*)
moreover have (*x3*,*x2*,*x1*) ∈*edges*(*rem-unPath xs G*)
by (*metis Cons.prems*(*2*) *corres distinct-elim-rev is-trail.simps*(*2*) *x*)
ultimately show *?thesis*
by (*metis* ‹*x3* = *v*′› *del-edge-undirected-degree-minus delete-edge-sym* *x*)
qed
also have ...=*even*(*degree v*′ (*rem-unPath xs G*))
proof −
have (*x1*,*x2*,*x3*)∈*E* **by** (*metis Cons.prems*(*2*) *is-trail.simps*(*2*) *x*)
hence (*x3*,*x2*,*x1*)∈*edges* (*rem-unPath xs G*)
by (*metis Cons.prems*(*2*) *corres distinct-elim-rev x*)
hence (*x3*,*x2*,*x1*)∈{*e* ∈ *edges* (*rem-unPath xs G*). *fst e* = *v*′}
using ‹*x3*=*v*′› **by** (*metis* (*mono-tags*) *fst-conv mem-Collect-eq*)
moreover have *finite* {*e* ∈ *edges* (*rem-unPath xs G*). *fst e* = *v*′}
using ‹*finite E*› **by** *auto*
ultimately have *degree v*′ (*rem-unPath xs G*)≠*0*
unfolding *degree-def* **by** *auto*
thus *?thesis* **by** *auto*
qed
also have ...=*even* (*degree v*′ *G*)
using ‹*x3* = *v*′› *assms*
by (*metis* (*mono-tags*) *Cons.hyps Cons.prems*(*2*) *is-trail.simps*(*2*) *x*)
finally have *odd* (*degree v*′ (*rem-unPath* (*x* # *xs*) *G*))=*even* (*degree v*′ *G*) **.**
thus *?thesis* **by** (*metis False*)
qed
moreover have *x3*≠*v*′⟹*?case*
proof (*cases v*=*v*′)
case *True*
assume *x3*≠*v*′
have *odd* (*degree v*′ (*rem-unPath* (*x* # *xs*) *G*)) =*odd*(*degree v*′ (
del-unEdge x1 x2 x3 (*rem-unPath xs G*))) **using** *rem-x* **.**

```
    also have ...=odd(degree v' (rem-unPath xs G) − 1)
      proof −
        have finite (edges (rem-unPath xs G))
          by (metis (full-types) assms(1) finite-Diff rem-unPath-edges)
        moreover have (x1,x2,x3) ∈edges( rem-unPath xs G)
          by (metis Cons.prems(2) distinct-elim is-trail.simps(2) x)
        moreover have (x3,x2,x1) ∈edges( rem-unPath xs G)
          by (metis Cons.prems(2) corres distinct-elim-rev is-trail.simps(2) x)
        ultimately show ?thesis
          using True x
       by (metis Cons.prems(2) del-edge-undirected-degree-minus is-trail.simps(2))
      qed
    also have ...=even(degree v' (rem-unPath xs G))
      proof −
        have (x1,x2,x3)∈E by (metis Cons.prems(2) is-trail.simps(2) x)
        hence (x1,x2,x3)∈edges (rem-unPath xs G)
          by (metis Cons.prems(2) distinct-elim x)
        hence (x1,x2,x3)∈{e ∈ edges (rem-unPath xs G). fst e = v'}
          using ‹v=v'› x  Cons
         by (metis (lifting, mono-tags) fst-conv is-trail.simps(2) mem-Collect-eq)

        moreover have finite {e ∈ edges (rem-unPath xs G). fst e = v'}
          using ‹finite E› by auto
        ultimately have degree v' (rem-unPath xs G)≠0
          unfolding degree-def by auto
        thus ?thesis by auto
      qed
    also have ...≠even (degree v' G)
      using ‹x3 ≠ v'› assms
      by (metis Cons.hyps Cons.prems(2)is-trail.simps(2) x)
    finally have odd (degree v' (rem-unPath (x # xs) G))≠even (degree v' G) .
    thus ?thesis by (metis True)
  next
    case False
    assume x3≠v'
    have odd (degree v' (rem-unPath (x # xs) G)) =odd(degree v' (
         del-unEdge x1 x2 x3 (rem-unPath xs G))) using rem-x .
    also have ...=odd(degree v' (rem-unPath xs G))
      proof −
        have v=x1 by (metis Cons.prems(2) is-trail.simps(2) x)
           hence v'∉{x1,x3} by (metis (mono-tags) False ‹x3 ≠ v'› empty-iff
insert-iff)
        moreover have valid-unMultigraph (rem-unPath xs G)
          using valid-unMultigraph-axioms by auto
        moreover have finite (edges (rem-unPath xs G))
          by (metis (full-types) assms(1) finite-Diff rem-unPath-edges)
        ultimately have degree v' (del-unEdge x1 x2 x3 (rem-unPath xs G))
                      =degree v' (rem-unPath xs G) using degree-frame
          by (metis valid-unMultigraph.degree-frame)
```

```
        thus ?thesis by simp
      qed
    also have ...=even (degree v' G)
      using assms x ‹x3 ≠ v'›
      by (metis Cons.hyps Cons.prems(2) is-trail.simps(2))
    finally have odd (degree v' (rem-unPath (x # xs) G))=even (degree v' G) .
    thus ?thesis by (metis False)
  qed
  ultimately show ?case by auto
qed

lemma (in valid-unMultigraph) rem-UnPath-parity-v:
  assumes finite E  is-trail v ps v'
  shows v≠v' ⟷ (odd (degree v (rem-unPath ps G)) = even(degree v G))
by (metis assms is-trail-rev rem-UnPath-parity-v' rem-unPath-graph)

lemma (in valid-unMultigraph) rem-UnPath-parity-others:
  assumes finite E  is-trail v ps v' n∉{v,v'}
  shows  even (degree n (rem-unPath ps G)) = even(degree n G) using assms
proof (induct ps arbitrary: v)
  case Nil
  thus ?case by auto
next
  case (Cons x xs)
  obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
  hence even (degree n (rem-unPath (x#xs) G))= even (degree n (
      del-unEdge x1 x2 x3 (rem-unPath xs G)))
    by (metis rem-unPath.simps(2) rem-unPath-com)
  have n=x3 ⟹?case
    proof -
      assume n=x3
      have even (degree n (rem-unPath (x#xs) G))= even (degree n (
        del-unEdge x1 x2 x3 (rem-unPath xs G)))
        by (metis rem-unPath.simps(2) rem-unPath-com x)
      also have ...=even(degree n (rem-unPath xs G) − 1)
        proof -
          have finite (edges (rem-unPath xs G))
            by (metis (full-types) assms(1) finite-Diff rem-unPath-edges)
          moreover have (x1,x2,x3) ∈edges( rem-unPath xs G)
            by (metis Cons.prems(2) distinct-elim is-trail.simps(2) x)
          moreover have (x3,x2,x1) ∈edges( rem-unPath xs G)
            by (metis Cons.prems(2) corres distinct-elim-rev is-trail.simps(2) x)
          ultimately show ?thesis
            using ‹n = x3› del-edge-undirected-degree-minus'
            by auto
        qed
      also have ...=odd(degree n (rem-unPath xs G))
        proof -
          have (x1,x2,x3)∈E by (metis Cons.prems(2) is-trail.simps(2) x)
```

```
        hence (x3,x2,x1)∈edges (rem-unPath xs G)
          by (metis Cons.prems(2) corres distinct-elim-rev x)
        hence (x3,x2,x1)∈{e ∈ edges (rem-unPath xs G). fst e = n}
          using ‹n=x3› by (metis (mono-tags) fst-conv mem-Collect-eq)
        moreover have finite {e ∈ edges (rem-unPath xs G). fst e = n}
          using ‹finite E› by auto
        ultimately have degree n (rem-unPath xs G)≠0
          unfolding degree-def by auto
        thus ?thesis by auto
      qed
    also have ...=even(degree n G)
      proof –
        have x3≠v' by (metis ‹n = x3› assms(3) insert-iff)
        hence odd (degree x3 (rem-unPath xs G)) = even(degree x3 G)
          using Cons assms
          by (metis is-trail.simps(2) rem-UnPath-parity-v x)
        thus ?thesis using ‹n=x3› by auto
      qed
    finally have even (degree n (rem-unPath (x#xs) G))=even(degree n G) .
    thus ?thesis .
  qed
moreover have n≠x3 ⟹?case
  proof –
    assume n≠x3
     have even (degree n (rem-unPath (x#xs) G))= even (degree n (
        del-unEdge x1 x2 x3 (rem-unPath xs G)))
      by (metis rem-unPath.simps(2) rem-unPath-com x)
    also have ...=even(degree n (rem-unPath xs G))
      proof –
        have v=x1 by (metis Cons.prems(2) is-trail.simps(2) x)
          hence n∉{x1,x3} by (metis Cons.prems(3) ‹n ≠ x3› insertE insertI1
singletonE)
        moreover have valid-unMultigraph (rem-unPath xs G)
          using valid-unMultigraph-axioms by auto
        moreover have finite (edges (rem-unPath xs G))
          by (metis (full-types) assms(1) finite-Diff rem-unPath-edges)
        ultimately have degree n (del-unEdge x1 x2 x3 (rem-unPath xs G))
                    =degree n (rem-unPath xs G) using degree-frame
          by (metis valid-unMultigraph.degree-frame)
        thus ?thesis by simp
      qed
    also have ...=even(degree n G)
      using Cons assms ‹n ≠ x3› x by auto
    finally have even (degree n (rem-unPath (x#xs) G))=even(degree n G) .
    thus ?thesis .
  qed
ultimately show ?case by auto
qed
```

lemma (**in** *valid-unMultigraph*) *rem-UnPath-even*:
  assumes *finite E finite V is-trail v ps v'*
  assumes *parity-assms*: *even* (*degree v' G*)
  shows *num-of-odd-nodes* (*rem-unPath ps G*) = *num-of-odd-nodes G*
        + (*if even* (*degree v G*)∧ *v*≠*v' then 2 else 0*) **using** *assms*
proof (*induct ps arbitrary*:*v*)
  case *Nil*
  thus *?case* **by** *auto*
next
  case (*Cons x xs*)
  obtain *x1 x2 x3* **where** *x*:*x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
  have *fin-nodes*: *finite* (*nodes* (*rem-unPath xs G*)) **using** *Cons* **by** *auto*
  have *fin-edges*: *finite* (*edges* (*rem-unPath xs G*)) **using** *Cons* **by** *auto*
  have *valid-rem-xs*: *valid-unMultigraph* (*rem-unPath xs G*) **using** *valid-unMultigraph-axioms*

    by *auto*
  have *x-in*:(*x1*,*x2*,*x3*)∈*edges* (*rem-unPath xs G*)
    by (*metis* (*full-types*) *Cons.prems*(*3*) *distinct-elim is-trail.simps*(*2*) *x*)
  have *even* (*degree x1* (*rem-unPath xs G*))
      ⟹ *even*(*degree x3* (*rem-unPath xs G*)) ⟹ *?case*
    proof −
      assume *parity-x1-x3*: *even* (*degree x1* (*rem-unPath xs G*))
                    even(*degree x3* (*rem-unPath xs G*))
      have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)= *num-of-odd-nodes*
        (*del-unEdge x1 x2 x3* (*rem-unPath xs G*))
        by (*metis rem-unPath.simps*(*2*) *rem-unPath-com x*)
      also have ... =*num-of-odd-nodes* (*rem-unPath xs G*)+*2*
      using *parity-x1-x3 fin-nodes fin-edges valid-rem-xs x-in del-UnEdge-even-even*

        by *metis*
      also have ...=*num-of-odd-nodes G*+(*if even*(*degree x3 G*) ∧ *x3*≠*v' then 2 else 0*)+*2*
        using *Cons.hyps*[*OF* ‹*finite E*› ‹*finite V*›, *of x3*] ‹*is-trail v* (*x # xs*) *v'*›
          ‹*even* (*degree v' G*)› *x*
        by *auto*
      also have ...=*num-of-odd-nodes G*+*2*
        proof −
          have *even*(*degree x3 G*) ∧ *x3*≠*v'* ⟷ *odd* (*degree x3* (*rem-unPath xs G*))
            using *Cons.prems assms*
            by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*2*) *rem-UnPath-parity-v x*)
          thus *?thesis* **using** *parity-x1-x3*(*2*) **by** *auto*
        qed
      also have ...=*num-of-odd-nodes G*+(*if even*(*degree v G*) ∧ *v*≠*v' then 2 else 0*)
        proof −
          have *x1*≠*x3* **by** (*metis valid-rem-xs valid-unMultigraph.no-id x-in*)
          moreover hence *x1*≠*v'*
            using *Cons assms*
            by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*1*) *rem-UnPath-parity-v' x*)

ultimately have *x1*∉{*x3*,*v'*} **by** *auto*
hence *even*(*degree x1 G*)
using *Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-x1-x3*(*1*)
by (*metis* (*full-types*) *is-trail.simps*(*2*) *rem-UnPath-parity-others x*)
hence *even*(*degree x1 G*) ∧ *x1*≠*v'* **using** ⟨*x1* ≠ *v'*⟩ **by** *auto*
hence *even*(*degree v G*) ∧ *v*≠*v'* **by** (*metis Cons.prems*(*3*) *is-trail.simps*(*2*) *x*)
thus *?thesis* **by** *auto*
qed
finally have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)=
num-of-odd-nodes G+(*if even*(*degree v G*) ∧ *v*≠*v'* *then 2 else 0*) **.**
thus *?thesis* **.**
qed
moreover have *even* (*degree x1* (*rem-unPath xs G*)) ⟹
odd(*degree x3* (*rem-unPath xs G*)) ⟹ *?case*
proof −
assume *parity-x1-x3*: *even* (*degree x1* (*rem-unPath xs G*))
odd (*degree x3* (*rem-unPath xs G*))
have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)= *num-of-odd-nodes*
(*del-unEdge x1 x2 x3* (*rem-unPath xs G*))
by (*metis rem-unPath.simps*(*2*) *rem-unPath-com x*)
also have ... =*num-of-odd-nodes* (*rem-unPath xs G*)
using *parity-x1-x3 fin-nodes fin-edges valid-rem-xs x-in*
by (*metis del-UnEdge-even-odd*)
also have ...=*num-of-odd-nodes G*+(*if even*(*degree x3 G*) ∧ *x3*≠*v'* *then 2 else 0*)
using *Cons.hyps Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-assms x*
by *auto*
also have ...=*num-of-odd-nodes G*+*2*
proof −
have *even*(*degree x3 G*) ∧ *x3*≠*v'* ⟷ *odd* (*degree x3* (*rem-unPath xs G*))
using *Cons.prems assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*2*) *rem-UnPath-parity-v x*)
thus *?thesis* **using** *parity-x1-x3*(*2*) **by** *auto*
qed
also have ...=*num-of-odd-nodes G*+(*if even*(*degree v G*) ∧ *v*≠*v'* *then 2 else 0*)
proof −
have *x1*≠*x3* **by** (*metis valid-rem-xs valid-unMultigraph.no-id x-in*)
moreover hence *x1*≠*v'*
using *Cons assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*1*) *rem-UnPath-parity-v' x*)
ultimately have *x1*∉{*x3*,*v'*} **by** *auto*
hence *even*(*degree x1 G*)
using *Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-x1-x3*(*1*)
by (*metis* (*full-types*) *is-trail.simps*(*2*) *rem-UnPath-parity-others x*)
hence *even*(*degree x1 G*) ∧ *x1*≠*v'* **using** ⟨*x1* ≠ *v'*⟩ **by** *auto*
hence *even*(*degree v G*) ∧ *v*≠*v'* **by** (*metis Cons.prems*(*3*) *is-trail.simps*(*2*)

x)
thus *?thesis* by *auto*
qed
finally have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)=
num-of-odd-nodes G+(*if even*(*degree v G*) ∧ *v≠v′ then 2 else 0*) .
thus *?thesis* .
qed
moreover have *odd* (*degree x1* (*rem-unPath xs G*)) ⟹
even(*degree x3* (*rem-unPath xs G*)) ⟹ *?case*
proof −
assume *parity-x1-x3*: *odd* (*degree x1* (*rem-unPath xs G*))
even (*degree x3* (*rem-unPath xs G*))
have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)= *num-of-odd-nodes*
(*del-unEdge x1 x2 x3* (*rem-unPath xs G*))
by (*metis rem-unPath.simps*(*2*) *rem-unPath-com x*)
also have ... =*num-of-odd-nodes* (*rem-unPath xs G*)
using *parity-x1-x3 fin-nodes fin-edges valid-rem-xs x-in*
by (*metis del-UnEdge-odd-even*)
also have ...=*num-of-odd-nodes G*+(*if even*(*degree x3 G*) ∧ *x3≠v′ then 2 else 0*)
using *Cons.hyps Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-assms x*
by *auto*
also have ...=*num-of-odd-nodes G*
proof −
have *even*(*degree x3 G*) ∧ *x3≠v′* ⟷ *odd* (*degree x3* (*rem-unPath xs G*))
using *Cons.prems assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*2*) *rem-UnPath-parity-v x*)
thus *?thesis* using *parity-x1-x3*(*2*) by *auto*
qed
also have ...=*num-of-odd-nodes G*+(*if even*(*degree v G*) ∧ *v≠v′ then 2 else 0*)
proof (*cases v≠v′*)
case *True*
have *x1≠x3* by (*metis valid-rem-xs valid-unMultigraph.no-id x-in*)
moreover have *is-trail x3 xs v′*
by (*metis Cons.prems*(*3*) *is-trail.simps*(*2*) *x*)
ultimately have *odd* (*degree x1* (*rem-unPath xs G*))
⟷ *odd*(*degree x1 G*)
using *True parity-x1-x3*(*1*) *rem-UnPath-parity-others x Cons.prems*(*3*) *assms*(*1*) *assms*(*2*)
by *auto*
hence *odd*(*degree x1 G*) by (*metis parity-x1-x3*(*1*))
thus *?thesis*
by (*metis* (*mono-tags*) *Cons.prems*(*3*) *Nat.add-0-right is-trail.simps*(*2*) *x*)
next
case *False*
then show *?thesis* by *auto*

qed
finally have *num-of-odd-nodes* (*rem-unPath* (*x*#*xs*) *G*)=
num-of-odd-nodes G+(*if even*(*degree v G*) $\wedge$ *v*$\neq$*v'* *then 2 else*
0) **.**
thus *?thesis* **.**
qed
moreover have *odd* (*degree x1* (*rem-unPath xs G*)) $\Longrightarrow$
odd(*degree x3* (*rem-unPath xs G*)) $\Longrightarrow$ *?case*
proof −
assume *parity-x1-x3*: *odd* (*degree x1* (*rem-unPath xs G*))
odd (*degree x3* (*rem-unPath xs G*))
have *num-of-odd-nodes* (*rem-unPath* (*x*#*xs*) *G*)= *num-of-odd-nodes*
(*del-unEdge x1 x2 x3* (*rem-unPath xs G*))
by (*metis rem-unPath.simps*(*2*) *rem-unPath-com x*)
also have ... =*num-of-odd-nodes* (*rem-unPath xs G*)−(*2*::*nat*)
using *del-UnEdge-odd-odd*
by (*metis add-implies-diff fin-edges fin-nodes parity-x1-x3 valid-rem-xs x-in*)

also have ...=*num-of-odd-nodes G*+(*if even*(*degree x3 G*) $\wedge$ *x3*$\neq$*v'* *then 2 else*
0)−(*2*::*nat*)
using *Cons assms*
by (*metis is-trail.simps*(*2*) *x*)
also have ...=*num-of-odd-nodes G*
proof −
have *even*(*degree x3 G*) $\wedge$ *x3*$\neq$*v'* $\longleftrightarrow$ *odd* (*degree x3* (*rem-unPath xs G*))
using *Cons.prems assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*2*) *rem-UnPath-parity-v x*)
thus *?thesis* **using** *parity-x1-x3*(*2*) **by** *auto*
qed
also have ...=*num-of-odd-nodes G*+(*if even*(*degree v G*) $\wedge$ *v*$\neq$*v'* *then 2 else*
0)
proof (*cases v*$\neq$*v'*)
case *True*
have *x1*$\neq$*x3* **by** (*metis valid-rem-xs valid-unMultigraph.no-id x-in*)
moreover have *is-trail x3 xs v'*
by (*metis Cons.prems*(*3*) *is-trail.simps*(*2*) *x*)
ultimately have *odd* (*degree x1* (*rem-unPath xs G*))
$\longleftrightarrow$ *odd*(*degree x1 G*)
using *True Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-x1-x3*(*1*) *rem-UnPath-parity-others*
x
by *auto*
hence *odd*(*degree x1 G*) **by** (*metis parity-x1-x3*(*1*))
thus *?thesis*
by (*metis* (*mono-tags*) *Cons.prems*(*3*) *Nat.add-0-right is-trail.simps*(*2*)
x)
next
case *False*
thus *?thesis* **by** (*metis* (*mono-tags*) *add-0-iff*)
qed

```
      finally have num-of-odd-nodes (rem-unPath (x#xs) G)=
                    num-of-odd-nodes G+(if even(degree v G) ∧ v≠v' then 2 else
0) .
      thus ?thesis .
    qed
  ultimately show ?case by metis
qed

lemma (in valid-unMultigraph) rem-UnPath-odd:
  assumes finite E finite V is-trail v ps v'
  assumes parity-assms: odd (degree v' G)
  shows num-of-odd-nodes (rem-unPath ps G) = num-of-odd-nodes G
        + (if odd (degree v G)∧ v≠v' then −2 else 0) using assms
proof (induct ps arbitrary:v)
  case Nil
  thus ?case by auto
next
  case (Cons x xs)
  obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
  have fin-nodes: finite (nodes (rem-unPath xs G)) using Cons by auto
  have fin-edges: finite (edges (rem-unPath xs G)) using Cons by auto
  have valid-rem-xs: valid-unMultigraph (rem-unPath xs G) using valid-unMultigraph-axioms

    by auto
  have x-in:(x1,x2,x3)∈edges (rem-unPath xs G)
    by (metis (full-types) Cons.prems(3) distinct-elim is-trail.simps(2) x)
  have even (degree x1 (rem-unPath xs G))
        ⟹ even(degree x3 (rem-unPath xs G)) ⟹ ?case
    proof −
      assume parity-x1-x3: even (degree x1 (rem-unPath xs G))
                      even (degree x3 (rem-unPath xs G))
      have num-of-odd-nodes (rem-unPath (x#xs) G)= num-of-odd-nodes
         (del-unEdge x1 x2 x3 (rem-unPath xs G))
        by (metis rem-unPath.simps(2) rem-unPath-com x)
      also have ... =num-of-odd-nodes (rem-unPath xs G)+2
      using parity-x1-x3 fin-nodes fin-edges valid-rem-xs x-in del-UnEdge-even-even

        by metis
      also have ...=num-of-odd-nodes G+(if odd(degree x3 G) ∧ x3≠v' then − 2
else 0 )+2
        using Cons.hyps[OF ‹finite E› ‹finite V›,of x3] ‹is-trail v (x # xs) v'›
          ‹odd (degree v' G)› x
        by auto
      also have ...=num-of-odd-nodes G
        proof −
         have odd (degree x3 G) ∧ x3≠v' ⟷ even (degree x3 (rem-unPath xs G))

            using Cons.prems assms
            by (metis is-trail.simps(2) parity-x1-x3(2) rem-UnPath-parity-v x)
```

thus *?thesis* **using** *parity-x1-x3(2)* **by** *auto*
qed
also have ...=*num-of-odd-nodes G+(if odd(degree v G) ∧ v≠v′ then −2 else 0)*
proof *(cases v≠v′)*
case *True*
have *x1≠x3* **by** *(metis valid-rem-xs valid-unMultigraph.no-id x-in)*
moreover have *is-trail x3 xs v′*
by *(metis Cons.prems(3) is-trail.simps(2) x)*
ultimately have *even (degree x1 (rem-unPath xs G))*
⟷ even (degree x1 G)
using *True Cons.prems(3) assms(1) assms(2) parity-x1-x3(1)*
rem-UnPath-parity-others x
by *auto*
hence *even (degree x1 G)* **by** *(metis parity-x1-x3(1))*
thus *?thesis*
by *(metis (hide-lams, mono-tags) Cons.prems(3) is-trail.simps(2)*
monoid-add-class.add.right-neutral x)
next
case *False*
then show *?thesis* **by** *auto*
qed
finally have *num-of-odd-nodes (rem-unPath (x#xs) G)=*
num-of-odd-nodes G+(if odd(degree v G) ∧ v≠v′ then −2 else 0) **.**
thus *?thesis* **.**
qed
moreover have *even (degree x1 (rem-unPath xs G)) ⟹*
odd(degree x3 (rem-unPath xs G)) ⟹ ?case
proof −
assume *parity-x1-x3: even (degree x1 (rem-unPath xs G))*
odd (degree x3 (rem-unPath xs G))
have *num-of-odd-nodes (rem-unPath (x#xs) G)= num-of-odd-nodes*
(del-unEdge x1 x2 x3 (rem-unPath xs G))
by *(metis rem-unPath.simps(2) rem-unPath-com x)*
also have ... =*num-of-odd-nodes (rem-unPath xs G)*
using *parity-x1-x3 fin-nodes fin-edges valid-rem-xs x-in*
by *(metis del-UnEdge-even-odd)*
also have ...=*num-of-odd-nodes G+(if odd(degree x3 G) ∧ x3≠v′ then − 2 else 0)*
using *Cons.hyps[OF ‹finite E› ‹finite V›, of x3] Cons.prems(3) assms(1) assms(2)*
parity-assms x
by *auto*
also have ...=*num-of-odd-nodes G*
proof −
have *odd(degree x3 G) ∧ x3≠v′ ⟷ even (degree x3 (rem-unPath xs G))*
using *Cons.prems assms*
by *(metis is-trail.simps(2) parity-x1-x3(2) rem-UnPath-parity-v x)*

thus *?thesis* **using** *parity-x1-x3(2)* **by** *auto*
qed
also have ...= *num-of-odd-nodes G+(if odd(degree v G)* $\wedge$ *v*$\neq$*v' then* $-$*2 else 0)*
proof *(cases v*$\neq$*v')*
case *True*
have *x1*$\neq$*x3* **by** *(metis valid-rem-xs valid-unMultigraph.no-id x-in)*
moreover have *is-trail x3 xs v'*
by *(metis Cons.prems(3) is-trail.simps(2) x)*
ultimately have *even (degree x1 (rem-unPath xs G))*
$\longleftrightarrow$ *even (degree x1 G)*
using *True Cons.prems(3) assms(1) assms(2) parity-x1-x3(1)*
rem-UnPath-parity-others x
by *auto*
hence *even (degree x1 G)* **by** *(metis parity-x1-x3(1))*
with *Cons.prems(3) x* **show** *?thesis* **by** *auto*
next
case *False*
then show *?thesis* **by** *auto*
qed
finally have *num-of-odd-nodes (rem-unPath (x#xs) G)=*
num-of-odd-nodes G+(if odd(degree v G) $\wedge$ *v*$\neq$*v' then* $-$*2 else 0)* **.**
thus *?thesis* **.**
qed
moreover have *odd (degree x1 (rem-unPath xs G))* $\Longrightarrow$
even(degree x3 (rem-unPath xs G)) $\Longrightarrow$ *?case*
proof $-$
assume *parity-x1-x3: odd (degree x1 (rem-unPath xs G))*
even (degree x3 (rem-unPath xs G))
have *num-of-odd-nodes (rem-unPath (x#xs) G)= num-of-odd-nodes*
(del-unEdge x1 x2 x3 (rem-unPath xs G))
by *(metis rem-unPath.simps(2) rem-unPath-com x)*
also have ... =*num-of-odd-nodes (rem-unPath xs G)*
using *parity-x1-x3 fin-nodes fin-edges valid-rem-xs x-in*
by *(metis del-UnEdge-odd-even)*
also have ...=*num-of-odd-nodes G+(if odd(degree x3 G)* $\wedge$ *x3*$\neq$*v' then* $-$*2 else 0)*
using *Cons.hyps Cons.prems(3) assms(1) assms(2) parity-assms x*
by *auto*
also have ...=*num-of-odd-nodes G + (*$-$ *2)*
proof $-$
have *odd(degree x3 G)* $\wedge$ *x3*$\neq$*v'* $\longleftrightarrow$ *even (degree x3 (rem-unPath xs G))*
using *Cons.prems assms*
by *(metis is-trail.simps(2) parity-x1-x3(2) rem-UnPath-parity-v x)*
hence *odd(degree x3 G)* $\wedge$ *x3*$\neq$*v'* **by** *(metis parity-x1-x3(2))*
thus *?thesis* **by** *auto*
qed
also have ...=*num-of-odd-nodes G+(if odd(degree v G)* $\wedge$ *v*$\neq$*v' then* $-$*2 else*

0)
proof −
have *x1≠x3* **by** (*metis valid-rem-xs valid-unMultigraph.no-id x-in*)
moreover hence *x1≠v'*
using *Cons assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*1*) *rem-UnPath-parity-v' x*)
ultimately have *x1∉*{*x3*,*v'*} **by** *auto*
hence *odd*(*degree x1 G*)
using *Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-x1-x3*(*1*)
by (*metis* (*full-types*) *is-trail.simps*(*2*) *rem-UnPath-parity-others x*)
hence *odd*(*degree x1 G*) ∧ *x1≠v'* **using** ‹*x1 ≠ v'*› **by** *auto*
hence *odd*(*degree v G*) ∧ *v≠v'* **by** (*metis Cons.prems*(*3*) *is-trail.simps*(*2*)
x)
thus *?thesis* **by** *auto*
qed
finally have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)=
num-of-odd-nodes G+(*if odd*(*degree v G*) ∧ *v≠v' then* −*2 else*
0) **.**
thus *?thesis* **.**
qed
moreover have *odd* (*degree x1* (*rem-unPath xs G*)) ⟹
odd(*degree x3* (*rem-unPath xs G*)) ⟹ *?case*
proof −
assume *parity-x1-x3*: *odd* (*degree x1* (*rem-unPath xs G*))
odd (*degree x3* (*rem-unPath xs G*))
have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)= *num-of-odd-nodes*
(*del-unEdge x1 x2 x3* (*rem-unPath xs G*))
by (*metis rem-unPath.simps*(*2*) *rem-unPath-com x*)
also have ... =*num-of-odd-nodes* (*rem-unPath xs G*)−(*2*::*nat*)
using *del-UnEdge-odd-odd*
by (*metis add-implies-diff fin-edges fin-nodes parity-x1-x3 valid-rem-xs x-in*)

also have ...=*num-of-odd-nodes G* −(*2*::*nat*)
proof −
have *odd*(*degree x3 G*) ∧ *x3≠v'* ⟷ *even* (*degree x3* (*rem-unPath xs G*))
using *Cons.prems assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*2*) *rem-UnPath-parity-v x*)
hence ¬(*odd*(*degree x3 G*) ∧ *x3≠v'*) **by** (*metis parity-x1-x3*(*2*))
have *num-of-odd-nodes* (*rem-unPath xs G*)=
num-of-odd-nodes G+(*if odd*(*degree x3 G*) ∧ *x3≠v' then* −*2 else 0*)
by (*metis Cons.hyps Cons.prems*(*3*) *assms*(*1*) *assms*(*2*)
is-trail.simps(*2*) *parity-assms x*)
thus *?thesis*
using ‹¬ (*odd* (*degree x3 G*) ∧ *x3 ≠ v'*)› **by** *auto*
qed
also have ...=*num-of-odd-nodes G*+(*if odd*(*degree v G*) ∧ *v≠v' then* −*2 else*
0)
proof −
have *x1≠x3* **by** (*metis valid-rem-xs valid-unMultigraph.no-id x-in*)

moreover hence *x1≠v′*
using *Cons assms*
by (*metis is-trail.simps*(*2*) *parity-x1-x3*(*1*) *rem-UnPath-parity-v′ x*)
ultimately have *x1∉{x3,v′}* **by** *auto*
hence *odd*(*degree x1 G*)
using *Cons.prems*(*3*) *assms*(*1*) *assms*(*2*) *parity-x1-x3*(*1*)
by (*metis* (*full-types*) *is-trail.simps*(*2*) *rem-UnPath-parity-others x*)
hence *odd*(*degree x1 G*) ∧ *x1≠v′* **using** ‹*x1 ≠ v′*› **by** *auto*
hence *odd*(*degree v G*) ∧ *v≠v′* **by** (*metis Cons.prems*(*3*) *is-trail.simps*(*2*)
x)
hence *v∈odd-nodes-set G*
using *Cons.prems*(*3*) *E-validD*(*1*) *x* **unfolding** *odd-nodes-set-def*
by *auto*
moreover have *v′∈odd-nodes-set G*
using *is-path-memb*[*OF is-trail-intro*[*OF assms*(*3*)]] *parity-assms*
unfolding *odd-nodes-set-def*
by *auto*
ultimately have {*v,v′*}⊆*odd-nodes-set G* **by** *auto*
moreover have *v≠v′* **by** (*metis* ‹*odd* (*degree v G*) ∧ *v ≠ v′*›)
hence *card*{*v,v′*}=*2* **by** *auto*
moreover have *finite*(*odd-nodes-set G*)
using ‹*finite V*› **unfolding** *odd-nodes-set-def*
by *auto*
ultimately have *num-of-odd-nodes G≥2* **by** (*metis card-mono num-of-odd-nodes-def*)

thus *?thesis* **using** ‹*odd* (*degree v G*) ∧ *v ≠ v′*› **by** *auto*
qed
finally have *num-of-odd-nodes* (*rem-unPath* (*x#xs*) *G*)=
num-of-odd-nodes G+(*if odd*(*degree v G*) ∧ *v≠v′ then −2 else*
0) **.**
thus *?thesis* **.**
qed
ultimately show *?case* **by** *metis*
qed

lemma (**in** *valid-unMultigraph*) *rem-UnPath-cycle*:
assumes *finite E finite V is-trail v ps v′ v=v′*
shows *num-of-odd-nodes* (*rem-unPath ps G*) = *num-of-odd-nodes G* (**is** *?L=?R*)
proof (*cases even*(*degree v′ G*))
case *True*
hence *?L* = *num-of-odd-nodes G* + (*if even* (*degree v G*)∧ *v≠v′ then 2 else 0*)
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *rem-UnPath-even*)
with *assms* **show** *?thesis* **by** *auto*
next
case *False*
hence *?L* = *num-of-odd-nodes G* + (*if odd* (*degree v G*)∧ *v≠v′ then −2 else 0*)
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *rem-UnPath-odd*)
thus *?thesis* **using** ‹*v = v′*› **by** *auto*
qed

3 Connectivity

definition (**in** *valid-unMultigraph*) *connected*::*bool* **where**
  connected $\equiv \forall\ v \in V.\ \forall v' \in V.\ v \neq v' \longrightarrow (\exists ps.\ \textit{is-path}\ v\ ps\ v')$

lemma (**in** *valid-unMultigraph*) *connected* $\Longrightarrow \forall v \in V.\ \forall v' \in V.\ v \neq v' \longrightarrow (\exists ps.\ \textit{is-trail}\ v\ ps\ v')$

proof (*rule*,*rule*,*rule*)
  fix $v\ v'$
  assume $v \in V\ v' \in V\ v \neq v'$
  assume *connected*
  obtain *ps* **where** *is-path v ps v'* **by** (*metis* ‹*connected*› ‹$v \in V$› ‹$v' \in V$› ‹$v \neq v'$› *connected-def*)
  then obtain *ps'* **where** *is-trail v ps' v'*
    proof (*induct ps arbitrary*:*v*)
      case *Nil*
      thus *?case* **by** (*metis is-trail.simps*(*1*) *is-path.simps*(*1*))
    next
      case (*Cons x xs*)
      obtain *x1 x2 x3* **where** *x*:*x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
      have *is-path x3 xs v'* **by** (*metis Cons.prems*(*2*) *is-path.simps*(*2*) *x*)
      moreover have $\bigwedge ps'.\ \textit{is-trail}\ x3\ ps'\ v' \Longrightarrow \textit{thesis}$
        proof −
          fix *ps'*
          assume *is-trail x3 ps' v'*
          hence $(x1,x2,x3) \notin \textit{set}\ ps' \wedge (x3,x2,x1) \notin \textit{set}\ ps' \Longrightarrow \textit{is-trail}\ v\ (x\#ps')\ v'$
            by (*metis Cons.prems*(*2*) *is-trail.simps*(*2*) *is-path.simps*(*2*) *x*)
          moreover have $(x1,x2,x3) \in \textit{set}\ ps' \Longrightarrow \exists ps1.\ \textit{is-trail}\ v\ ps1\ v'$
            proof −
              assume $(x1,x2,x3) \in \textit{set}\ ps'$
                then obtain *ps1 ps2* **where** $ps' = ps1 @ (x1,x2,x3) \# ps2$ **by** (*metis split-list*)
              hence *is-trail v* (*x*#*ps2*) *v'*
                using ‹*is-trail x3 ps' v'*› *x*
                by (*metis Cons.prems*(*2*) *is-trail.simps*(*2*)
                    is-trail-split is-path.simps(*2*))
              thus *?thesis* **by** *rule*
            qed
          moreover have $(x3,x2,x1) \in \textit{set}\ ps' \Longrightarrow \exists ps1.\ \textit{is-trail}\ v\ ps1\ v'$
            proof −
              assume $(x3,x2,x1) \in \textit{set}\ ps'$
                then obtain *ps1 ps2* **where** $ps' = ps1 @ (x3,x2,x1) \# ps2$ **by** (*metis split-list*)
              hence *is-trail v ps2 v'*
                using ‹*is-trail x3 ps' v'*› *x*
                by (*metis Cons.prems*(*2*) *is-trail.simps*(*2*)
                    is-trail-split is-path.simps(*2*))
              thus *?thesis* **by** *rule*
            qed

        ultimately show *thesis* **using** *Cons* **by** *auto*
      qed
    ultimately show *?case* **using** *Cons* **by** *auto*
  qed
 thus $\exists$ *ps. is-trail v ps v'* **by** *rule*
qed

lemma (**in** *valid-unMultigraph*) *no-rep-length*: *is-trail v ps v'*$\Longrightarrow$*length ps*=*card*(*set ps*)
 by (*induct ps arbitrary*:*v*, *auto*)

lemma (**in** *valid-unMultigraph*) *path-in-edges*:*is-trail v ps v'* $\Longrightarrow$ *set ps* $\subseteq$ *E*
proof (*induct ps arbitrary*:*v*)
 case *Nil*
 show *?case* **by** *auto*
next
 case (*Cons x xs*)
 obtain *x1 x2 x3* **where** *x*:*x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
 hence *is-trail x3 xs v'* **using** *Cons* **by** *auto*
 hence *set xs* $\subseteq$ *E* **using** *Cons* **by** *auto*
 moreover have *x*$\in$*E* **using** *Cons* **by** (*metis is-trail-intro is-path.simps*(*2*) *x*)
 ultimately show *?case* **by** *auto*
qed

lemma (**in** *valid-unMultigraph*) *trail-bound*:
  assumes *finite E is-trail v ps v'*
  shows *length ps* $\leq$*card E*
by (*metis* (*hide-lams*, *no-types*) *assms*(*1*) *assms*(*2*) *card-mono no-rep-length path-in-edges*)

definition (**in** *valid-unMultigraph*) *exist-path-length*:: *'v* $\Rightarrow$ *nat* $\Rightarrow$*bool* **where**
 exist-path-length v l$\equiv$$\exists$ *v' ps. is-trail v' ps v* $\wedge$ *length ps*=*l*

lemma (**in** *valid-unMultigraph*) *longest-path*:
 assumes *finite E n* $\in$ *V*
 shows $\exists$ *v.* $\exists$ *max-path. is-trail v max-path n* $\wedge$
    ($\forall$ *v'.* $\forall$ *e*$\in$*E.* $\neg$*is-trail v'* (*e*#*max-path*) *n*)
proof (*rule ccontr*)
 assume *contro*:$\neg$ ($\exists$ *v max-path. is-trail v max-path n*
     $\wedge$ ($\forall$ *v'.* $\forall$ *e*$\in$*E.* $\neg$*is-trail v'* (*e*#*max-path*) *n*))
 hence *induct*:($\forall$ *v max-path. is-trail v max-path n*
     $\longrightarrow$ ($\exists$ *v'.* $\exists$ *e*$\in$*E. is-trail v'* (*e*#*max-path*) *n*)) **by** *auto*
 have *is-trail n* [] *n* **using** ‹*n* $\in$ *V*› **by** *auto*
 hence *exist-path-length n 0* **unfolding** *exist-path-length-def* **by** *auto*
 moreover have $\forall$ *y. exist-path-length n y* $\longrightarrow$ *y* $\leq$ *card E*
  using *trail-bound*[*OF* ‹*finite E*›] **unfolding** *exist-path-length-def*
  by *auto*
 hence *bound*:$\forall$ *y. exist-path-length n y* $\longrightarrow$ *y* $\leq$ *card E* **by** *auto*
 ultimately have *exist-path-length n* (*GREATEST x. exist-path-length n x*)

using *GreatestI-nat* **by** *auto*
then obtain *v max-path* **where**
max-path:*is-trail v max-path n length max-path*=(*GREATEST x*. *exist-path-length n x*)
by (*metis exist-path-length-def*)
hence ∃ *v′ e*. *is-trail v′* (*e*#*max-path*) *n* **using** *induct* **by** *metis*
hence *exist-path-length n* (*length max-path* +*1*)
by (*metis One-nat-def exist-path-length-def list*.*size*(*4*))
hence *length max-path* + *1* ≤ (*GREATEST x*. *exist-path-length n x*)
by (*metis Greatest-le-nat bound*)
hence *length max-path* + *1* ≤ *length max-path* **using** *max-path* **by** *auto*
thus *False* **by** *auto*
qed

lemma *even-card′*:
assumes *even*(*card A*) *x*∈*A*
shows ∃ *y*∈*A*. *y*≠*x*
proof (*rule ccontr*)
assume ¬ (∃ *y*∈*A*. *y* ≠ *x*)
hence ∀ *y*∈*A*. *y*=*x* **by** *auto*
hence *A*={*x*} **by** (*metis all-not-in-conv assms*(*2*) *insertI2 mk-disjoint-insert*)
hence *card*(*A*)=*1* **by** *auto*
thus *False* **using** ‹*even*(*card A*)› **by** *auto*
qed

lemma *odd-card*:
assumes *finite A odd*(*card A*)
shows ∃ *x*. *x*∈*A*
by (*metis all-not-in-conv assms*(*2*) *card-empty even-zero*)

lemma (**in** *valid-unMultigraph*) *extend-distinct-path*:
assumes *finite E is-trail v′ ps v*
assumes *parity-assms*:(*even* (*degree v′ G*)∧*v′*≠*v*)∨(*odd* (*degree v′ G*)∧*v′*=*v*)
shows ∃ *e v1*. *is-trail v1* (*e*#*ps*) *v*
proof −
have (*even* (*degree v′ G*)∧*v′*≠*v*) ⟹ *odd*(*degree v′* (*rem-unPath ps G*))
by (*metis assms*(*1*) *assms*(*2*) *rem-UnPath-parity-v*)
moreover have (*odd* (*degree v′ G*)∧*v′*=*v*) ⟹ *odd*(*degree v′* (*rem-unPath ps G*))
by (*metis assms*(*1*) *assms*(*2*) *rem-UnPath-parity-v′*)
ultimately have *odd*(*degree v′* (*rem-unPath ps G*)) **using** *parity-assms* **by** *auto*
hence *odd* (*card* {*e*. *fst e*=*v′* ∧ *e*∈*edges G* − (*set ps* ∪ *set* (*rev-path ps*))})
using *rem-unPath-edges* **unfolding** *degree-def*
by (*metis* (*lifting*, *no-types*) *Collect-cong*)
hence {*e*. *fst e*=*v′* ∧ *e*∈*E* − (*set ps* ∪ *set* (*rev-path ps*))}≠{}
by (*metis empty-iff finite*.*emptyI odd-card*)
then obtain *v0 w* **where** *v0w*: (*v′*,*w*,*v0*)∈*E* (*v′*,*w*,*v0*)∉*set ps* ∪ *set* (*rev-path ps*) **by** *auto*

```
  hence is-trail v0 ((v0,w,v')#ps) v
   by (metis (hide-lams, mono-tags) Un-iff assms(2) corres in-set-rev-path is-trail.simps(2))

  thus ?thesis by metis
qed
```

replace an edge (or its reverse in a path) by another path (in an undirected graph)

```
fun replace-by-UnPath:: ('v,'w) path ⇒ 'v ×'w ×'v ⇒ ('v,'w) path ⇒  ('v,'w)
path where
  replace-by-UnPath [] - - = [] |
  replace-by-UnPath (x#xs) (v,e,v') ps =
    (if x=(v,e,v') then ps@replace-by-UnPath xs (v,e,v') ps
     else if x=(v',e,v) then (rev-path ps)@replace-by-UnPath xs (v,e,v') ps
     else x#replace-by-UnPath xs (v,e,v') ps)

lemma (in valid-unMultigraph) del-unEdge-connectivity:
  assumes connected ∃ps. valid-graph.is-path (del-unEdge v e v' G) v ps v'
  shows valid-unMultigraph.connected (del-unEdge v e v' G)
proof –
  have valid-unMulti:valid-unMultigraph (del-unEdge v e v' G)
    using valid-unMultigraph-axioms by simp
  have valid-graph: valid-graph (del-unEdge v e v' G)
    using valid-graph-axioms del-undirected by (metis delete-edge-valid)
 obtain ex-path where ex-path:valid-graph.is-path (del-unEdge v e v' G) v ex-path
v'
    by (metis assms(2))
  show ?thesis unfolding valid-unMultigraph.connected-def[OF valid-unMulti]
  proof (rule,rule,rule)
    fix n n'
    assume n : n ∈nodes (del-unEdge v e v' G)
    assume n': n'∈nodes (del-unEdge v e v' G)
    assume n≠n'
    obtain ps where ps:is-path n ps n'
      by (metis ‹n≠n'› n n' ‹connected› connected-def del-UnEdge-node)
    hence valid-graph.is-path (del-unEdge v e v' G)
          n (replace-by-UnPath ps (v,e,v') ex-path) n'
      proof (induct ps arbitrary:n)
        case Nil
           thus ?case by (metis is-path.simps(1) n' replace-by-UnPath.simps(1)
valid-graph
          valid-graph.is-path-simps(1))
      next
        case (Cons x xs)
        obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
        have x=(v,e,v') ⟹ ?case
          proof –
            assume x=(v,e,v')
            hence valid-graph.is-path (del-unEdge v e v' G)
```

```
              n (replace-by-UnPath (x#xs) (v,e,v') ex-path) n'
              = valid-graph.is-path (del-unEdge v e v' G)
              n (ex-path@(replace-by-UnPath xs (v,e,v') ex-path)) n'
             by (metis replace-by-UnPath.simps(2))
           also have ...=True
          by (metis Cons.hyps Cons.prems ‹x = (v, e, v')› ex-path is-path.simps(2)
valid-graph
              valid-graph.is-path-split)
           finally show ?thesis by simp
         qed
       moreover have x=(v',e,v) ⟹ ?case
         proof -
           assume x=(v',e,v)
           hence valid-graph.is-path (del-unEdge v e v' G)
              n (replace-by-UnPath (x#xs) (v,e,v') ex-path) n'
              = valid-graph.is-path (del-unEdge v e v' G)
              n ((rev-path ex-path)@(replace-by-UnPath xs (v,e,v') ex-path)) n'
         by (metis Cons.prems is-path.simps(2) no-id replace-by-UnPath.simps(2))
           also have ...=True
              by (metis Cons.hyps Cons.prems ‹x = (v', e, v)› is-path.simps(2)
ex-path valid-graph
              valid-graph.is-path-split valid-unMulti valid-unMultigraph.is-path-rev)
           finally show ?thesis by simp
         qed
       moreover have x≠(v,e,v')∧x≠(v',e,v)⟹?case
             by (metis Cons.hyps Cons.prems del-UnEdge-frame is-path.simps(2)
replace-by-UnPath.simps(2)
             valid-graph valid-graph.is-path.simps(2) x)
       ultimately show ?case by auto
     qed
   thus ∃ps. valid-graph.is-path (del-unEdge v e v' G) n ps n' by auto
  qed
qed

lemma (in valid-unMultigraph) path-between-odds:
 assumes odd(degree v G) odd(degree v' G) finite E  v≠v' num-of-odd-nodes G=2
 shows ∃ps. is-trail v ps v'
proof -
   have v∈V
    proof (rule ccontr)
      assume v∉V
      hence ∀ e ∈ E. fst e ≠ v by (metis E-valid(1) imageI subsetD)
      hence degree v G=0 unfolding degree-def using ‹finite E›
        by force
      thus False using ‹odd(degree v G)› by auto
    qed
  have v'∈V
    proof (rule ccontr)
      assume v'∉V
```

hence ∀ *e* ∈ *E*. *fst e* ≠ *v′* **by** (*metis E-valid*(*1*) *imageI subsetD*)
hence *degree v′ G=0* **unfolding** *degree-def* **using** ‹*finite E*›
by *force*
thus *False* **using** ‹*odd*(*degree v′ G*)› **by** *auto*
qed
then obtain *max-path v0* **where** *max-path*:
is-trail v0 max-path v′
(∀ *n*. ∀ *w*∈*E*. ¬*is-trail n* (*w*#*max-path*) *v′*)
using *longest-path*[*of v′*] **by** (*metis assms*(*3*))
have *even*(*degree v0 G*)⟹*v0=v′* ⟹ *v0=v*
by (*metis assms*(*2*))
moreover have *even*(*degree v0 G*)⟹*v0*≠*v′* ⟹ *v0=v*
proof −
assume*even*(*degree v0 G*) *v0*≠*v′*
hence ∃ *w v1*. *is-trail*
v1 (*w*#*max-path*) *v′*
by (*metis assms*(*3*) *extend-distinct-path max-path*(*1*))
thus *?thesis* **by** (*metis* (*full-types*) *is-trail.simps*(*2*) *max-path*(*2*) *prod.exhaust*)
qed
moreover have *odd*(*degree v0 G*)⟹*v0=v′*⟹*v0=v*
proof −
assume*odd*(*degree v0 G*) *v0=v′*
hence ∃ *w v1*. *is-trail v1* (*w*#*max-path*) *v′*
by (*metis assms*(*3*) *extend-distinct-path max-path*(*1*))
thus *?thesis* **by** (*metis* (*full-types*) *List.set-simps*(*2*) *insert-subset max-path*(*2*)
path-in-edges)
qed
moreover have *odd*(*degree v0 G*)⟹*v0*≠*v′*⟹*v0=v*
proof (*rule ccontr*)
assume *v0* ≠ *v odd*(*degree v0 G*) *v0*≠*v′*
moreover have *v*∈*odd-nodes-set G*
using ‹*v* ∈ *V*› ‹ *odd* (*degree v G*)› **unfolding** *odd-nodes-set-def*
by *auto*
moreover have *v′*∈*odd-nodes-set G*
using ‹*v′* ∈ *V*› ‹*odd* (*degree v′ G*)›
unfolding *odd-nodes-set-def*
by *auto*
ultimately have {*v*,*v′*,*v0*} ⊆ *odd-nodes-set G*
using *is-path-memb*[*OF is-trail-intro*[*OF* ‹*is-trail v0 max-path v′*›]] *max-path*(*1*)
unfolding *odd-nodes-set-def*
by *auto*
moreover have *card* {*v*,*v′*,*v0*}=*3* **using** ‹*v0*≠*v*› ‹*v*≠*v′*› ‹*v0*≠*v′*› **by** *auto*
moreover have *finite* (*odd-nodes-set G*)
using *assms*(*5*) *card-eq-0-iff* [*of odd-nodes-set G*] **unfolding** *num-of-odd-nodes-def*

by *auto*
ultimately have *3*≤*card*(*odd-nodes-set G*) **by** (*metis card-mono*)
thus *False* **using** ‹*num-of-odd-nodes G=2*› **unfolding** *num-of-odd-nodes-def*
by *auto*

qed
ultimately have *v0=v* **by** *auto*
thus *?thesis* **by** (*metis max-path*(*1*))
qed

lemma (**in** *valid-unMultigraph*) *del-unEdge-even-connectivity*:
assumes *finite E finite V connected* $\forall n \in V.$ *even*(*degree n G*) $(v,e,v') \in E$
shows *valid-unMultigraph.connected* (*del-unEdge v e v' G*)
proof –
have *valid-unMulti*:*valid-unMultigraph* (*del-unEdge v e v' G*)
using *valid-unMultigraph-axioms* **by** *simp*
have *valid-graph*: *valid-graph* (*del-unEdge v e v' G*)
using *valid-graph-axioms del-undirected* **by** (*metis delete-edge-valid*)
have *fin-E'*: *finite*(*edges* (*del-unEdge v e v' G*))
by (*metis* (*hide-lams*, *no-types*) *assms*(*1*) *del-undirected delete-edge-def finite-Diff select-convs*(*2*))
have *fin-V'*: *finite*(*nodes* (*del-unEdge v e v' G*))
by (*metis* (*mono-tags*) *assms*(*2*) *del-undirected delete-edge-def select-convs*(*1*))
have *all-even*: $\forall n \in$ *nodes* (*del-unEdge v e v' G*). $n \notin \{v,v'\}$
$\longrightarrow$ *even*(*degree n* (*del-unEdge v e v' G*))
by (*metis* (*full-types*) *assms*(*1*) *assms*(*4*) *degree-frame del-UnEdge-node*)
have *even* (*degree v G*) **by** (*metis* (*full-types*) *E-validD*(*1*) *assms*(*4*) *assms*(*5*))
moreover have *even* (*degree v' G*) **by** (*metis* (*full-types*) *E-validD*(*2*) *assms*(*4*) *assms*(*5*))
moreover have *num-of-odd-nodes G* = *0*
using ‹$\forall n \in V.$ *even*(*degree n G*)› ‹*finite V*›
unfolding *num-of-odd-nodes-def odd-nodes-set-def* **by** *auto*
ultimately have *num-of-odd-nodes* (*del-unEdge v e v' G*) = *2*
using *del-UnEdge-even-even*[*of G v e v'*,*OF valid-unMultigraph-axioms*]
by (*metis assms*(*1*) *assms*(*2*) *assms*(*5*) *monoid-add-class.add.left-neutral*)
moreover have *odd* (*degree v* (*del-unEdge v e v' G*))
using ‹*even* (*degree v G*)› *del-UnEdge-even*[*OF* ‹$(v,e,v') \in E$› ‹*finite E*›]
unfolding *odd-nodes-set-def*
by *auto*
moreover have *odd* (*degree v'* (*del-unEdge v e v' G*))
using ‹*even* (*degree v' G*)› *del-UnEdge-even'*[*OF* ‹$(v,e,v') \in E$› ‹*finite E*›]
unfolding *odd-nodes-set-def*
by *auto*
moreover have *finite* (*edges* (*del-unEdge v e v' G*))
using ‹*finite E*› **by** *auto*
moreover have $v \neq v'$ **using** *no-id* ‹$(v,e,v') \in E$› **by** *auto*
ultimately have $\exists$ *ps. valid-unMultigraph.is-trail* (*del-unEdge v e v' G*) *v ps v'*
using *valid-unMultigraph.path-between-odds*[*OF valid-unMulti*,*of v v'*]
by *auto*
thus *?thesis*
by (*metis* (*full-types*) *assms*(*3*) *del-unEdge-connectivity valid-unMulti valid-unMultigraph.is-trail-intro*)
qed

lemma (**in** *valid-graph*) *path-end*:$ps \neq [] \Longrightarrow$ *is-path v ps v'* $\Longrightarrow v'$=*snd* (*snd*(*last ps*))
by (*induct ps arbitrary*:*v*,*auto*)

lemma (**in** *valid-unMultigraph*) *connectivity-split*:
assumes *connected* ¬*valid-unMultigraph.connected* (*del-unEdge v w v' G*)
(*v*,*w*,*v'*)∈*E*
obtains *G1 G2* **where**
nodes G1={*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v*}
and *edges G1*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge v w v' G*)
∧ *n*∈*nodes G1* ∧ *n'*∈*nodes G1*}
and *nodes G2*={*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v'*}
and *edges G2*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge v w v' G*)
∧ *n*∈*nodes G2* ∧ *n'*∈*nodes G2*}
and *edges G1* ∪ *edges G2* = *edges* (*del-unEdge v w v' G*)
and *edges G1* ∩ *edges G2*={}
and *nodes G1* ∪ *nodes G2*=*nodes* (*del-unEdge v w v' G*)
and *nodes G1* ∩ *nodes G2*={}
and *valid-unMultigraph G1*
and *valid-unMultigraph G2*
and *valid-unMultigraph.connected G1*
and *valid-unMultigraph.connected G2*
proof −
have *valid0*:*valid-graph* (*del-unEdge v w v' G*) **using** *valid-graph-axioms*
by (*metis del-undirected delete-edge-valid*)
have *valid0'*:*valid-unMultigraph* (*del-unEdge v w v' G*) **using** *valid-unMultigraph-axioms*

by (*metis del-unEdge-valid*)
obtain *G1-nodes* **where** *G1-nodes*:*G1-nodes*=
{*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v*}
by *metis*
then obtain *G1* **where** *G1*:*G1*=
(|*nodes*=*G1-nodes*, *edges*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge v w v' G*)
∧ *n*∈*G1-nodes* ∧ *n'*∈*G1-nodes*}|)
by *metis*
obtain *G2-nodes* **where** *G2-nodes*:*G2-nodes*=
{*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v'*}
by *metis*
then obtain *G2* **where** *G2*:*G2*=
(|*nodes*=*G2-nodes*, *edges*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge v w v' G*)
∧ *n*∈*G2-nodes* ∧ *n'*∈*G2-nodes*}|)
by *metis*
have *valid-G1*:*valid-unMultigraph G1*
using *G1 valid-unMultigraph.corres*[*OF valid0'*] *valid-unMultigraph.no-id*[*OF valid0'*]
by (*unfold-locales*,*auto*)
hence *valid-G1'*:*valid-graph G1* **using** *valid-unMultigraph-def* **by** *auto*
have *valid-G2*:*valid-unMultigraph G2*

using *G2 valid-unMultigraph.corres*[*OF valid0'*] *valid-unMultigraph.no-id*[*OF valid0'*]
by (*unfold-locales*,*auto*)
hence *valid-G2'*: *valid-graph G2* **using** *valid-unMultigraph-def* **by** *auto*
have *nodes G1*={*n*. ∃ *ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v*}
using *G1-nodes G1* **by** *auto*
moreover have *edges G1*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge v w v' G*)
∧ *n*∈*nodes G1* ∧ *n'*∈*nodes G1*}
using *G1-nodes G1* **by** *auto*
moreover have *nodes G2*={*n*. ∃ *ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v'*}
using *G2-nodes G2* **by** *auto*
moreover have *edges G2*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge v w v' G*)
∧ *n*∈*nodes G2* ∧ *n'*∈*nodes G2*}
using *G2-nodes G2* **by** *auto*
moreover have *nodes G1* ∪ *nodes G2*=*nodes* (*del-unEdge v w v' G*)
proof (*rule ccontr*)
assume *nodes G1* ∪ *nodes G2* ≠ *nodes* (*del-unEdge v w v' G*)
moreover have *nodes G1* ⊆ *nodes* (*del-unEdge v w v' G*)
using *valid-graph.is-path-memb*[*OF valid0*] *G1 G1-nodes* **by** *auto*
moreover have *nodes G2* ⊆ *nodes* (*del-unEdge v w v' G*)
using *valid-graph.is-path-memb*[*OF valid0*] *G2 G2-nodes* **by** *auto*
ultimately obtain *n* **where** *n*:
n∈*nodes* (*del-unEdge v w v' G*) *n*∉*nodes G1 n*∉*nodes G2*
by *auto*
hence *n-neg-v* : ¬(∃ *ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v*) **and**
n-neg-v': ¬(∃ *ps*. *valid-graph.is-path* (*del-unEdge v w v' G*) *n ps v'*)
using *G1 G1-nodes G2 G2-nodes* **by** *auto*
hence *n*≠*v* **by** (*metis n*(*1*) *valid0 valid-graph.is-path-simps*(*1*))
then obtain *nvs* **where** *nvs*: *is-path n nvs v* **using** ‹*connected*›
by (*metis E-validD*(*1*) *assms*(*3*) *connected-def del-UnEdge-node n*(*1*))
then obtain *nvs'* **where** *nvs'*: *nvs'*=*takeWhile* (λ*x*. *x*≠(*v*,*w*,*v'*)∧*x*≠(*v'*,*w*,*v*)) *nvs* **by** *auto*
moreover have *nvs-nvs'*:*nvs*=*nvs'*@*dropWhile* (λ*x*. *x*≠(*v*,*w*,*v'*)∧*x*≠(*v'*,*w*,*v*)) *nvs*
using *nvs' takeWhile-dropWhile-id* **by** *auto*
ultimately obtain *n'* **where** *is-path-nvs'*: *is-path n nvs' n'*
and *is-path n'* (*dropWhile* (λ*x*. *x*≠(*v*,*w*,*v'*)∧*x*≠(*v'*,*w*,*v*)) *nvs*) *v*
using *nvs is-path-split*[*of n nvs' dropWhile* (λ*x*. *x*≠(*v*,*w*,*v'*)∧*x*≠(*v'*,*w*,*v*)) *nvs*] **by** *auto*
have *n'*=*v* ∨ *n'*=*v'*
proof (*cases dropWhile* (λ*x*. *x*≠(*v*,*w*,*v'*)∧*x*≠(*v'*,*w*,*v*)) *nvs*)
case *Nil*
hence *nvs*=*nvs'* **using** *nvs-nvs'* **by** (*metis append-Nil2*)
hence *n'*=*v* **using** *nvs is-path-nvs' path-end* **by** (*metis* (*mono-tags*) *is-path.simps*(*1*))
thus *?thesis* **by** *auto*
next

```
        case (Cons x xs)
        hence dropWhile (λx. x≠(v,w,v')∧x≠(v',w,v)) nvs≠[] by auto
        hence hd (dropWhile (λx. x≠(v,w,v')∧x≠(v',w,v)) nvs)=(v,w,v')
              ∨ hd (dropWhile (λx. x≠(v,w,v')∧x≠(v',w,v)) nvs)=(v',w,v)
          by (metis (lifting, full-types) hd-dropWhile)
        hence x=(v,w,v')∨x=(v',w,v) using Cons by auto
        thus ?thesis
          using ‹is-path n' (dropWhile (λx. x ≠ (v, w, v') ∧ x ≠ (v', w, v)) nvs)
v›
          by (metis Cons  is-path.simps(2))
      qed
    moreover have valid-graph.is-path (del-unEdge v w v' G) n nvs' n'
      using is-path-nvs' nvs'
      proof (induct nvs' arbitrary:n nvs)
        case Nil
      thus ?case by (metis del-UnEdge-node is-path.simps(1) valid0 valid-graph.is-path-simps(1))
      next
        case (Cons x xs)
        obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
        hence is-path x3 xs n' using Cons by auto
        moreover have xs = takeWhile (λx. x ≠ (v, w, v') ∧ x ≠ (v', w, v)) (tl
nvs)
          using ‹x # xs = takeWhile (λx. x ≠ (v, w, v') ∧ x ≠ (v', w, v)) nvs›
      by (metis (lifting, no-types) append-Cons list.distinct(1) takeWhile.simps(2)

              takeWhile-dropWhile-id list.sel(3))
        ultimately have valid-graph.is-path (del-unEdge v w v' G) x3 xs n'
          using Cons by auto
        moreover have x≠(v,w,v') ∧ x≠(v',w,v)
          using Cons(3) set-takeWhileD[of x (λx. x ≠ (v, w, v') ∧ x ≠ (v', w,
v)) nvs]
          by (metis List.set-simps(2) insertI1)
        hence x∈edges (del-unEdge v w v' G)
          by (metis Cons.prems(1) del-UnEdge-frame is-path.simps(2) x)
        ultimately show ?case using x
      by (metis Cons.prems(1) is-path.simps(2) valid0 valid-graph.is-path.simps(2))
      qed
    ultimately show False using n-neg-v n-neg-v' by auto
  qed
moreover have nodes G1 ∩ nodes G2={}
  proof (rule ccontr)
    assume nodes G1 ∩ nodes G2 ≠ {}
    then obtain n where n:n∈nodes G1 n∈nodes G2 by auto
    then obtain nvs nv's where
        nvs  : valid-graph.is-path (del-unEdge v w v' G) n nvs v and
        nv's : valid-graph.is-path (del-unEdge v w v' G) n nv's v'
      using G1 G2 G1-nodes G2-nodes by auto
    hence valid-graph.is-path (del-unEdge v w v' G) v ((rev-path nvs)@nv's) v'
     using valid-unMultigraph.is-path-rev[OF valid0'] valid-graph.is-path-split[OF
```

valid0]
by *auto*
hence *valid-unMultigraph.connected* (*del-unEdge v w v′ G*)
by (*metis assms*(*1*) *del-unEdge-connectivity*)
thus *False* **by** (*metis assms*(*2*))
qed
moreover have *edges G1* ∪ *edges G2* = *edges* (*del-unEdge v w v′ G*)
proof (*rule ccontr*)
assume *edges G1* ∪ *edges G2* ≠ *edges* (*del-unEdge v w v′ G*)
moreover have *edges G1* ⊆ *edges* (*del-unEdge v w v′ G*) **using** *G1* **by** *auto*
moreover have *edges G2* ⊆ *edges* (*del-unEdge v w v′ G*) **using** *G2* **by** *auto*
ultimately obtain *n e n′* **where**
nen′:
(*n*,*e*,*n′*)∈*edges* (*del-unEdge v w v′ G*)
(*n*,*e*,*n′*)∉*edges G1* (*n*,*e*,*n′*)∉*edges G2*
by *auto*
moreover have *n*∈*nodes* (*del-unEdge v w v′ G*)
by (*metis nen′*(*1*) *valid0 valid-graph.E-validD*(*1*))
moreover have *n′*∈*nodes* (*del-unEdge v w v′ G*)
by (*metis nen′*(*1*) *valid0 valid-graph.E-validD*(*2*))
ultimately have (*n*∈*nodes G1* ∧ *n′*∈*nodes G2*)∨(*n*∈*nodes G2*∧*n′*∈*nodes G1*)
using *G1 G2* ‹*nodes G1* ∪ *nodes G2*=*nodes* (*del-unEdge v w v′ G*)› **by** *auto*
moreover have *n*∈*nodes G1* ⟹ *n′*∈*nodes G2* ⟹ *False*
proof −
assume *n*∈*nodes G1 n′*∈*nodes G2*
then obtain *nvs nv′s* **where**
nvs : *valid-graph.is-path* (*del-unEdge v w v′ G*) *n nvs v* **and**
nv′s : *valid-graph.is-path* (*del-unEdge v w v′ G*) *n′ nv′s v′*
using *G1 G2 G1-nodes G2-nodes* **by** *auto*
hence *valid-graph.is-path* (*del-unEdge v w v′ G*) *v*
((*rev-path nvs*)@(*n*,*e*,*n′*)#*nv′s*) *v′*
using *valid-unMultigraph.is-path-rev*[*OF valid0′*] *valid-graph.is-path-split′*[*OF valid0*]
‹(*n*,*e*,*n′*)∈*edges* (*del-unEdge v w v′ G*)›
by *auto*
hence *valid-unMultigraph.connected* (*del-unEdge v w v′ G*)
by (*metis assms*(*1*) *del-unEdge-connectivity*)
thus *False* **by** (*metis assms*(*2*))
qed
moreover have *n*∈*nodes G2* ⟹ *n′*∈*nodes G1* ⟹ *False*
proof −
assume *n′*∈*nodes G1 n*∈*nodes G2*
then obtain *n′vs nvs* **where**
n′vs : *valid-graph.is-path* (*del-unEdge v w v′ G*) *n′ n′vs v* **and**
nvs : *valid-graph.is-path* (*del-unEdge v w v′ G*) *n nvs v′*
using *G1 G2 G1-nodes G2-nodes* **by** *auto*
moreover have (*n′*,*e*,*n*)∈*edges* (*del-unEdge v w v′ G*)
by (*metis nen′*(*1*) *valid0′ valid-unMultigraph.corres*)

```
          ultimately have valid-graph.is-path (del-unEdge v w v' G) v
                ((rev-path n'vs)@(n',e,n)#nvs) v'
         using valid-unMultigraph.is-path-rev[OF valid0'] valid-graph.is-path-split'[OF
valid0]
           by auto
          hence valid-unMultigraph.connected (del-unEdge v w v' G)
            by (metis assms(1) del-unEdge-connectivity)
          thus False by (metis assms(2))
        qed
      ultimately show False by auto
    qed
  moreover have edges G1 ∩ edges G2={}
    proof (rule ccontr)
      assume edges G1 ∩ edges G2 ≠ {}
      then obtain n e n' where (n,e,n')∈edges G1 (n,e,n')∈edges G2 by auto
      hence n∈nodes G1 n∈nodes G2 using G1 G2 by auto
      thus False using ‹nodes G1 ∩ nodes G2={}› by auto
    qed
  moreover have valid-unMultigraph.connected G1
    unfolding valid-unMultigraph.connected-def[OF valid-G1]
    proof (rule,rule,rule)
      fix n n'
      assume n : n ∈nodes G1
      assume n': n'∈nodes G1
      assume n≠n'
      obtain ps where valid-graph.is-path (del-unEdge v w v' G) n ps v
        using G1 G1-nodes n by auto
      hence ps:valid-graph.is-path G1 n ps v
        proof (induct ps arbitrary:n)
          case Nil
          moreover have v∈nodes G1 using G1 G1-nodes valid0
            by (metis (lifting, no-types) calculation mem-Collect-eq select-convs(1)
                valid-graph.is-path.simps(1))
          ultimately show ?case
            by (metis valid0 valid-G1 valid-unMultigraph.is-trail.simps(1)
                 valid-graph.is-path.simps(1)  valid-unMultigraph.is-trail-intro)
        next
          case (Cons x xs)
          obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
          have x1∈nodes G1 using G1 G1-nodes Cons.prems x
         by (metis (lifting) mem-Collect-eq select-convs(1) valid0 valid-graph.is-path.simps(2))
          moreover have (x1,x2,x3)∈edges (del-unEdge v w v' G)
            by (metis Cons.prems valid0 valid-graph.is-path.simps(2) x)
          ultimately have (x1,x2,x3)∈edges G1
            using G1 G2 ‹nodes G1 ∩ nodes G2={}› ‹edges G1 ∪ edges G2=edges
(del-unEdge v w v' G)›
         by (metis (full-types) IntI Un-iff bex-empty  valid-G2' valid-graph.E-validD(1)
)
          moreover have valid-graph.is-path (del-unEdge v w v' G) x3 xs v
```

by (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*) *x*)
hence *valid-graph.is-path G1 x3 xs v* **using** *Cons.hyps* **by** *auto*
moreover have *x1=n* **by** (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*)
x)
ultimately show *?case* **using** *x valid-G1′* **by** (*metis valid-graph.is-path.simps*(*2*))

qed
obtain *ps′* **where** *valid-graph.is-path* (*del-unEdge v w v′ G*) *n′ ps′ v*
using *G1 G1-nodes n′* **by** *auto*
hence *ps′*:*valid-graph.is-path G1 n′ ps′ v*
proof (*induct ps′ arbitrary*:*n′*)
case *Nil*
moreover have *v*∈*nodes G1* **using** *G1 G1-nodes valid0*
by (*metis* (*lifting*, *no-types*) *calculation mem-Collect-eq select-convs*(*1*)
valid-graph.is-path.simps(*1*))
ultimately show *?case*
by (*metis valid0 valid-G1 valid-unMultigraph.is-trail.simps*(*1*)
valid-graph.is-path.simps(*1*) *valid-unMultigraph.is-trail-intro*)
next
case (*Cons x xs*)
obtain *x1 x2 x3* **where** *x*:*x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
have *x1*∈*nodes G1* **using** *G1 G1-nodes Cons.prems x*
by (*metis* (*lifting*) *mem-Collect-eq select-convs*(*1*) *valid0 valid-graph.is-path.simps*(*2*))
moreover have (*x1*,*x2*,*x3*)∈*edges* (*del-unEdge v w v′ G*)
by (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*) *x*)
ultimately have (*x1*,*x2*,*x3*)∈*edges G1*
using *G1 G2* ‹*nodes G1* ∩ *nodes G2*={}›
‹*edges G1* ∪ *edges G2*=*edges* (*del-unEdge v w v′ G*)›
by (*metis* (*full-types*) *IntI Un-iff bex-empty valid-G2′ valid-graph.E-validD*(*1*))
moreover have *valid-graph.is-path* (*del-unEdge v w v′ G*) *x3 xs v*
by (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*) *x*)
hence *valid-graph.is-path G1 x3 xs v* **using** *Cons.hyps* **by** *auto*
moreover have *x1=n′* **by** (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*)
x)
ultimately show *?case* **using** *x valid-G1′* **by** (*metis valid-graph.is-path.simps*(*2*))

qed
hence *valid-graph.is-path G1 v* (*rev-path ps′*) *n′*
using *valid-unMultigraph.is-path-rev*[*OF valid-G1*]
by *auto*
hence *valid-graph.is-path G1 n* (*ps*@(*rev-path ps′*)) *n′*
using *ps valid-graph.is-path-split*[*OF valid-G1′*,*of n ps rev-path ps′ n′*]
by *auto*
thus ∃*ps. valid-graph.is-path G1 n ps n′* **by** *auto*
qed
moreover have *valid-unMultigraph.connected G2*
unfolding *valid-unMultigraph.connected-def*[*OF valid-G2*]
proof (*rule*,*rule*,*rule*)
fix *n n′*

```
assume n : n ∈nodes G2
assume n': n'∈nodes G2
assume n≠n'
obtain ps where valid-graph.is-path (del-unEdge v w v' G) n ps v'
  using G2 G2-nodes n by auto
hence ps:valid-graph.is-path G2 n ps v'
  proof (induct ps arbitrary:n)
    case Nil
    moreover have v'∈nodes G2 using G2 G2-nodes valid0
      by (metis (lifting, no-types) calculation mem-Collect-eq select-convs(1)
          valid-graph.is-path.simps(1))
    ultimately show ?case
      by (metis valid0 valid-G2 valid-unMultigraph.is-trail.simps(1)
           valid-graph.is-path.simps(1)  valid-unMultigraph.is-trail-intro)
  next
    case (Cons x xs)
    obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
    have x1∈nodes G2 using G2 G2-nodes Cons.prems x
   by (metis (lifting) mem-Collect-eq select-convs(1) valid0 valid-graph.is-path.simps(2))
    moreover have (x1,x2,x3)∈edges (del-unEdge v w v' G)
      by (metis Cons.prems valid0 valid-graph.is-path.simps(2) x)
    ultimately have (x1,x2,x3)∈edges G2
   using ‹nodes G1 ∩ nodes G2={}› ‹edges G1 ∪ edges G2=edges (del-unEdge
v w v' G)›
     by (metis IntI Un-iff assms(1) bex-empty connected-def del-UnEdge-node
valid0 valid0'
    valid-G1' valid-graph.E-validD(1) valid-graph.E-validD(2) valid-unMultigraph.no-id)
    moreover have valid-graph.is-path (del-unEdge v w v' G) x3 xs v'
      by (metis Cons.prems valid0 valid-graph.is-path.simps(2) x)
    hence valid-graph.is-path G2 x3 xs v' using Cons.hyps by auto
  moreover have x1=n by (metis Cons.prems valid0 valid-graph.is-path.simps(2)
x)
  ultimately show ?case using x valid-G2' by (metis valid-graph.is-path.simps(2))

  qed
obtain ps' where valid-graph.is-path (del-unEdge v w v' G) n' ps' v'
  using G2 G2-nodes n' by auto
hence ps':valid-graph.is-path G2 n' ps' v'
  proof (induct ps' arbitrary:n')
    case Nil
    moreover have v'∈nodes G2 using G2 G2-nodes valid0
      by (metis (lifting, no-types) calculation mem-Collect-eq select-convs(1)
          valid-graph.is-path.simps(1))
    ultimately show ?case
      by (metis valid0 valid-G2 valid-unMultigraph.is-trail.simps(1)
           valid-graph.is-path.simps(1)  valid-unMultigraph.is-trail-intro)
  next
    case (Cons x xs)
    obtain x1 x2 x3 where x:x=(x1,x2,x3) by (metis prod-cases3)
```

have *x1∈nodes G2* **using** *G2 G2-nodes Cons.prems x*
by (*metis* (*lifting*) *mem-Collect-eq select-convs*(*1*) *valid0 valid-graph.is-path.simps*(*2*))
moreover have (*x1*,*x2*,*x3*)∈*edges* (*del-unEdge v w v′ G*)
by (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*) *x*)
ultimately have (*x1*,*x2*,*x3*)∈*edges G2*
using ‹*nodes G1* ∩ *nodes G2*={}› ‹*edges G1* ∪ *edges G2*=*edges* (*del-unEdge v w v′ G*)›
by (*metis IntI Un-iff assms*(*1*) *bex-empty connected-def del-UnEdge-node valid0 valid0′*
valid-G1′ valid-graph.E-validD(*1*) *valid-graph.E-validD*(*2*) *valid-unMultigraph.no-id*)
moreover have *valid-graph.is-path* (*del-unEdge v w v′ G*) *x3 xs v′*
by (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*) *x*)
hence *valid-graph.is-path G2 x3 xs v′* **using** *Cons.hyps* **by** *auto*
moreover have *x1*=*n′* **by** (*metis Cons.prems valid0 valid-graph.is-path.simps*(*2*) *x*)
ultimately show *?case* **using** *x valid-G2′* **by** (*metis valid-graph.is-path.simps*(*2*))

qed
hence *valid-graph.is-path G2 v′* (*rev-path ps′*) *n′*
using *valid-unMultigraph.is-path-rev*[*OF valid-G2*]
by *auto*
hence *valid-graph.is-path G2 n* (*ps*@(*rev-path ps′*)) *n′*
using *ps valid-graph.is-path-split*[*OF valid-G2′*,*of n ps rev-path ps′ n′*]
by *auto*
thus ∃*ps*. *valid-graph.is-path G2 n ps n′* **by** *auto*
qed
ultimately show *?thesis* **using** *valid-G1 valid-G2 that* **by** *auto*
qed

lemma *sub-graph-degree-frame*:
assumes *valid-graph G2 edges G1* ∪ *edges G2* =*edges G nodes G1* ∩ *nodes G2*={} *n*∈*nodes G1*
shows *degree n G*=*degree n G1*
proof −
have {*e* ∈ *edges G*. *fst e* = *n*}⊆{*e* ∈ *edges G1*. *fst e* = *n*}
proof
fix *e* **assume** *e* ∈ {*e* ∈ *edges G*. *fst e* = *n*}
hence *e*∈*edges G fst e*=*n* **by** *auto*
moreover have *n*∉*nodes G2*
using ‹*nodes G1* ∩ *nodes G2*={}› ‹*n*∈*nodes G1*›
by *auto*
hence *e*∉*edges G2* **using** *valid-graph.E-validD*[*OF* ‹*valid-graph G2*›] ‹*fst e*=*n*›

by (*metis prod.exhaust fst-conv*)
ultimately have *e*∈*edges G1* **using** ‹*edges G1* ∪ *edges G2* =*edges G*› **by** *auto*
thus *e* ∈ {*e* ∈ *edges G1*. *fst e* = *n*} **using** ‹*fst e*=*n*› **by** *auto*
qed

moreover have $\{e \in \mathit{edges}\ G1.\ \mathit{fst}\ e = n\} \subseteq \{e \in \mathit{edges}\ G.\ \mathit{fst}\ e = n\}$
by (*metis* (*lifting*) *Collect-mono Un-iff assms*(*2*))
ultimately show *?thesis* **unfolding** *degree-def* **by** *auto*
qed

lemma *odd-nodes-no-edge*[*simp*]: *finite* (*nodes g*) $\Longrightarrow$ *num-of-odd-nodes* (*g* (|*edges*:={}
|)) = *0*
unfolding *num-of-odd-nodes-def odd-nodes-set-def degree-def* **by** *simp*

4 Adjacent nodes

definition (**in** *valid-unMultigraph*) *adjacent*:: $'v \Rightarrow 'v \Rightarrow$ *bool* **where**
adjacent v v' $\equiv \exists\, w.\ (v,w,v') \in E$

lemma (**in** *valid-unMultigraph*) *adjacent-sym*: *adjacent v v'* $\longleftrightarrow$ *adjacent v' v*
unfolding *adjacent-def* **by** *auto*

lemma (**in** *valid-unMultigraph*) *adjacent-no-loop*[*simp*]: *adjacent v v'* $\Longrightarrow v \neq v'$
unfolding *adjacent-def* **by** *auto*

lemma (**in** *valid-unMultigraph*) *adjacent-V*[*simp*]:
assumes *adjacent v v'*
shows $v \in V$ $v' \in V$
using *assms E-validD* **unfolding** *adjacent-def* **by** *auto*

lemma (**in** *valid-unMultigraph*) *adjacent-finite*:
finite E $\Longrightarrow$ *finite* $\{n.$ *adjacent v n*$\}$
proof −
assume *finite E*
{ fix *S v*
have *finite S* $\Longrightarrow$ *finite* $\{n.\ \exists\, w.\ (v,w,n) \in S\}$
proof (*induct S rule*: *finite-induct*)
case *empty*
thus *?case* **by** *auto*
next
case (*insert x F*)
obtain *x1 x2 x3* **where** *x*: *x*=(*x1*,*x2*,*x3*) **by** (*metis prod-cases3*)
have *x1*=*v* $\Longrightarrow$ *?case*
proof −
assume *x1*=*v*
hence $\{n.\ \exists\, w.\ (v,\ w,\ n) \in$ *insert x F*$\}$=*insert x3* $\{n.\ \exists\, w.\ (v,\ w,\ n) \in$
F$\}$
using *x* **by** *auto*
thus *?thesis* **using** *insert* **by** *auto*
qed
moreover have $x1 \neq v \Longrightarrow$ *?case*
proof −
assume $x1 \neq v$

hence $\{n. \exists w. (v, w, n) \in$ *insert x F*$\}=\{n. \exists w. (v, w, n) \in F\}$ **using** *x* **by** *auto*
thus *?thesis* **using** *insert* **by** *auto*
qed
ultimately show *?case* **by** *auto*
qed }
note *aux=this*
show *?thesis* **using** *aux*[*OF* ‹*finite E*›, *of v*] **unfolding** *adjacent-def* **by** *auto*
qed

5 Undirected simple graph

locale *valid-unSimpGraph=valid-unMultigraph G* **for** *G*::('*v*,'*w*) *graph*+
assumes *no-multi*[*simp*]: $(v,w,u) \in$ *edges G* $\Longrightarrow (v,w',u) \in$*edges G* $\Longrightarrow w = w'$

lemma (**in** *valid-unSimpGraph*) *finV-to-finE*[*simp*]:
assumes *finite V*
shows *finite E*
proof (*cases* $\{(v1,v2).$ *adjacent v1 v2*$\}=\{\}$)
case *True*
hence $E=\{\}$ **unfolding** *adjacent-def* **by** *auto*
thus *finite E* **by** *auto*
next
case *False*
have $\{(v1,v2).$ *adjacent v1 v2*$\} \subseteq V \times V$ **using** *adjacent-V* **by** *auto*
moreover have *finite* $(V \times V)$ **using** ‹*finite V*› **by** *auto*
ultimately have *finite* $\{(v1,v2).$ *adjacent v1 v2*$\}$ **using** *finite-subset* **by** *auto*
hence *card* $\{(v1,v2).$ *adjacent v1 v2*$\}\neq 0$ **using** *False card-eq-0-iff* **by** *auto*
moreover have *card E=card* $\{(v1,v2).$ *adjacent v1 v2*$\}$
proof −
have $(\lambda(v1,w,v2). (v1,v2))$‘$E = \{(v1,v2).$ *adjacent v1 v2*$\}$
proof −
have $\bigwedge x. x\in(\lambda(v1,w,v2). (v1,v2))$‘$E \Longrightarrow x\in \{(v1,v2).$ *adjacent v1 v2*$\}$
unfolding *adjacent-def* **by** *auto*
moreover have $\bigwedge x. x\in\{(v1,v2).$ *adjacent v1 v2*$\} \Longrightarrow x\in(\lambda(v1,w,v2). (v1,v2))$‘$E$
unfolding *adjacent-def* **by** *force*
ultimately show *?thesis* **by** *force*
qed
moreover have *inj-on* $(\lambda(v1,w,v2). (v1,v2))$ *E* **unfolding** *inj-on-def* **by** *auto*
ultimately show *?thesis* **by** (*metis card-image*)
qed
ultimately show *finite E* **by** (*metis card-infinite*)
qed

lemma *del-unEdge-valid'*[*simp*]:*valid-unSimpGraph G*$\Longrightarrow$

```
    valid-unSimpGraph (del-unEdge v w u G)
proof -
  assume valid-unSimpGraph G
  hence valid-unMultigraph (del-unEdge v w u G)
    using valid-unSimpGraph-def[of G] del-unEdge-valid[of G] by auto
  moreover have valid-unSimpGraph-axioms (del-unEdge v w u G)
    using valid-unSimpGraph.no-multi[OF ‹valid-unSimpGraph G›]
    unfolding valid-unSimpGraph-axioms-def del-unEdge-def by auto
  ultimately show valid-unSimpGraph (del-unEdge v w u G) using valid-unSimpGraph-def
    by auto
qed

lemma (in valid-unSimpGraph) del-UnEdge-non-adj:
    (v,w,u)∈E ⟹ ¬valid-unMultigraph.adjacent (del-unEdge v w u G) v u
proof
  assume (v, w, u) ∈ E
      and ccontr:valid-unMultigraph.adjacent (del-unEdge v w u G) v u
  have valid:valid-unMultigraph (del-unEdge v w u G)
    using valid-unMultigraph-axioms by auto
  then obtain w' where vw'u:(v,w',u)∈edges (del-unEdge v w u G)
    using ccontr unfolding valid-unMultigraph.adjacent-def[OF valid] by auto
  hence (v,w',u)∉{(v,w,u),(u,w,v)} unfolding del-unEdge-def by auto
  hence w'≠w by auto
  moreover have (v,w',u)∈E using vw'u unfolding del-unEdge-def by auto
  ultimately show False using no-multi[of v w u w'] ‹(v, w, u) ∈ E› by auto
qed

lemma (in valid-unSimpGraph) degree-adjacent: finite E ⟹ degree v G=card {n.
adjacent v n}
  using valid-unSimpGraph-axioms
proof (induct degree v G arbitrary: G)
  case 0
  note valid3=‹valid-unSimpGraph G›
  hence valid2: valid-unMultigraph G using valid-unSimpGraph-def by auto
  have {a. valid-unMultigraph.adjacent G v a}={}
    proof (rule ccontr)
      assume {a. valid-unMultigraph.adjacent G v a} ≠ {}
      then obtain w u where (v,w,u)∈edges G
        unfolding valid-unMultigraph.adjacent-def[OF valid2] by auto
      hence degree v G≠0 using ‹finite (edges G)› unfolding degree-def by auto
      thus False using ‹0 = degree v G› by auto
    qed
  thus ?case by (metis 0.hyps card-empty)
next
  case (Suc n)
  hence {e ∈ edges G. fst e = v}≠{} using card-empty unfolding degree-def by
force
  then obtain w u where (v,w,u)∈edges G by auto
  have valid:valid-unMultigraph G using ‹valid-unSimpGraph G› valid-unSimpGraph-def
```

by *auto*
hence *valid'*:*valid-unMultigraph* (*del-unEdge v w u G*) **by** *auto*
have *valid-unSimpGraph* (*del-unEdge v w u G*)
using *del-unEdge-valid'* ‹*valid-unSimpGraph G*› **by** *auto*
moreover have *n* = *degree v* (*del-unEdge v w u G*)
using ‹*Suc n* = *degree v G*›‹(*v*, *w*, *u*) ∈ *edges G*› *del-edge-undirected-degree-plus*[*of G v w u*]
by (*metis Suc.prems*(*1*) *Suc-eq-plus1 diff-Suc-1 valid valid-unMultigraph.corres*)
moreover have *finite* (*edges* (*del-unEdge v w u G*))
using ‹*finite* (*edges G*)› **unfolding** *del-unEdge-def*
by *auto*
ultimately have *degree v* (*del-unEdge v w u G*)
= *card* (*Collect* (*valid-unMultigraph.adjacent* (*del-unEdge v w u G*) *v*))
using *Suc.hyps* **by** *auto*
moreover have *Suc*(*card* ({*n*. *valid-unMultigraph.adjacent* (*del-unEdge v w u G*)
v n})) = *card* ({*n*. *valid-unMultigraph.adjacent G v n*})
using *valid-unMultigraph.adjacent-def*[*OF valid'*]
proof −
have {*n*. *valid-unMultigraph.adjacent* (*del-unEdge v w u G*) *v n*} ⊆
{*n*. *valid-unMultigraph.adjacent G v n*}
using *del-unEdge-def*[*of v w u G*]
unfolding *valid-unMultigraph.adjacent-def*[*OF valid'*]
valid-unMultigraph.adjacent-def[*OF valid*]
by *auto*
moreover have *u*∈{*n*. *valid-unMultigraph.adjacent G v n*}
using ‹(*v*,*w*,*u*)∈*edges G*› **unfolding** *valid-unMultigraph.adjacent-def*[*OF valid*] **by** *auto*
ultimately have {*n*. *valid-unMultigraph.adjacent* (*del-unEdge v w u G*) *v n*} ∪ {*u*}
⊆ {*n*. *valid-unMultigraph.adjacent G v n*} **by** *auto*
moreover have {*n*. *valid-unMultigraph.adjacent G v n*} − {*u*}
⊆ {*n*. *valid-unMultigraph.adjacent* (*del-unEdge v w u G*) *v n*}
using *del-unEdge-def*[*of v w u G*]
unfolding *valid-unMultigraph.adjacent-def*[*OF valid'*]
valid-unMultigraph.adjacent-def[*OF valid*]
by *auto*
ultimately have {*n*. *valid-unMultigraph.adjacent* (*del-unEdge v w u G*) *v n*} ∪ {*u*}
= {*n*. *valid-unMultigraph.adjacent G v n*} **by** *auto*
moreover have *u*∉{*n*. *valid-unMultigraph.adjacent* (*del-unEdge v w u G*) *v n*}
using *valid-unSimpGraph.del-UnEdge-non-adj*[*OF* ‹*valid-unSimpGraph G*› ‹(*v*,*w*,*u*)∈*edges G*›]
by *auto*
moreover have *finite* {*n*. *valid-unMultigraph.adjacent G v n*}
using *valid-unMultigraph.adjacent-finite*[*OF valid* ‹*finite* (*edges G*)›] **by** *simp*

ultimately show *?thesis*

by (*metis Un-insert-right card-insert-disjoint finite-Un sup-bot-right*)
qed
ultimately show *?case* **by** (*metis Suc.hyps*(*2*) ‹*n* = *degree v* (*del-unEdge v w u G*)›)
qed

end

theory *KoenigsbergBridge* **imports** *MoreGraph*
begin

6 Definition of Eulerian trails and circuits

definition (**in** *valid-unMultigraph*) *is-Eulerian-trail*:: $'v \Rightarrow ('v,'w)$ *path* $\Rightarrow 'v \Rightarrow$ *bool*
where
is-Eulerian-trail v ps v$'\equiv$ *is-trail v ps v*$'$ $\wedge$ *edges* (*rem-unPath ps G*) = {}

definition (**in** *valid-unMultigraph*) *is-Eulerian-circuit*:: $'v \Rightarrow ('v,'w)$ *path* $\Rightarrow 'v \Rightarrow$ *bool* **where**
is-Eulerian-circuit v ps v$'\equiv$ $(v=v')$ $\wedge$ (*is-Eulerian-trail v ps v*$'$)

7 Necessary conditions for Eulerian trails and circuits

lemma (**in** *valid-unMultigraph*) *euclerian-rev*:
is-Eulerian-trail v$'$ (*rev-path ps*) *v*=*is-Eulerian-trail v ps v*$'$
proof −
have *is-trail v*$'$ (*rev-path ps*) *v*=*is-trail v ps v*$'$
by (*metis is-trail-rev*)
moreover have *edges* (*rem-unPath* (*rev-path ps*) *G*)=*edges* (*rem-unPath ps G*)
by (*metis rem-unPath-graph*)
ultimately show *?thesis* **unfolding** *is-Eulerian-trail-def* **by** *auto*
qed

theorem (**in** *valid-unMultigraph*) *euclerian-cycle-ex*:
assumes *is-Eulerian-circuit v ps v*$'$ *finite V finite E*
shows $\forall v\in V.$ *even* (*degree v G*)
proof −
obtain *v ps v*$'$ **where** *cycle*:*is-Eulerian-circuit v ps v*$'$ **using** *assms* **by** *auto*
hence *edges* (*rem-unPath ps G*) = {}
unfolding *is-Eulerian-circuit-def is-Eulerian-trail-def*
by *simp*
moreover have *nodes* (*rem-unPath ps G*)=*nodes G* **by** *auto*
ultimately have *rem-unPath ps G* = *G* (|*edges*:={}|) **by** *auto*
hence *num-of-odd-nodes* (*rem-unPath ps G*) = *0* **by** (*metis assms*(*2*) *odd-nodes-no-edge*)
moreover have *v*=*v*$'$

by (*metis* ‹*is-Eulerian-circuit v ps v'*› *is-Eulerian-circuit-def*)
hence *num-of-odd-nodes* (*rem-unPath ps G*)=*num-of-odd-nodes G*
by (*metis assms*(*2*) *assms*(*3*) *cycle is-Eulerian-circuit-def*
is-Eulerian-trail-def rem-UnPath-cycle)
ultimately have *num-of-odd-nodes G*=*0* **by** *auto*
moreover have *finite*(*odd-nodes-set G*)
using ‹*finite V*› **unfolding** *odd-nodes-set-def* **by** *auto*
ultimately have *odd-nodes-set G* = {} **unfolding** *num-of-odd-nodes-def* **by** *auto*
thus *?thesis* **unfolding** *odd-nodes-set-def* **by** *auto*
qed

theorem (**in** *valid-unMultigraph*) *euclerian-path-ex*:
assumes *is-Eulerian-trail v ps v' finite V finite E*
shows (∀ *v*∈*V*. *even* (*degree v G*)) ∨ (*num-of-odd-nodes G* =*2*)
proof −
obtain *v ps v'* **where** *path*:*is-Eulerian-trail v ps v'* **using** *assms* **by** *auto*
hence *edges* (*rem-unPath ps G*) = {}
unfolding *is-Eulerian-trail-def*
by *simp*
moreover have *nodes* (*rem-unPath ps G*)=*nodes G* **by** *auto*
ultimately have *rem-unPath ps G* = *G* (|*edges*:={}|) **by** *auto*
hence *odd-nodes*: *num-of-odd-nodes* (*rem-unPath ps G*) = *0*
by (*metis assms*(*2*) *odd-nodes-no-edge*)
have *v*≠*v'* ⟹ *?thesis*
proof (*cases even*(*degree v' G*))
case *True*
assume *v*≠*v'*
have *is-trail v ps v'* **by** (*metis is-Eulerian-trail-def path*)
hence *num-of-odd-nodes* (*rem-unPath ps G*) = *num-of-odd-nodes G*
+ (*if even* (*degree v G*) *then 2 else 0*)
using *rem-UnPath-even True* ‹*finite V*› ‹*finite E*› ‹*v*≠*v'*› **by** *auto*
hence *num-of-odd-nodes G* + (*if even* (*degree v G*) *then 2 else 0*)=*0*
using *odd-nodes* **by** *auto*
hence *num-of-odd-nodes G* = *0* **by** *auto*
moreover have *finite*(*odd-nodes-set G*)
using ‹*finite V*› **unfolding** *odd-nodes-set-def* **by** *auto*
ultimately have *odd-nodes-set G* = {} **unfolding** *num-of-odd-nodes-def* **by** *auto*
thus *?thesis* **unfolding** *odd-nodes-set-def* **by** *auto*
next
case *False*
assume *v*≠*v'*
have *is-trail v ps v'* **by** (*metis is-Eulerian-trail-def path*)
hence *num-of-odd-nodes* (*rem-unPath ps G*) = *num-of-odd-nodes G*
+ (*if odd* (*degree v G*) *then* −*2 else 0*)
using *rem-UnPath-odd False* ‹*finite V*› ‹*finite E*› ‹*v*≠*v'*› **by** *auto*
hence *odd-nodes-if*: *num-of-odd-nodes G* + (*if odd* (*degree v G*) *then* −*2 else*

0)=0
    using *odd-nodes* **by** *auto*
  have *odd* (*degree v G*) $\Longrightarrow$ *?thesis*
    proof −
      assume *odd* (*degree v G*)
      hence *num-of-odd-nodes G* = *2* **using** *odd-nodes-if* **by** *auto*
      thus *?thesis* **by** *simp*
    qed
  moreover have *even*(*degree v G*) $\Longrightarrow$ *?thesis*
    proof −
      assume *even* (*degree v G*)
      hence *num-of-odd-nodes G* = *0* **using** *odd-nodes-if* **by** *auto*
      moreover have *finite*(*odd-nodes-set G*)
        using ‹*finite V*› **unfolding** *odd-nodes-set-def* **by** *auto*
      ultimately have *odd-nodes-set G* = {} **unfolding** *num-of-odd-nodes-def*
by *auto*
      thus *?thesis* **unfolding** *odd-nodes-set-def* **by** *auto*
    qed
  ultimately show *?thesis* **by** *auto*
qed
moreover have *v*=*v'*$\Longrightarrow$ *?thesis*
  by (*metis assms*(*2*) *assms*(*3*) *euclerian-cycle-ex is-Eulerian-circuit-def path*)
ultimately show *?thesis* **by** *auto*
qed

8 Specific case of the Konigsberg Bridge Problem

datatype *kon-node* = *a* | *b* | *c* | *d*

datatype *kon-bridge* = *ab1* | *ab2* | *ac1* | *ac2* | *ad1* | *bd1* | *cd1*

definition *kon-graph* :: (*kon-node*,*kon-bridge*) *graph* **where**
kon-graph≡(|*nodes*={*a*,*b*,*c*,*d*},
    edges={(*a*,*ab1*,*b*), (*b*,*ab1*,*a*),
      (*a*,*ab2*,*b*), (*b*,*ab2*,*a*),
      (*a*,*ac1*,*c*), (*c*,*ac1*,*a*),
      (*a*,*ac2*,*c*), (*c*,*ac2*,*a*),
      (*a*,*ad1*,*d*), (*d*,*ad1*,*a*),
      (*b*,*bd1*,*d*), (*d*,*bd1*,*b*),
      (*c*,*cd1*,*d*), (*d*,*cd1*,*c*)} |)

instantiation *kon-node* :: *enum*
begin
definition [*simp*]: *enum-class.enum* =[*a*,*b*,*c*,*d*]
definition [*simp*]: *enum-class.enum-all P* $\longleftrightarrow$ *P a* ∧ *P b* ∧ *P c* ∧ *P d*
definition [*simp*]:*enum-class.enum-ex P* $\longleftrightarrow$ *P a* ∨ *P b* ∨ *P c* ∨ *P d*
instance proof qed (*auto*,(*case-tac x*,*auto*)+)
end

instantiation *kon-bridge* :: *enum*
begin
definition [*simp*]:*enum-class.enum* =[*ab1*,*ab2*,*ac1*,*ac2*,*ad1*,*cd1*,*bd1*]
definition [*simp*]:*enum-class.enum-all* $P \longleftrightarrow P\ ab1 \wedge P\ ab2 \wedge P\ ac1 \wedge P\ ac2 \wedge P\ ad1 \wedge P\ bd1$
$\wedge\ P\ cd1$
definition [*simp*]:*enum-class.enum-ex* $P \longleftrightarrow P\ ab1 \vee P\ ab2 \vee P\ ac1 \vee P\ ac2 \vee P\ ad1 \vee P\ bd1$
$\vee\ P\ cd1$
instance proof qed (*auto*,(*case-tac x*,*auto*)+)
end

interpretation *kon-graph*: *valid-unMultigraph kon-graph*
proof (*unfold-locales*)
  show *fst* ' *edges kon-graph* $\subseteq$ *nodes kon-graph* **by** *eval*
next
  show *snd* ' *snd* ' *edges kon-graph* $\subseteq$ *nodes kon-graph* **by** *eval*
next
  have $\forall v\ w\ u'.\ ((v,\ w,\ u') \in$ *edges kon-graph*$) = ((u',\ w,\ v) \in$ *edges kon-graph*$)$
    by *eval*
  thus $\bigwedge v\ w\ u'.\ ((v,\ w,\ u') \in$ *edges kon-graph*$) = ((u',\ w,\ v) \in$ *edges kon-graph*$)$ **by** *simp*
next
  have $\forall v\ w.\ (v,\ w,\ v) \notin$ *edges kon-graph* **by** *eval*
  thus $\bigwedge v\ w.\ (v,\ w,\ v) \notin$ *edges kon-graph* **by** *simp*
qed

theorem $\neg$*kon-graph.is-Eulerian-trail v1 p v2*
proof
  assume *kon-graph.is-Eulerian-trail v1 p v2*
  moreover have *finite* (*nodes kon-graph*) **by** (*metis finite-code*)
  moreover have *finite* (*edges kon-graph*) **by** (*metis finite-code*)
  ultimately have *contra*:
   ($\forall v \in$*nodes kon-graph*. *even* (*degree v kon-graph*)) $\vee$(*num-of-odd-nodes kon-graph* =*2*)
    by (*metis kon-graph.euclerian-path-ex*)
  have *odd*(*degree a kon-graph*) **by** *eval*
  moreover have *odd*(*degree b kon-graph*) **by** *eval*
  moreover have *odd*(*degree c kon-graph*) **by** *eval*
  moreover have *odd*(*degree d kon-graph*) **by** *eval*
  ultimately have $\neg$(*num-of-odd-nodes kon-graph* =*2*) **by** *eval*
  moreover have $\neg$($\forall v \in$*nodes kon-graph*. *even* (*degree v kon-graph*)) **by** *eval*
  ultimately show *False* **using** *contra* **by** *auto*
qed

9 Sufficient conditions for Eulerian trails and circuits

```
lemma (in valid-unMultigraph) eulerian-cons:
  assumes
    valid-unMultigraph.is-Eulerian-trail (del-unEdge v0 w v1 G) v1 ps v2
    (v0,w,v1)∈ E
  shows is-Eulerian-trail v0 ((v0,w,v1)#ps) v2
proof −
  have valid:valid-unMultigraph (del-unEdge v0 w v1 G)
    using  valid-unMultigraph-axioms by auto
  hence distinct:valid-unMultigraph.is-trail (del-unEdge v0 w v1 G) v1 ps v2
    using assms unfolding valid-unMultigraph.is-Eulerian-trail-def [OF valid]
    by auto
  hence set ps ⊆ edges (del-unEdge v0 w v1 G)
    using valid-unMultigraph.path-in-edges[OF valid] by auto
  moreover have (v0,w,v1)∉edges (del-unEdge v0 w v1 G)
    unfolding del-unEdge-def by auto
  moreover have (v1,w,v0)∉edges (del-unEdge v0 w v1 G)
    unfolding del-unEdge-def by auto
  ultimately have (v0,w,v1)∉set ps (v1,w,v0)∉set ps  by auto
  moreover have is-trail v1 ps v2
    using distinct-path-intro[OF distinct] .
  ultimately have is-trail v0 ((v0,w,v1)#ps) v2
    using ‹(v0,w,v1)∈ E› by auto
  moreover have edges (rem-unPath ps (del-unEdge v0 w v1 G)) ={}
    using assms unfolding valid-unMultigraph.is-Eulerian-trail-def [OF valid]
    by auto
  hence edges (rem-unPath ((v0,w,v1)#ps) G)={}
    by (metis rem-unPath.simps(2))
  ultimately show ?thesis unfolding is-Eulerian-trail-def by auto
qed

lemma (in valid-unMultigraph) eulerian-cons′:
  assumes
    valid-unMultigraph.is-Eulerian-trail (del-unEdge v2 w v3 G) v1 ps v2
    (v2,w,v3)∈ E
  shows is-Eulerian-trail v1 (ps@[(v2,w,v3)]) v3
proof −
  have valid:valid-unMultigraph (del-unEdge v3 w v2 G)
    using valid-unMultigraph-axioms del-unEdge-valid by auto
  have del-unEdge v2 w v3 G=del-unEdge v3 w v2 G
    by (metis delete-edge-sym)
  hence valid-unMultigraph.is-Eulerian-trail (del-unEdge v3 w v2 G) v2
       (rev-path ps) v1 using assms valid-unMultigraph.euclerian-rev[OF valid]
    by auto
  hence is-Eulerian-trail v3 ((v3,w,v2)#(rev-path ps)) v1
    using eulerian-cons by (metis assms(2) corres)
  hence is-Eulerian-trail v1 (rev-path((v3,w,v2)#(rev-path ps))) v3
```

using *euclerian-rev* **by** *auto*
moreover have *rev-path*$((v3,w,v2)\#(rev\text{-}path\ ps)) = rev\text{-}path(rev\text{-}path\ ps)@[(v2,w,v3)]$
unfolding *rev-path-def* **by** *auto*
hence *rev-path*$((v3,w,v2)\#(rev\text{-}path\ ps))=ps@[(v2,w,v3)]$ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *eulerian-split*:
assumes *nodes G1* $\cap$ *nodes G2* = {} *edges G1* $\cap$ *edges G2*={}
valid-unMultigraph G1 valid-unMultigraph G2
valid-unMultigraph.is-Eulerian-trail G1 v1 ps1 v1'
valid-unMultigraph.is-Eulerian-trail G2 v2 ps2 v2'
shows *valid-unMultigraph.is-Eulerian-trail* (|*nodes=nodes G1* $\cup$ *nodes G2*,
edges=edges G1 $\cup$ *edges G2* $\cup$ {(*v1'*,*w*,*v2*),(*v2*,*w*,*v1'*)}|) *v1* (*ps1*@(*v1'*,*w*,*v2*)#*ps2*)
v2'
proof –
have *valid-graph G1* **using** ‹*valid-unMultigraph G1*› *valid-unMultigraph-def* **by**
auto
have *valid-graph G2* **using** ‹*valid-unMultigraph G2*› *valid-unMultigraph-def* **by**
auto
obtain *G* **where** *G*:*G*=(|*nodes=nodes G1* $\cup$ *nodes G2*, *edges=edges G1* $\cup$ *edges*
G2
$\cup$ {(*v1'*,*w*,*v2*),(*v2*,*w*,*v1'*)}|)
by *metis*
have *v1'*$\in$*nodes G1*
by (*metis* (*full-types*) ‹*valid-graph G1*› *assms*(*3*) *assms*(*5*) *valid-graph.is-path-memb*
valid-unMultigraph.is-trail-intro valid-unMultigraph.is-Eulerian-trail-def)
moreover have *v2*$\in$*nodes G2*
by (*metis* (*full-types*) ‹*valid-graph G2*› *assms*(*4*) *assms*(*6*) *valid-graph.is-path-memb*
valid-unMultigraph.is-trail-intro valid-unMultigraph.is-Eulerian-trail-def)
ultimately have *valid-unMultigraph* (|*nodes=nodes G1* $\cup$ *nodes G2*, *edges=edges*
G1 $\cup$ *edges G2* $\cup$
{(*v1'*,*w*,*v2*),(*v2*,*w*,*v1'*)}|)
using
valid-unMultigraph.corres[*OF* ‹*valid-unMultigraph G1*›]
valid-unMultigraph.no-id[*OF* ‹*valid-unMultigraph G1*›]
valid-unMultigraph.corres[*OF* ‹*valid-unMultigraph G2*›]
valid-unMultigraph.no-id[*OF* ‹*valid-unMultigraph G2*›]
valid-graph.E-validD[*OF* ‹*valid-graph G1*›]
valid-graph.E-validD[*OF* ‹*valid-graph G2*›]
‹*nodes G1* $\cap$ *nodes G2* = {}›
proof (*unfold-locales*,*auto*)
fix *aa ab ba*
assume (*aa*, *ab*, *ba*) $\in$ *edges G1*
thus *ba* $\in$ *nodes G1* **by** (*metis* ‹$\bigwedge v'\ v\ e.\ (v,\ e,\ v') \in$ *edges G1* $\Longrightarrow v' \in$ *nodes*
G1›)
next
fix *aa ab ba*
assume *ba* $\notin$ *nodes G2* (*aa*, *ab*, *ba*) $\in$ *edges G2*

thus *ba* ∈ *nodes G1* **by** (*metis* ‹*valid-graph G2*› *valid-graph.E-validD*(*2*))
qed
hence *valid*: *valid-unMultigraph G* **using** *G* **by** *auto*
hence *valid'*:*valid-graph G* **using** *valid-unMultigraph-def* **by** *auto*
moreover have *valid-unMultigraph.is-trail G v1* (*ps1*@((*v1'*,*w*,*v2*)#*ps2*)) *v2'*
proof −
have *ps1-G*:*valid-unMultigraph.is-trail G v1 ps1 v1'*
proof −
have *valid-unMultigraph.is-trail G1 v1 ps1 v1'* **using** *assms*
by (*metis valid-unMultigraph.is-Eulerian-trail-def*)
moreover have *edges G1* ⊆ *edges G* **by** (*metis G UnI1 Un-assoc select-convs*(*2*) *subrelI*)
moreover have *nodes G1* ⊆ *nodes G* **by** (*metis G inf-sup-absorb le-iff-inf select-convs*(*1*))
ultimately show *?thesis*
using *distinct-path-subset*[*of G1 G*,*OF* ‹*valid-unMultigraph G1*› *valid*] **by** *auto*
qed
have *ps2-G*:*valid-unMultigraph.is-trail G v2 ps2 v2'*
proof −
have *valid-unMultigraph.is-trail G2 v2 ps2 v2'* **using** *assms*
by (*metis valid-unMultigraph.is-Eulerian-trail-def*)
moreover have *edges G2* ⊆ *edges G* **by** (*metis G inf-sup-ord*(*3*) *le-supE select-convs*(*2*))
moreover have *nodes G2* ⊆ *nodes G* **by** (*metis G inf-sup-ord*(*4*) *select-convs*(*1*))
ultimately show *?thesis*
using *distinct-path-subset*[*of G2 G*,*OF* ‹*valid-unMultigraph G2*› *valid*] **by** *auto*
qed
have *valid-graph.is-path G v1* (*ps1*@((*v1'*,*w*,*v2*)#*ps2*)) *v2'*
proof −
have *valid-graph.is-path G v1 ps1 v1'*
by (*metis ps1-G valid valid-unMultigraph.is-trail-intro*)
moreover have *valid-graph.is-path G v2 ps2 v2'*
by (*metis ps2-G valid valid-unMultigraph.is-trail-intro*)
moreover have (*v1'*,*w*,*v2*) ∈ *edges G*
using *G* **by** *auto*
ultimately show *?thesis*
using *valid-graph.is-path-split'*[*OF valid'*,*of v1 ps1 v1' w v2 ps2 v2'*] **by** *auto*
qed
moreover have *distinct* (*ps1*@((*v1'*,*w*,*v2*)#*ps2*))
proof −
have *distinct ps1* **by** (*metis ps1-G valid valid-unMultigraph.is-trail-path*)
moreover have *distinct ps2*
by (*metis ps2-G valid valid-unMultigraph.is-trail-path*)
moreover have *set ps1* ∩ *set ps2* = {}
proof −

have *set ps1* ⊆*edges G1*
by (*metis assms*(*3*) *assms*(*5*) *valid-unMultigraph.is-Eulerian-trail-def*
valid-unMultigraph.path-in-edges)
moreover have *set ps2* ⊆ *edges G2*
by (*metis assms*(*4*) *assms*(*6*) *valid-unMultigraph.is-Eulerian-trail-def*
valid-unMultigraph.path-in-edges)
ultimately show *?thesis* **using** ‹*edges G1* ∩ *edges G2*={}› **by** *auto*
qed
moreover have (*v1'*,*w*,*v2*)∉*edges G1*
using ‹*v2* ∈ *nodes G2*› ‹*valid-graph G1*›
by (*metis Int-iff all-not-in-conv assms*(*1*) *valid-graph.E-validD*(*2*))
hence (*v1'*,*w*,*v2*)∉*set ps1*
by (*metis* (*full-types*) *assms*(*3*) *assms*(*5*) *subsetD valid-unMultigraph.path-in-edges*
valid-unMultigraph.is-Eulerian-trail-def)
moreover have (*v1'*,*w*,*v2*)∉*edges G2*
using ‹*v1'* ∈ *nodes G1*› ‹*valid-graph G2*›
by (*metis assms*(*1*) *disjoint-iff-not-equal valid-graph.E-validD*(*1*))
hence (*v1'*,*w*,*v2*)∉*set ps2*
by (*metis* (*full-types*) *assms*(*4*) *assms*(*6*) *in-mono valid-unMultigraph.path-in-edges*
valid-unMultigraph.is-Eulerian-trail-def)
ultimately show *?thesis* **using** *distinct-append* **by** *auto*
qed
moreover have *set* (*ps1*@((*v1'*,*w*,*v2*)#*ps2*)) ∩ *set* (*rev-path* (*ps1*@((*v1'*,*w*,*v2*)#*ps2*)))
= {}
proof –
have *set ps1* ∩ *set* (*rev-path ps1*) = {}
by (*metis ps1-G valid valid-unMultigraph.is-trail-path*)
moreover have *set* (*rev-path ps2*) ⊆ *edges G2*
by (*metis assms*(*4*) *assms*(*6*) *valid-unMultigraph.is-trail-rev*
valid-unMultigraph.is-Eulerian-trail-def valid-unMultigraph.path-in-edges)
hence *set ps1* ∩ *set* (*rev-path ps2*) = {}
using *assms*
valid-unMultigraph.path-in-edges[*OF* ‹*valid-unMultigraph G1*›, *of v1 ps1*
v1']
valid-unMultigraph.path-in-edges[*OF* ‹*valid-unMultigraph G2*›, *of v2 ps2*
v2']
unfolding *valid-unMultigraph.is-Eulerian-trail-def*[*OF* ‹*valid-unMultigraph*
G1›]
valid-unMultigraph.is-Eulerian-trail-def[*OF* ‹*valid-unMultigraph G2*›]
by *auto*
moreover have *set ps2* ∩ *set* (*rev-path ps2*) = {}
by (*metis ps2-G valid valid-unMultigraph.is-trail-path*)
moreover have *set* (*rev-path ps1*) ⊆*edges G1*
by (*metis assms*(*3*) *assms*(*5*) *valid-unMultigraph.is-Eulerian-trail-def*
valid-unMultigraph.path-in-edges valid-unMultigraph.euclerian-rev)
hence *set ps2* ∩ *set* (*rev-path ps1*) = {}
by (*metis calculation*(*2*) *distinct-append distinct-rev-path ps1-G ps2-G*
rev-path-append
rev-path-double valid valid-unMultigraph.is-trail-path)

```
          moreover have (v2,w,v1')∉set (ps1@((v1',w,v2)#ps2))
            proof -
              have (v2,w,v1')∉edges G1
                using ‹v2 ∈ nodes G2› ‹valid-graph G1›
                by (metis Int-iff all-not-in-conv assms(1) valid-graph.E-validD(1))
              hence (v2,w,v1')∉set ps1
              by (metis assms(3) assms(5) split-list valid-unMultigraph.is-trail-split'
                  valid-unMultigraph.is-Eulerian-trail-def)
              moreover have (v2,w,v1')∉edges G2
                using ‹v1' ∈ nodes G1› ‹valid-graph G2›
                by (metis IntI assms(1) empty-iff valid-graph.E-validD(2))
              hence (v2,w,v1')∉set ps2
            by (metis (full-types) assms(4) assms(6) in-mono valid-unMultigraph.path-in-edges
                  valid-unMultigraph.is-Eulerian-trail-def)
              moreover have (v2,w,v1')≠(v1',w,v2)
                using ‹v1' ∈ nodes G1› ‹v2 ∈ nodes G2›
                by (metis IntI Pair-inject assms(1) assms(5) bex-empty)
              ultimately show ?thesis by auto
            qed
          ultimately show ?thesis using rev-path-append by auto
        qed
      ultimately show ?thesis using valid-unMultigraph.is-trail-path[OF valid]
        by auto
    qed
  moreover have edges (rem-unPath (ps1@((v1',w,v2)#ps2)) G)= {}
    proof -
      have edges (rem-unPath (ps1@((v1',w,v2)#ps2)) G)=edges G -
          (set (ps1@((v1',w,v2)#ps2)) ∪ set (rev-path (ps1@((v1',w,v2)#ps2))))
        by (metis rem-unPath-edges)
      also have ...=edges G - (set ps1 ∪ set ps2 ∪ set (rev-path ps1) ∪ set (rev-path
ps2)
                ∪ {(v1',w,v2),(v2,w,v1')}) using rev-path-append by auto
      finally have edges (rem-unPath (ps1@((v1',w,v2)#ps2)) G) = edges G -
(set ps1 ∪
              set ps2 ∪ set (rev-path ps1) ∪ set (rev-path ps2) ∪ {(v1',w,v2),(v2,w,v1')})
.
      moreover have edges (rem-unPath ps1 G1)={}
        by (metis assms(3) assms(5) valid-unMultigraph.is-Eulerian-trail-def)
      hence edges G1 - (set ps1 ∪ set (rev-path ps1))={}
        by (metis rem-unPath-edges)
      moreover have edges (rem-unPath ps2 G2)={}
        by (metis assms(4) assms(6) valid-unMultigraph.is-Eulerian-trail-def)
      hence edges G2 - (set ps2 ∪ set (rev-path ps2))={}
        by (metis rem-unPath-edges)
      ultimately show ?thesis using G by auto
    qed
  ultimately show ?thesis by (metis G valid valid-unMultigraph.is-Eulerian-trail-def)
qed
```

lemma (**in** *valid-unMultigraph*) *eulerian-sufficient*:
assumes *finite V finite E connected* $V \neq \{\}$
shows *num-of-odd-nodes* $G = 2 \Longrightarrow$
($\exists\, v \in V. \exists\, v' \in V. \exists\, ps.$ *odd*(*degree v G*)$\wedge$*odd*(*degree v′ G*)$\wedge(v \neq v')\wedge$*is-Eulerian-trail v ps v′*)
and *num-of-odd-nodes* $G=0 \Longrightarrow (\forall\, v \in V. \exists\, ps.$ *is-Eulerian-circuit v ps v*)
using ‹*finite E*› ‹*finite V*› *valid-unMultigraph-axioms* ‹$V \neq \{\}$› ‹*connected*›
proof (*induct card E arbitrary*: *G rule*: *less-induct*)
case *less*
assume *finite* (*edges G*) **and** *finite* (*nodes G*) **and** *valid-unMultigraph G* **and** *nodes* $G \neq \{\}$
and *valid-unMultigraph.connected G* **and** *num-of-odd-nodes* $G = 2$
have *valid-graph G* **using** ‹*valid-unMultigraph G*› *valid-unMultigraph-def* **by** *auto*
obtain *n1 n2* **where**
n1: *n1*∈*nodes G odd*(*degree n1 G*)
and *n2*: *n2*∈*nodes G odd*(*degree n2 G*)
and *n1*≠*n2* **unfolding** *num-of-odd-nodes-def odd-nodes-set-def*
proof −
have $\forall\, S.$ *card* $S=2 \longrightarrow (\exists\, n1\ n2.\ n1 \in S \wedge n2 \in S \wedge n1 \neq n2)$
by (*metis card-eq-0-iff equals0I even-card′ even-numeral zero-neq-numeral*)
then obtain *t1 t2*
where *t1*∈{*v* ∈ *nodes G*. *odd* (*degree v G*)} *t2*∈{*v* ∈ *nodes G*. *odd* (*degree v G*)} *t1*≠*t2*
using ‹*num-of-odd-nodes G* = *2*› **unfolding** *num-of-odd-nodes-def odd-nodes-set-def*
by *force*
thus *?thesis* **by** (*metis* (*lifting*) *that mem-Collect-eq*)
qed
have *even-except-two*:$\bigwedge n.$ *n*∈*nodes G*⟹ *n*≠*n1* ⟹ *n*≠*n2* ⟹ *even*(*degree n G*)
proof (*rule ccontr*)
fix *n* **assume** *n* ∈ *nodes G* *n* ≠ *n1* *n* ≠ *n2* *odd* (*degree n G*)
have *n*∈ *odd-nodes-set G*
by (*metis* (*mono-tags*) ‹*n* ∈ *nodes G*› ‹*odd* (*degree n G*)› *mem-Collect-eq odd-nodes-set-def*)
moreover have *n1* ∈ *odd-nodes-set G*
by (*metis* (*mono-tags*) *mem-Collect-eq n1*(*1*) *n1*(*2*) *odd-nodes-set-def*)
moreover have *n2* ∈ *odd-nodes-set G*
using *n2*(*1*) *n2*(*2*) **unfolding** *odd-nodes-set-def* **by** *auto*
ultimately have {*n*,*n1*,*n2*}⊆ *odd-nodes-set G* **by** *auto*
moreover have *card*{*n*,*n1*,*n2*} ≥*3* **using** ‹*n1*≠*n2*› ‹*n*≠*n1*› ‹*n*≠*n2*› **by** *auto*
moreover have *finite* (*odd-nodes-set G*)
using ‹*finite* (*nodes G*)› **unfolding** *odd-nodes-set-def* **by** *auto*
ultimately have *card* (*odd-nodes-set G*) ≥ *3*
using *card-mono*[*of odd-nodes-set G* {*n*, *n1*, *n2*}] **by** *auto*
thus *False* **using** ‹*num-of-odd-nodes G* = *2*› **unfolding** *num-of-odd-nodes-def* **by** *auto*
qed
have {*e* ∈ *edges G*. *fst e* = *n1*}≠{}
using *n1*

by (*metis* (*full-types*) *degree-def empty-iff finite.emptyI odd-card*)
then obtain *v′ w* **where** (*n1*,*w*,*v′*)∈*edges G* **by** *auto*
have *v′*=*n2* ⟹ (∃ *v*∈*nodes G*. ∃ *v′*∈*nodes G*.∃ *ps*. *odd* (*degree v G*) ∧ *odd* (*degree v′ G*) ∧ *v* ≠ *v′*
∧ *valid-unMultigraph.is-Eulerian-trail G v ps v′*)
proof (*cases valid-unMultigraph.connected* (*del-unEdge n1 w n2 G*))
assume *v′*=*n2*
assume *conneted′*:*valid-unMultigraph.connected* (*del-unEdge n1 w n2 G*)
moreover have *num-of-odd-nodes* (*del-unEdge n1 w n2 G*) = *0*
using ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*finite* (*edges G*)› ‹*finite* (*nodes G*)› ‹*v′* = *n2*›
‹*num-of-odd-nodes G* = *2*› ‹*valid-unMultigraph G*› *del-UnEdge-odd-odd n1*(*2*) *n2*(*2*)
by *force*
moreover have *finite* (*edges* (*del-unEdge n1 w n2 G*))
using ‹*finite* (*edges G*)› **by** *auto*
moreover have *finite* (*nodes* (*del-unEdge n1 w n2 G*))
using ‹*finite* (*nodes G*)› **by** *auto*
moreover have *edges G* − {(*n1*,*w*,*n2*),(*n2*,*w*,*n1*)} ⊂ *edges G*
using *Diff-iff Diff-subset* ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*v′* = *n2*›
by *fast*
hence *card* (*edges* (*del-unEdge n1 w n2 G*)) < *card* (*edges G*)
using ‹*finite* (*edges G*)› *psubset-card-mono*[*of edges G edges G* − {(*n1*,*w*,*n2*),(*n2*,*w*,*n1*)}]
unfolding *del-unEdge-def* **by** *auto*
moreover have *valid-unMultigraph* (*del-unEdge n1 w n2 G*)
using ‹*valid-unMultigraph G*› *del-unEdge-valid* **by** *auto*
moreover have *nodes* (*del-unEdge n1 w n2 G*) ≠ {}
by (*metis* (*full-types*) *del-UnEdge-node empty-iff n1*(*1*))
ultimately have ∀ *v*∈*nodes* (*del-unEdge n1 w n2 G*). ∃ *ps*. *valid-unMultigraph.is-Eulerian-circuit* (*del-unEdge n1 w n2 G*) *v ps v*
using *less.hyps*[*of del-unEdge n1 w n2 G*] **by** *auto*
thus *?thesis* **using** *eulerian-cons*
by (*metis* ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*n1* ≠ *n2*› ‹*v′* = *n2*› ‹*valid-unMultigraph G*›
‹*valid-unMultigraph* (*del-unEdge n1 w n2 G*)› *del-UnEdge-node n1*(*1*) *n1*(*2*) *n2*(*1*) *n2*(*2*)
valid-unMultigraph.eulerian-cons valid-unMultigraph.is-Eulerian-circuit-def)
next
assume *v′*=*n2*
assume *not-conneted*:¬*valid-unMultigraph.connected* (*del-unEdge n1 w n2 G*)
have *valid0*:*valid-unMultigraph* (*del-unEdge n1 w n2 G*)
using ‹*valid-unMultigraph G*› *del-unEdge-valid* **by** *auto*
hence *valid0′*:*valid-graph* (*del-unEdge n1 w n2 G*)
using *valid-unMultigraph-def* **by** *auto*
have *all-even*:∀ *n*∈*nodes* (*del-unEdge n1 w n2 G*). *even*(*degree n* (*del-unEdge n1 w n2 G*))
proof −
have *even* (*degree n1* (*del-unEdge n1 w n2 G*))
using ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*finite* (*edges G*)› ‹*v′* = *n2*› ‹*valid-unMultigraph G*› *n1*

by (*auto simp add*: *valid-unMultigraph.corres*)
moreover have *even* (*degree n2* (*del-unEdge n1 w n2 G*))
using ‹(*n1*, *w*, *v'*) ∈ *edges G*› ‹*finite* (*edges G*)› ‹*v'* = *n2*› ‹*valid-unMultigraph G*› *n2*
by (*auto simp add*: *valid-unMultigraph.corres*)
moreover have $\bigwedge n.\ n \in$ *nodes* (*del-unEdge n1 w n2 G*) $\Longrightarrow n \neq n1 \Longrightarrow n \neq n2 \Longrightarrow$
even (*degree n* (*del-unEdge n1 w n2 G*))
using *valid-unMultigraph.degree-frame*[*OF* ‹*valid-unMultigraph G*›,
of - n1 n2 w] *even-except-two*
by (*metis* (*no-types*) ‹*finite* (*edges G*)› *del-unEdge-def empty-iff insert-iff*
select-convs(*1*))
ultimately show *?thesis* **by** *auto*
qed
have (*n1*,*w*,*n2*)∈*edges G* **by** (*metis* ‹(*n1*, *w*, *v'*) ∈ *edges G*› ‹*v'* = *n2*›)
hence (*n2*,*w*,*n1*)∈*edges G* **by** (*metis* ‹*valid-unMultigraph G*› *valid-unMultigraph.corres*)
obtain *G1 G2* **where**
G1-nodes: *nodes G1*={*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge n1 w n2 G*) *n ps n1*}
and *G1-edges*: *edges G1*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge n1 w n2 G*)
∧ *n*∈*nodes G1* ∧ *n'*∈*nodes G1*}
and *G2-nodes*:*nodes G2*={*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge n1 w n2 G*) *n ps n2*}
and *G2-edges*:*edges G2*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge n1 w n2 G*) ∧ *n*∈*nodes G2*
∧ *n'*∈*nodes G2*}
and *G1-G2-edges-union*:*edges G1* ∪ *edges G2* = *edges* (*del-unEdge n1 w n2 G*)
and *edges G1* ∩ *edges G2*={}
and *G1-G2-nodes-union*:*nodes G1* ∪ *nodes G2*=*nodes* (*del-unEdge n1 w n2 G*)
and *nodes G1* ∩ *nodes G2*={}
and *valid-unMultigraph G1*
and *valid-unMultigraph G2*
and *valid-unMultigraph.connected G1*
and *valid-unMultigraph.connected G2*
using *valid-unMultigraph.connectivity-split*[*OF* ‹*valid-unMultigraph G*›
‹*valid-unMultigraph.connected G*› ‹¬ *valid-unMultigraph.connected* (*del-unEdge n1 w n2 G*)›
‹(*n1*, *w*, *n2*) ∈ *edges G*›] **.**
have *edges* (*del-unEdge n1 w n2 G*) ⊂ *edges G*
unfolding *del-unEdge-def* **using** ‹(*n1*, *w*, *n2*)∈*edges G*› ‹(*n2*, *w*, *n1*)∈*edges G*› **by** *auto*
hence *card* (*edges G1*) < *card* (*edges G*) **using** *G1-G2-edges-union*
by (*metis* (*full-types*) ‹*finite* (*edges G*)› *inf-sup-absorb less-infI2 psubset-card-mono*)
moreover have *finite* (*edges G1*)
using *G1-G2-edges-union* ‹*finite* (*edges G*)›
by (*metis* ‹*edges* (*del-unEdge n1 w n2 G*) ⊂ *edges G*› *finite-Un less-imp-le*

rev-finite-subset)
moreover have *nodes G1* ⊆ *nodes* (*del-unEdge n1 w n2 G*)
by (*metis G1-G2-nodes-union Un-upper1*)
hence *finite* (*nodes G1*)
using ‹*finite* (*nodes G*)› *del-UnEdge-node rev-finite-subset* **by** *auto*
moreover have *n1* ∈ *nodes G1*
proof −
have *n1*∈*nodes* (*del-unEdge n1 w n2 G*) **using** ‹*n1*∈*nodes G*› **by** *auto*
hence *valid-graph.is-path* (*del-unEdge n1 w n2 G*) *n1* [] *n1*
using *valid0′* **by** (*metis valid-graph.is-path-simps*(*1*))
thus *?thesis* **using** *G1-nodes* **by** *auto*
qed
hence *nodes G1* ≠ {} **by** *auto*
moreover have *num-of-odd-nodes G1* = *0*
proof −
have *valid-graph G2* **using** ‹*valid-unMultigraph G2*› *valid-unMultigraph-def*
by *auto*
hence ∀ *n*∈*nodes G1*. *degree n G1* = *degree n* (*del-unEdge n1 w n2 G*)
using *sub-graph-degree-frame*[*of G2 G1* (*del-unEdge n1 w n2 G*)]
by (*metis G1-G2-edges-union* ‹*nodes G1* ∩ *nodes G2* = {}›)
hence ∀ *n*∈*nodes G1*. *even*(*degree n G1*) **using** *all-even*
by (*metis G1-G2-nodes-union Un-iff*)
thus *?thesis*
unfolding *num-of-odd-nodes-def odd-nodes-set-def*
by (*metis* (*lifting*) *Collect-empty-eq card-eq-0-iff*)
qed
ultimately have ∀ *v*∈*nodes G1*. ∃ *ps*. *valid-unMultigraph.is-Eulerian-circuit*
G1 v ps v
using *less.hyps*[*of G1*] ‹*valid-unMultigraph G1*› ‹*valid-unMultigraph.connected*
G1›
by *auto*
then obtain *ps1* **where** *ps1*:*valid-unMultigraph.is-Eulerian-trail G1 n1 ps1*
n1
using ‹*n1*∈*nodes G1*›
by (*metis* (*full-types*) ‹*valid-unMultigraph G1*› *valid-unMultigraph.is-Eulerian-circuit-def*)
have *card* (*edges G2*) < *card* (*edges G*)
using *G1-G2-edges-union* ‹*edges* (*del-unEdge n1 w n2 G*) ⊂ *edges G*›
by (*metis* (*full-types*) ‹*finite* (*edges G*)› *inf-sup-ord*(*4*) *le-less-trans psubset-card-mono*)
moreover have *finite* (*edges G2*)
using *G1-G2-edges-union* ‹*finite* (*edges G*)›
by (*metis* ‹*edges* (*del-unEdge n1 w n2 G*) ⊂ *edges G*› *finite-Un less-imp-le*
rev-finite-subset)
moreover have *nodes G2* ⊆ *nodes* (*del-unEdge n1 w n2 G*)
by (*metis G1-G2-nodes-union Un-upper2*)
hence *finite* (*nodes G2*)
using ‹*finite* (*nodes G*)› *del-UnEdge-node rev-finite-subset* **by** *auto*
moreover have *n2* ∈ *nodes G2*
proof −
have *n2*∈*nodes* (*del-unEdge n1 w n2 G*)

using ‹*n2*∈*nodes G*› **by** *auto*
hence *valid-graph.is-path* (*del-unEdge n1 w n2 G*) *n2* [] *n2*
using *valid0′* **by** (*metis valid-graph.is-path-simps*(*1*))
thus *?thesis* **using** *G2-nodes* **by** *auto*
qed
hence *nodes G2* ≠ {} **by** *auto*
moreover have *num-of-odd-nodes G2* = *0*
proof −
have *valid-graph G1* **using** ‹*valid-unMultigraph G1*› *valid-unMultigraph-def*
by *auto*
hence ∀ *n*∈*nodes G2*. *degree n G2* = *degree n* (*del-unEdge n1 w n2 G*)
using *sub-graph-degree-frame*[*of G1 G2* (*del-unEdge n1 w n2 G*)]
by (*metis G1-G2-edges-union* ‹*nodes G1* ∩ *nodes G2* = {}› *inf-commute*
sup-commute)
hence ∀ *n*∈*nodes G2*. *even*(*degree n G2*) **using** *all-even*
by (*metis G1-G2-nodes-union Un-iff*)
thus *?thesis*
unfolding *num-of-odd-nodes-def odd-nodes-set-def*
by (*metis* (*lifting*) *Collect-empty-eq card-eq-0-iff*)
qed
ultimately have ∀ *v*∈*nodes G2*. ∃ *ps*. *valid-unMultigraph.is-Eulerian-circuit*
G2 v ps v
using *less.hyps*[*of G2*] ‹*valid-unMultigraph G2*› ‹*valid-unMultigraph.connected*
G2›
by *auto*
then obtain *ps2* **where** *ps2*:*valid-unMultigraph.is-Eulerian-trail G2 n2 ps2*
n2
using ‹*n2*∈*nodes G2*›
by (*metis* (*full-types*) ‹*valid-unMultigraph G2*› *valid-unMultigraph.is-Eulerian-circuit-def*)
have (|*nodes* = *nodes G1* ∪ *nodes G2*, *edges* = *edges G1* ∪ *edges G2* ∪ {(*n1*,
w, *n2*),
(*n2*, *w*, *n1*)}|)=*G*
proof −
have *edges* (*del-unEdge n1 w n2 G*) ∪ {(*n1*, *w*, *n2*),(*n2*, *w*, *n1*)} =*edges*
G
using ‹(*n1*,*w*,*n2*)∈*edges G*› ‹(*n2*,*w*,*n1*)∈*edges G*›
unfolding *del-unEdge-def* **by** *auto*
moreover have *nodes* (*del-unEdge n1 w n2 G*)=*nodes G*
unfolding *del-unEdge-def* **by** *auto*
ultimately have (|*nodes* = *nodes* (*del-unEdge n1 w n2 G*), *edges* =
edges (*del-unEdge n1 w n2 G*) ∪ {(*n1*, *w*, *n2*), (*n2*, *w*, *n1*)}|)=*G*
by *auto*
moreover have (|*nodes* = *nodes G1* ∪ *nodes G2*, *edges* = *edges G1* ∪
edges G2 ∪
{(*n1*, *w*, *n2*),(*n2*, *w*, *n1*)}|)=(|*nodes* = *nodes* (*del-unEdge n1 w n2*
G),*edges*
= *edges* (*del-unEdge n1 w n2 G*) ∪ {(*n1*, *w*, *n2*), (*n2*, *w*, *n1*)}|)
by (*metis G1-G2-edges-union G1-G2-nodes-union*)
ultimately show *?thesis* **by** *auto*

qed
moreover have *valid-unMultigraph.is-Eulerian-trail* (|*nodes* = *nodes G1* ∪ *nodes G2*,
edges = *edges G1* ∪ *edges G2* ∪ {(*n1*, *w*, *n2*), (*n2*, *w*, *n1*)}|) *n1* (*ps1* @ (*n1*, *w*, *n2*) # *ps2*) *n2*
using *eulerian-split*[*of G1 G2 n1 ps1 n1 n2 ps2 n2 w*]
by (*metis* ‹*edges G1* ∩ *edges G2* = {}› ‹*nodes G1* ∩ *nodes G2* = {}› ‹*valid-unMultigraph G1*›
‹*valid-unMultigraph G2*› *ps1 ps2*)
ultimately show *?thesis* **by** (*metis* ‹*n1* ≠ *n2*› *n1*(*1*) *n1*(*2*) *n2*(*1*) *n2*(*2*))
qed
moreover have *v′*≠*n2* ⟹ (∃ *v*∈*nodes G*. ∃ *v′*∈*nodes G*.∃ *ps*. *odd* (*degree v G*) ∧ *odd* (*degree v′ G*)
∧ *v* ≠ *v′* ∧ *valid-unMultigraph.is-Eulerian-trail G v ps v′*)
proof (*cases valid-unMultigraph.connected* (*del-unEdge n1 w v′ G*))
case *True*
assume *v′* ≠ *n2*
assume *connected′*:*valid-unMultigraph.connected* (*del-unEdge n1 w v′ G*)
have *n1* ∈ *nodes* (*del-unEdge n1 w v′ G*) **by** (*metis del-UnEdge-node n1*(*1*))
hence *even-n1*:*even*(*degree n1* (*del-unEdge n1 w v′ G*))
using *valid-unMultigraph.del-UnEdge-even*[*OF* ‹*valid-unMultigraph G*› ‹(*n1*, *w*, *v′*) ∈ *edges G*›
‹*finite* (*edges G*)›] ‹*odd* (*degree n1 G*)›
unfolding *odd-nodes-set-def* **by** *auto*
moreover have *odd-n2*:*odd*(*degree n2* (*del-unEdge n1 w v′ G*))
using *valid-unMultigraph.degree-frame*[*OF* ‹*valid-unMultigraph G*› ‹*finite* (*edges G*)›,
of n2 n1 v′ w] ‹*n1* ≠ *n2*› ‹*v′* ≠ *n2*›
by (*metis empty-iff insert-iff n2*(*2*))
moreover have *even* (*degree v′ G*)
using *even-except-two*[*of v′*]
by (*metis* (*full-types*) ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*v′* ≠ *n2*› ‹*valid-graph G*›
‹*valid-unMultigraph G*› *valid-graph.E-validD*(*2*) *valid-unMultigraph.no-id*)
hence *odd-v′*:*odd*(*degree v′* (*del-unEdge n1 w v′ G*))
using *valid-unMultigraph.del-UnEdge-even′*[*OF* ‹*valid-unMultigraph G*› ‹(*n1*, *w*, *v′*) ∈ *edges G*›
‹*finite* (*edges G*)›]
unfolding *odd-nodes-set-def* **by** *auto*
ultimately have *two-odds*:*num-of-odd-nodes* (*del-unEdge n1 w v′ G*) = *2*
by (*metis* (*lifting*) ‹*v′* ≠ *n2*› ‹*valid-graph G*› ‹*valid-unMultigraph G*›
‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*finite* (*edges G*)› ‹*finite* (*nodes G*)› ‹*num-of-odd-nodes G* = *2*›
del-UnEdge-odd-even even-except-two n1(*2*) *valid-graph.E-validD*(*2*))
moreover have *valid0*:*valid-unMultigraph* (*del-unEdge n1 w v′ G*)
using *del-unEdge-valid* ‹*valid-unMultigraph G*› **by** *auto*
moreover have *edges G* − {(*n1*, *w*, *v′*), (*v′*, *w*, *n1*)} ⊂ *edges G*
using ‹(*n1*,*w*,*v′*)∈*edges G*› **by** *auto*
hence *card* (*edges* (*del-unEdge n1 w v′ G*)) < *card* (*edges G*)
using ‹*finite* (*edges G*)› **unfolding** *del-unEdge-def*

by (*metis* (*hide-lams*, *no-types*) *psubset-card-mono select-convs*(*2*))
moreover have *finite* (*edges* (*del-unEdge n1 w v′ G*))
unfolding *del-unEdge-def*
by (*metis* (*full-types*) ‹*finite* (*edges G*)› *finite-Diff select-convs*(*2*))
moreover have *finite* (*nodes* (*del-unEdge n1 w v′ G*))
unfolding *del-unEdge-def* **by** (*metis* ‹*finite* (*nodes G*)› *select-convs*(*1*))
moreover have *nodes* (*del-unEdge n1 w v′ G*) ≠ {}
by (*metis* (*full-types*) *del-UnEdge-node empty-iff n1*(*1*))
ultimately obtain *s t ps* **where**
s: *s*∈*nodes* (*del-unEdge n1 w v′ G*) *odd* (*degree s* (*del-unEdge n1 w v′ G*))
and *t*:*t*∈*nodes* (*del-unEdge n1 w v′ G*) *odd* (*degree t* (*del-unEdge n1 w v′
G*))
and *s* ≠ *t*
and *s-ps-t*: *valid-unMultigraph.is-Eulerian-trail* (*del-unEdge n1 w v′ G*) *s
ps t*
using *connected′ less.hyps*[*of* (*del-unEdge n1 w v′ G*)] **by** *auto*
hence (*s*=*n2*∧*t*=*v′*)∨(*s*=*v′*∧*t*=*n2*)
using *odd-n2 odd-v′ two-odds* ‹*finite* (*edges G*)›‹*valid-unMultigraph G*›
by (*metis* (*mono-tags*) *del-UnEdge-node empty-iff even-except-two even-n1
insert-iff*
valid-unMultigraph.degree-frame)
moreover have *s*=*n2*⟹*t*=*v′*⟹*?thesis*
by (*metis* ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*n1* ≠ *n2*› ‹*valid-unMultigraph G*› *n1*(*1*)
n1(*2*) *n2*(*1*) *n2*(*2*)
s-ps-t valid0 valid-unMultigraph.euclerian-rev valid-unMultigraph.eulerian-cons)
moreover have *s*=*v′*⟹*t*=*n2*⟹*?thesis*
by (*metis* ‹(*n1*, *w*, *v′*) ∈ *edges G*› ‹*n1* ≠ *n2*› ‹*valid-unMultigraph G*› *n1*(*1*)
n1(*2*) *n2*(*1*) *n2*(*2*)
s-ps-t valid-unMultigraph.eulerian-cons)
ultimately show *?thesis* **by** *auto*
next
case *False*
assume *v′*≠*n2*
assume *not-conneted*:¬*valid-unMultigraph.connected* (*del-unEdge n1 w v′ G*)
have (*v′*,*w*,*n1*)∈*edges G* **using** ‹(*n1*,*w*,*v′*)∈*edges G*›
by (*metis* ‹*valid-unMultigraph G*› *valid-unMultigraph.corres*)
have *valid0*:*valid-unMultigraph* (*del-unEdge n1 w v′ G*)
using ‹*valid-unMultigraph G*› *del-unEdge-valid* **by** *auto*
hence *valid0′*:*valid-graph* (*del-unEdge n1 w v′ G*)
using *valid-unMultigraph-def* **by** *auto*
have *even-n1*:*even*(*degree n1* (*del-unEdge n1 w v′ G*))
using *valid-unMultigraph.del-UnEdge-even*[*OF* ‹*valid-unMultigraph G*›
‹(*n1*,*w*,*v′*)∈*edges G*›
‹*finite* (*edges G*)›] *n1*
unfolding *odd-nodes-set-def* **by** *auto*
moreover have *odd-n2*:*odd*(*degree n2* (*del-unEdge n1 w v′ G*))
using ‹*n1* ≠ *n2*› ‹*v′* ≠ *n2*› *n2 valid-unMultigraph.degree-frame*[*OF* ‹*valid-unMultigraph
G*›
‹*finite* (*edges G*)›, *of n2 n1 v′ w*]

by *auto*
moreover have $v' \neq n1$
using *valid-unMultigraph.no-id*[*OF* ‹*valid-unMultigraph G*›] ‹($n1$,w,v')∈*edges* *G*› **by** *auto*
hence *odd-v'*:*odd*(*degree v'* (*del-unEdge n1 w v' G*))
using ‹$v' \neq n2$› *even-except-two*[*of v'*]
valid-graph.E-validD(*2*)[*OF* ‹*valid-graph G*› ‹($n1$, w, v') ∈ *edges G*›]
valid-unMultigraph.del-UnEdge-even'[*OF* ‹*valid-unMultigraph G*› ‹($n1$, w, v') ∈ *edges G*›
‹*finite* (*edges G*)›]
unfolding *odd-nodes-set-def* **by** *auto*
ultimately have *even-except-two'*:⋀*n*. *n*∈*nodes* (*del-unEdge n1 w v' G*)⟹ *n*≠*n2*
⟹ *n*≠*v'*⟹ *even*(*degree n* (*del-unEdge n1 w v' G*))
using *del-UnEdge-node*[*of - n1 w v' G*] *even-except-two* *valid-unMultigraph.degree-frame*[*OF*
‹*valid-unMultigraph G*› ‹*finite* (*edges G*)›, *of - n1 v' w*]
by *force*
obtain *G1 G2* **where**
G1-nodes: *nodes G1*={*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge n1 w v' G*) *n ps n1*}
and *G1-edges*: *edges G1*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge n1 w v' G*) ∧ *n*∈*nodes G1*
∧ *n'*∈*nodes G1*}
and *G2-nodes*:*nodes G2*={*n*. ∃*ps*. *valid-graph.is-path* (*del-unEdge n1 w v' G*) *n ps v'*}
and *G2-edges*:*edges G2*={(*n*,*e*,*n'*). (*n*,*e*,*n'*)∈*edges* (*del-unEdge n1 w v' G*) ∧ *n*∈*nodes G2*
∧ *n'*∈*nodes G2*}
and *G1-G2-edges-union*:*edges G1* ∪ *edges G2* = *edges* (*del-unEdge n1 w v' G*)
and *edges G1* ∩ *edges G2*={}
and *G1-G2-nodes-union*:*nodes G1* ∪ *nodes G2*=*nodes* (*del-unEdge n1 w v' G*)
and *nodes G1* ∩ *nodes G2*={}
and *valid-unMultigraph G1*
and *valid-unMultigraph G2*
and *valid-unMultigraph.connected G1*
and *valid-unMultigraph.connected G2*
using *valid-unMultigraph.connectivity-split*[*OF* ‹*valid-unMultigraph G*›
‹*valid-unMultigraph.connected G*› *not-conneted* ‹($n1$,w,v')∈*edges G*›]
.
have *n2*∈*nodes G2* **using** *extend-distinct-path*
proof −
have *finite* (*edges* (*del-unEdge n1 w v' G*))
unfolding *del-unEdge-def* **using** ‹*finite* (*edges G*)› **by** *auto*
moreover have *num-of-odd-nodes* (*del-unEdge n1 w v' G*) = *2*
by (*metis* ‹($n1$, w, v') ∈ *edges G*› ‹(v', w, $n1$) ∈ *edges G*› ‹*num-of-odd-nodes G* = *2*›
‹$v' \neq n2$› ‹*valid-graph G*› *del-UnEdge-even-odd* *delete-edge-sym*

even-except-two
⟨*finite* (*edges G*)⟩ ⟨*finite* (*nodes G*)⟩ ⟨*valid-unMultigraph G*⟩
n1(*2*) *valid-graph.E-validD*(*2*) *valid-unMultigraph.no-id*)
ultimately have ∃ *ps. valid-unMultigraph.is-trail* (*del-unEdge n1 w v′ G*) *n2 ps v′*
using *valid-unMultigraph.path-between-odds*[*OF valid0*,*of n2 v′*,*OF odd-n2 odd-v′*] ⟨*v′≠n2*⟩
by *auto*
hence ∃ *ps. valid-graph.is-path* (*del-unEdge n1 w v′ G*) *n2 ps v′*
by (*metis valid0 valid-unMultigraph.is-trail-intro*)
thus *?thesis* **using** *G2-nodes* **by** *auto*
qed
have *v′∈nodes G2*
proof −
have *valid-graph.is-path* (*del-unEdge n1 w v′ G*) *v′* [] *v′*
by (*metis* (*full-types*) ⟨(*n1*, *w*, *v′*) ∈ *edges G*⟩ ⟨*valid-graph G*⟩ *del-UnEdge-node valid0′ valid-graph.E-validD*(*2*) *valid-graph.is-path-simps*(*1*))
thus *?thesis* **by** (*metis* (*lifting*) *G2-nodes mem-Collect-eq*)
qed
have *edges-subset*:*edges* (*del-unEdge n1 w v′ G*) ⊂ *edges G*
using ⟨(*n1*,*w*,*v′*)∈*edges G*⟩ ⟨(*v′*,*w*,*n1*)∈*edges G*⟩
unfolding *del-unEdge-def* **by** *auto*
hence *card* (*edges G1*) < *card* (*edges G*)
by (*metis G1-G2-edges-union inf-sup-absorb* ⟨*finite* (*edges G*)⟩ *less-infI2 psubset-card-mono*)
moreover have *finite* (*edges G1*)
by (*metis* (*full-types*) *G1-G2-edges-union edges-subset finite-Un finite-subset* ⟨*finite* (*edges G*)⟩ *less-imp-le*)
moreover have *finite* (*nodes G1*)
using *G1-G2-nodes-union* ⟨*finite* (*nodes G*)⟩
unfolding *del-unEdge-def*
by (*metis* (*full-types*) *finite-Un select-convs*(*1*))
moreover have *n1∈nodes G1*
proof −
have *valid-graph.is-path* (*del-unEdge n1 w v′ G*) *n1* [] *n1*
by (*metis* (*full-types*) *del-UnEdge-node n1*(*1*) *valid0′ valid-graph.is-path-simps*(*1*))
thus *?thesis* **by** (*metis* (*lifting*) *G1-nodes mem-Collect-eq*)
qed
moreover hence *nodes G1* ≠ {} **by** *auto*
moreover have *num-of-odd-nodes G1* = *0*
proof −
have ∀ *n∈nodes G1*. *even*(*degree n* (*del-unEdge n1 w v′ G*))
using *even-except-two′ odd-v′ odd-n2* ⟨*n2∈nodes G2*⟩ ⟨*nodes G1* ∩ *nodes G2* = {}⟩
⟨*v′∈nodes G2*⟩
by (*metis* (*full-types*) *G1-G2-nodes-union Un-iff disjoint-iff-not-equal*)
moreover have *valid-graph G2*
using ⟨*valid-unMultigraph G2*⟩ *valid-unMultigraph-def*
by *auto*

ultimately have ∀ *n*∈*nodes G1*. *even*(*degree n G1*)
using *sub-graph-degree-frame*[*of G2 G1 del-unEdge n1 w v′ G*]
by (*metis G1-G2-edges-union* ‹*nodes G1* ∩ *nodes G2* = {}›)
thus *?thesis* **unfolding** *num-of-odd-nodes-def odd-nodes-set-def*
by (*metis* (*lifting*) *card-eq-0-iff empty-Collect-eq*)
qed
ultimately obtain *ps1* **where** *ps1*:*valid-unMultigraph.is-Eulerian-trail G1 n1 ps1 n1*
using ‹*valid-unMultigraph G1*› ‹*valid-unMultigraph.connected G1*› *less.hyps*[*of G1*]
by (*metis valid-unMultigraph.is-Eulerian-circuit-def*)
have *card* (*edges G2*) < *card* (*edges G*)
by (*metis G1-G2-edges-union* ‹*finite* (*edges G*)› *edges-subset inf-sup-absorb less-infI2*
psubset-card-mono sup-commute)
moreover have *finite* (*edges G2*)
by (*metis* (*full-types*) *G1-G2-edges-union edges-subset finite-Un* ‹*finite* (*edges G*)› *less-le*
rev-finite-subset)
moreover have *finite* (*nodes G2*)
by (*metis* (*mono-tags*) *G1-G2-nodes-union del-UnEdge-node le-sup-iff* ‹*finite* (*nodes G*)›
rev-finite-subset subsetI)
moreover have *nodes G2* ≠ {} **using** ‹*v′*∈*nodes G2*› **by** *auto*
moreover have *num-of-odd-nodes G2* = *2*
proof −
have ∀ *n*∈*nodes G2*. *n*∉{*n2*,*v′*}⟶*even*(*degree n* (*del-unEdge n1 w v′ G*))
using *even-except-two′*
by (*metis* (*full-types*) *G1-G2-nodes-union Un-iff insert-iff*)
moreover have *valid-graph G1*
using ‹*valid-unMultigraph G1*› *valid-unMultigraph-def* **by** *auto*
ultimately have ∀ *n*∈*nodes G2*. *n*∉{*n2*,*v′*}⟶*even*(*degree n G2*)
using *sub-graph-degree-frame*[*of G1 G2 del-unEdge n1 w v′ G*]
by (*metis G1-G2-edges-union Int-commute Un-commute* ‹*nodes G1* ∩ *nodes G2* = {}›)
hence ∀ *n*∈*nodes G2*. *n*∉{*n2*,*v′*}⟶*n*∉{*v* ∈ *nodes G2*. *odd* (*degree v G2*)}
by (*metis* (*lifting*) *mem-Collect-eq*)
moreover have *odd*(*degree n2 G2*)
using *sub-graph-degree-frame*[*of G1 G2 del-unEdge n1 w v′ G*]
by (*metis* (*hide-lams*, *no-types*) *G1-G2-edges-union* ‹*nodes G1* ∩ *nodes G2* = {}›
‹*valid-graph G1*› ‹*n2* ∈ *nodes G2*› *inf-assoc inf-bot-right inf-sup-absorb*
odd-n2 sup-bot-right sup-commute)
hence *n2*∈{*v* ∈ *nodes G2*. *odd* (*degree v G2*)}
by (*metis* (*lifting*) ‹*n2* ∈ *nodes G2*› *mem-Collect-eq*)
moreover have *odd*(*degree v′ G2*)
using *sub-graph-degree-frame*[*of G1 G2 del-unEdge n1 w v′ G*]
by (*metis G1-G2-edges-union Int-commute Un-commute* ‹*nodes G1* ∩ *nodes G2* = {}›

⟨v' ∈ *nodes G2*⟩ ⟨*valid-graph G1*⟩ *odd-v'*)
hence v'∈{v ∈ *nodes G2*. *odd* (*degree v G2*)}
by (*metis* (*full-types*) *Collect-conj-eq Collect-mem-eq Int-Collect* ⟨v' ∈ *nodes G2*⟩)
ultimately have {v ∈ *nodes G2*. *odd* (*degree v G2*)}={*n2*,v'}
using ⟨*finite* (*nodes G2*)⟩ **by** (*induct G2*,*auto*)
thus *?thesis* **using** ⟨v'≠*n2*⟩
unfolding *num-of-odd-nodes-def odd-nodes-set-def* **by** *auto*
qed
ultimately obtain *s t ps2* **where**
s: *s*∈*nodes G2 odd* (*degree s G2*)
and *t*:*t*∈*nodes G2 odd* (*degree t G2*)
and *s* ≠ *t*
and *s-ps2-t*: *valid-unMultigraph.is-Eulerian-trail G2 s ps2 t*
using ⟨*valid-unMultigraph G2*⟩ ⟨*valid-unMultigraph.connected G2*⟩ *less.hyps*[*of G2*]
by *auto*
moreover have *valid-graph G1*
using ⟨*valid-unMultigraph G1*⟩ *valid-unMultigraph-def* **by** *auto*
ultimately have (*s*=*n2*∧*t*=v')∨(*s*=v'∧*t*=*n2*)
using *odd-n2 odd-v' even-except-two'*
sub-graph-degree-frame[*of G1 G2* (*del-unEdge n1 w v' G*)]
by (*metis G1-G2-edges-union G1-G2-nodes-union UnI1* ⟨*nodes G1* ∩ *nodes G2* = {}⟩ *inf-commute*
sup.commute)
moreover have *merge-G1-G2*:⦇*nodes* = *nodes G1* ∪ *nodes G2*, *edges* = *edges G1* ∪ *edges G2* ∪
{(*n1*, *w*,v'),(v', *w*, *n1*)}⦈=*G*
proof −
have *edges* (*del-unEdge n1 w v' G*) ∪ {(*n1*, *w*, v'),(v', *w*, *n1*)} =*edges G*
using ⟨(*n1*,*w*,v')∈*edges G*⟩ ⟨(v',*w*,*n1*)∈*edges G*⟩
unfolding *del-unEdge-def* **by** *auto*
moreover have *nodes* (*del-unEdge n1 w v' G*)=*nodes G*
unfolding *del-unEdge-def* **by** *auto*
ultimately have ⦇*nodes* = *nodes* (*del-unEdge n1 w v' G*), *edges* =
edges (*del-unEdge n1 w v' G*) ∪ {(*n1*, *w*, v'), (v', *w*, *n1*)}⦈=*G*
by *auto*
moreover have ⦇*nodes* = *nodes G1* ∪ *nodes G2*, *edges* = *edges G1* ∪ *edges G2* ∪
{(*n1*, *w*, v'),(v', *w*, *n1*)}⦈=⦇*nodes* = *nodes* (*del-unEdge n1 w v' G*),*edges*
= *edges* (*del-unEdge n1 w v' G*) ∪ {(*n1*, *w*, v'), (v', *w*, *n1*)}⦈
by (*metis G1-G2-edges-union G1-G2-nodes-union*)
ultimately show *?thesis* **by** *auto*
qed
moreover have *s*=*n2*⟹*t*=v'⟹*?thesis*
using *eulerian-split*[*of G1 G2 n1 ps1 n1 v'* (*rev-path ps2*) *n2 w*] *merge-G1-G2*
by (*metis* ⟨*edges G1* ∩ *edges G2* = {}⟩ ⟨*n1* ≠ *n2*⟩ ⟨*nodes G1* ∩ *nodes G2* = {}⟩
⟨*valid-unMultigraph G1*⟩ ⟨*valid-unMultigraph G2*⟩ *n1*(*1*) *n1*(*2*) *n2*(*1*)

n2(*2*) *ps1 s-ps2-t*
valid-unMultigraph.euclerian-rev)
moreover have *s*=*v*′⟹*t*=*n2*⟹*?thesis*
using *eulerian-split*[*of G1 G2 n1 ps1 n1 v*′ *ps2 n2 w*] *merge-G1-G2*
by (*metis* ‹*edges G1* ∩ *edges G2* = {}› ‹*n1* ≠ *n2*› ‹*nodes G1* ∩ *nodes G2* = {}›
‹*valid-unMultigraph G1*› ‹*valid-unMultigraph G2*› *n1*(*1*) *n1*(*2*) *n2*(*1*) *n2*(*2*) *ps1 s-ps2-t*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show ∃ *v*∈*nodes G*. ∃ *v*′∈*nodes G*.∃ *ps*. *odd* (*degree v G*) ∧ *odd* (*degree v*′ *G*) ∧ *v* ≠ *v*′
∧ *valid-unMultigraph.is-Eulerian-trail G v ps v*′
by *auto*
next
case *less*
assume *finite* (*edges G*) **and** *finite* (*nodes G*) **and** *valid-unMultigraph G* **and** *nodes G*≠{}
and *valid-unMultigraph.connected G* **and** *num-of-odd-nodes G* = *0*
show ∀ *v*∈*nodes G*. ∃ *ps*. *valid-unMultigraph.is-Eulerian-circuit G v ps v*
proof (*rule*,*cases card* (*nodes G*)=*1*)
fix *v* **assume** *v*∈*nodes G*
assume *card* (*nodes G*) = *1*
hence *nodes G*={*v*}
using ‹*v* ∈ *nodes G*› *card-Suc-eq*[*of nodes G 0*] *empty-iff insert-iff* [*of - v*]
by *auto*
have *edges G*={}
proof (*rule ccontr*)
assume *edges G* ≠ {}
then obtain *e1 e2 e3* **where** *e*:(*e1*,*e2*,*e3*)∈*edges G* **by** (*metis ex-in-conv prod-cases3*)
hence *e1*=*e3* **using** ‹*nodes G*={*v*}›
by (*metis* (*hide-lams*, *no-types*) *append-Nil2 valid-unMultigraph.is-trail-rev*
valid-unMultigraph.is-trail.simps(*1*) ‹*valid-unMultigraph G*› *singletonE*
valid-unMultigraph.is-trail-split valid-unMultigraph.singleton-distinct-path)
thus *False* **by** (*metis e* ‹*valid-unMultigraph G*› *valid-unMultigraph.no-id*)
qed
hence *valid-unMultigraph.is-Eulerian-circuit G v* [] *v*
by (*metis* ‹*nodes G* = {*v*}› *insert-subset* ‹*valid-unMultigraph G*› *rem-unPath.simps*(*1*)
subsetI valid-unMultigraph.is-trail.simps(*1*)
valid-unMultigraph.is-Eulerian-circuit-def
valid-unMultigraph.is-Eulerian-trail-def)
thus ∃ *ps*. *valid-unMultigraph.is-Eulerian-circuit G v ps v* **by** *auto*
next
fix *v* **assume** *v*∈*nodes G*
assume *card* (*nodes G*) ≠ *1*
moreover have *card* (*nodes G*)≠*0* **using** ‹*nodes G*≠{}›
by (*metis card-eq-0-iff* ‹*finite* (*nodes G*)›)
ultimately have *card* (*nodes G*) ≥*2* **by** *auto*

then obtain n **where** *card* (*nodes* G) = *Suc* (*Suc* n)
by (*metis le-iff-add add-2-eq-Suc*)
hence $\exists\, n{\in}$*nodes* G. $n{\neq}v$ **by** (*auto dest*!: *card-eq-SucD*)
then obtain v' w **where** $(v,w,v'){\in}$*edges* G
proof −
assume *pre*:$\bigwedge w\ v'$. $(v,\ w,\ v') \in$ *edges* $G \Longrightarrow$ *thesis*
assume $\exists\, n{\in}$*nodes* G. $n \neq v$
then obtain *ps* **where** *ps*:$\exists\, v'$. *valid-graph.is-path* G v *ps* $v' \wedge$ *ps*$\neq$*Nil*
using *valid-unMultigraph-def*
by (*metis* (*full-types*) ‹$v \in$ *nodes* G› ‹*valid-unMultigraph* G› *valid-graph.is-path.simps*(*1*)
‹*valid-unMultigraph.connected* G› *valid-unMultigraph.connected-def*)
then obtain *v0* w v' **where** $\exists\, ps'$. *ps*=*Cons* (*v0*,w,v') ps' **by** (*metis*
neq-Nil-conv prod-cases3)
hence *v0*=v
using *valid-unMultigraph-def*
by (*metis* ‹*valid-unMultigraph* G› *ps valid-graph.is-path.simps*(*2*))
hence $(v,w,v'){\in}$*edges* G
using *valid-unMultigraph-def*
by (*metis* ‹$\exists\, ps'$. *ps* = (*v0*, w, v') # ps'› ‹*valid-unMultigraph* G› *ps*
valid-graph.is-path.simps(*2*))
thus *?thesis* **by** (*metis pre*)
qed
have *all-even*:$\forall\, x{\in}$*nodes* G. *even*(*degree* x G)
using ‹*finite* (*nodes* G)› ‹*num-of-odd-nodes* G = *0*›
unfolding *num-of-odd-nodes-def odd-nodes-set-def* **by** *auto*
have *odd-v*: *odd* (*degree* v (*del-unEdge* v w v' G))
using ‹$v \in$ *nodes* G› *all-even valid-unMultigraph.del-UnEdge-even*[*OF*
‹*valid-unMultigraph* G›
‹$(v,\ w,\ v') \in$ *edges* G› ‹*finite* (*edges* G)›]
unfolding *odd-nodes-set-def* **by** *auto*
have *odd-v'*: *odd* (*degree* v' (*del-unEdge* v w v' G))
using *valid-unMultigraph.del-UnEdge-even'*[*OF* ‹*valid-unMultigraph* G› ‹$(v,$
$w,\ v') \in$ *edges* G›
‹*finite* (*edges* G)›]
all-even *valid-graph.E-validD*(*2*)[*OF* - ‹$(v,\ w,\ v') \in$ *edges* G›]
‹*valid-unMultigraph* G›
unfolding *valid-unMultigraph-def odd-nodes-set-def*
by *auto*
have *valid-unMulti*:*valid-unMultigraph* (*del-unEdge* v w v' G)
by (*metis del-unEdge-valid* ‹*valid-unMultigraph* G›)
moreover have *valid-graph*: *valid-graph* (*del-unEdge* v w v' G)
using *valid-unMultigraph-def del-undirected*
by (*metis* ‹*valid-unMultigraph* G› *delete-edge-valid*)
moreover have *fin-E'*: *finite*(*edges* (*del-unEdge* v w v' G))
using ‹*finite*(*edges* G)› **unfolding** *del-unEdge-def* **by** *auto*
moreover have *fin-V'*: *finite*(*nodes* (*del-unEdge* v w v' G))
using ‹*finite*(*nodes* G)› **unfolding** *del-unEdge-def* **by** *auto*
moreover have *less-card*:*card*(*edges* (*del-unEdge* v w v' G))<*card*(*edges* G)
unfolding *del-unEdge-def* **using** ‹$(v,w,v'){\in}$*edges* G›

by (*metis Diff-insert2 card-Diff2-less* ‹*finite* (*edges G*)› ‹*valid-unMultigraph G*›
select-convs(*2*) *valid-unMultigraph.corres*)
moreover have *num-of-odd-nodes* (*del-unEdge v w v′ G*) = *2*
using ‹*valid-unMultigraph G*› ‹*num-of-odd-nodes G* = *0*› ‹*v* ∈ *nodes G*› *all-even*
del-UnEdge-even-even[*OF* ‹*valid-unMultigraph G*› ‹*finite* (*edges G*)› ‹*finite* (*nodes G*)›
‹(*v*, *w*, *v′*) ∈ *edges G*›] *valid-graph.E-validD*(*2*)[*OF* - ‹(*v*, *w*, *v′*) ∈ *edges G*›]
unfolding *valid-unMultigraph-def*
by *auto*
moreover have *valid-unMultigraph.connected* (*del-unEdge v w v′ G*)
using ‹*finite* (*edges G*)› ‹*finite* (*nodes G*)› ‹*valid-unMultigraph G*›
‹*valid-unMultigraph.connected G*›
by (*metis* ‹(*v*, *w*, *v′*) ∈ *edges G*› *all-even valid-unMultigraph.del-unEdge-even-connectivity*)
moreover have *nodes*(*del-unEdge v w v′ G*)≠{}
by (*metis* ‹*v* ∈ *nodes G*› *del-UnEdge-node emptyE*)
ultimately obtain *n1 n2 ps* **where**
n1-n2:
n1∈*nodes* (*del-unEdge v w v′ G*)
n2∈*nodes* (*del-unEdge v w v′ G*)
odd (*degree n1* (*del-unEdge v w v′ G*))
odd (*degree n2* (*del-unEdge v w v′ G*))
n1≠*n2*
and
ps-eulerian:
valid-unMultigraph.is-Eulerian-trail (*del-unEdge v w v′ G*) *n1 ps n2*
by (*metis* ‹*num-of-odd-nodes* (*del-unEdge v w v′ G*) = *2*› *less.hyps*(*1*))
have *n1*=*v*⟹*n2*=*v′*⟹*valid-unMultigraph.is-Eulerian-circuit G v* (*ps*@[(*v′*,*w*,*v*)]) *v*
using *ps-eulerian*
by (*metis* ‹(*v*, *w*, *v′*) ∈ *edges G*› *delete-edge-sym* ‹*valid-unMultigraph G*›
valid-unMultigraph.corres valid-unMultigraph.eulerian-cons′
valid-unMultigraph.is-Eulerian-circuit-def)
moreover have *n1*=*v′*⟹*n2*=*v*⟹∃*ps*. *valid-unMultigraph.is-Eulerian-circuit G v ps v*
by (*metis* ‹(*v*, *w*, *v′*) ∈ *edges G*› ‹*valid-unMultigraph G*› *ps-eulerian*
valid-unMultigraph.eulerian-cons valid-unMultigraph.is-Eulerian-circuit-def)
moreover have (*n1*=*v*∧*n2*=*v′*)∨(*n2*=*v*∧*n1*=*v′*)
by (*metis* (*mono-tags*) *all-even del-UnEdge-node insert-iff* ‹*finite* (*edges G*)›
‹*valid-unMultigraph G*› *n1-n2*(*1*) *n1-n2*(*2*) *n1-n2*(*3*) *n1-n2*(*4*) *n1-n2*(*5*) *singletonE*
valid-unMultigraph.degree-frame)
ultimately show ∃*ps*. *valid-unMultigraph.is-Eulerian-circuit G v ps v* **by** *auto*
qed
qed
end

theory *FriendshipTheory*
imports *MoreGraph HOL−Number-Theory.Number-Theory*
begin

10 Common steps

definition (**in** *valid-unSimpGraph*) *non-adj* :: $'v \Rightarrow 'v \Rightarrow bool$ **where**
$non\text{-}adj\ v\ v' \equiv v \in V \wedge v' \in V \wedge v \neq v' \wedge \neg adjacent\ v\ v'$

lemma (**in** *valid-unSimpGraph*) *no-quad*:
assumes $\bigwedge v\ u.\ v \in V \Longrightarrow u \in V \Longrightarrow v \neq u \Longrightarrow \exists!\ n.\ adjacent\ v\ n \wedge adjacent\ u\ n$
shows $\neg\ (\exists\, v1\ v2\ v3\ v4.\ v2 \neq v4 \wedge v1 \neq v3 \wedge adjacent\ v1\ v2 \wedge adjacent\ v2\ v3 \wedge$
$adjacent\ v3\ v4$
$\wedge\ adjacent\ v4\ v1)$
proof
assume $\exists\, v1\ v2\ v3\ v4.\ v2 \neq v4 \wedge v1 \neq v3 \wedge adjacent\ v1\ v2 \wedge adjacent\ v2\ v3 \wedge$
$adjacent\ v3\ v4 \wedge adjacent\ v4\ v1$
then obtain *v1 v2 v3 v4* **where**
$v2 \neq v4$ $v1 \neq v3$ *adjacent v1 v2 adjacent v2 v3 adjacent v3 v4 adjacent v4 v1*
by *auto*
hence $\exists!n.\ adjacent\ v1\ n \wedge adjacent\ v3\ n$ **using** *assms*[*of v1 v3*] **by** *auto*
thus *False*
by (*metis* ‹*adjacent v1 v2*› ‹*adjacent v2 v3*› ‹*adjacent v3 v4*› ‹*adjacent v4 v1*›
‹$v2 \neq v4$›
adjacent-sym)
qed

lemma *even-card-set*:
assumes *finite A* **and** $\forall x \in A.\ f\ x \in A \wedge f\ x \neq x \wedge f\ (f\ x) = x$
shows *even*(*card A*) **using** *assms*
proof (*induct card A arbitrary*:*A rule*:*less-induct*)
case *less*
have $A = \{\} \Longrightarrow ?case$ **by** *auto*
moreover have $A \neq \{\} \Longrightarrow ?case$
proof −
assume $A \neq \{\}$
then obtain *x* **where** $x \in A$ **by** *auto*
hence $f\ x \in A$ **and** $f\ x \neq x$ **by** (*metis less.prems*(*2*))+
obtain *B* **where** $B{:}B = A - \{x, f\ x\}$ **by** *auto*
hence *finite B* **using** ‹*finite A*› **by** *auto*
moreover have $card\ B < card\ A$ **using** *B* ‹*finite A*›
by (*metis Diff-insert* ‹$f\ x \in A$› ‹$x \in A$› *card-Diff2-less*)
moreover have $\forall x \in B.\ f\ x \in B \wedge f\ x \neq x \wedge f\ (f\ x) = x$
proof
fix *y* **assume** $y \in B$
hence $y \in A$ **using** *B* **by** *auto*
hence $f\ y \neq y$ **and** $f\ (f\ y) = y$ **by** (*metis less.prems*(*2*))+

moreover have *f y*∈*B*
proof (*rule ccontr*)
assume *f y*∉*B*
have *f y*∈*A* **by** (*metis* ‹*y* ∈ *A*› *less.prems*(*2*))
hence *f y*∈{*x*, *f x*} **by** (*metis B DiffI* ‹*f y* ∉ *B*›)
moreover have *f y*=*x* ⟹ *False*
by (*metis B Diff-iff Diff-insert2* ‹*f* (*f y*) = *y*› ‹*y* ∈ *B*› *singleton-iff*)
moreover have *f y*= *f x*⟹ *False*
by (*metis B Diff-iff* ‹*x* ∈ *A*› ‹*y* ∈ *B*› *insertCI less.prems*(*2*))
ultimately show *False* **by** *auto*
qed
ultimately show *f y* ∈ *B* ∧ *f y* ≠ *y* ∧ *f* (*f y*) = *y* **by** *auto*
qed
ultimately have *even* (*card B*) **by** (*metis* (*full-types*) *less.hyps*)
moreover have {*x*,*f x*}⊆*A* **using** ‹*f x*∈*A*› ‹*x*∈*A*› **by** *auto*
moreover have *card* {*x*, *f x*} = *2* **using** ‹*f x*≠*x*› **by** *auto*
ultimately show *?case* **using** *B* ‹*finite A*› *card-mono* [*of A* {*x*, *f x*}]
by (*simp add*: *card-Diff-subset*)
qed
ultimately show *?case* **by** *metis*
qed

lemma (**in** *valid-unSimpGraph*) *even-degree*:
assumes *friend-assm*:⋀*v u*. *v*∈*V* ⟹ *u*∈*V* ⟹ *v*≠*u* ⟹ ∃! *n*. *adjacent v n* ∧ *adjacent u n*
and *finite E*
shows ∀ *v*∈*V*. *even*(*degree v G*)
proof
fix *v* **assume** *v*∈*V*
obtain *f* **where** *f*:*f* = (λ*n*. (*SOME v'*. *n*∈*V* ⟶*n*≠*v*⟶*adjacent n v'* ∧ *adjacent v v'*)) **by** *auto*
have ⋀*n*. *n*∈*V* ⟶ *n*≠*v* ⟶ (∃ *v'*. *adjacent n v'* ∧ *adjacent v v'*)
proof (*rule*,*rule*)
fix *n* **assume** *n* ∈ *V n* ≠ *v*
hence ∃!*v'*. *adjacent n v'* ∧ *adjacent v v'*
using *friend-assm*[*of n v*] ‹*v*∈*V*› **unfolding** *non-adj-def* **by** *auto*
thus ∃ *v'*. *adjacent n v'* ∧ *adjacent v v'* **by** *auto*
qed
hence *f-ex*:⋀*n*. (∃ *v'*. *n*∈*V* ⟶ *n*≠*v* ⟶ *adjacent n v'* ∧ *adjacent v v'*) **by** *auto*
have ∀ *x*∈{*n*. *adjacent v n*}. *f x*∈{*n*. *adjacent v n*} ∧ *f x*≠ *x* ∧ *f* (*f x*)=*x*
proof
fix *x* **assume** *x* ∈ {*n*. *adjacent v n*}
hence *adjacent v x* **by** *auto*
have *f x*∈{*n*. *adjacent v n*}
using *someI-ex*[*OF f-ex*,*of x*]
by (*metis* ‹*adjacent v x*› *adjacent-V*(*2*) *adjacent-no-loop f mem-Collect-eq*)
moreover have *f x*≠*x*
using *someI-ex*[*OF f-ex*,*of x*]

```
        by (metis ‹adjacent v x› adjacent-V(2) adjacent-no-loop f)
      moreover have f (f x)=x
        proof (rule ccontr)
          assume f (f x)≠x
          have adjacent (f x) (f (f x))
            using someI-ex[OF f-ex,of f x]
              by (metis (full-types) adjacent-V(2) adjacent-no-loop calculation(1) f
mem-Collect-eq)
          moreover have adjacent (f (f x)) v
              using someI-ex[OF f-ex,of f x] by (metis adjacent-V(1) adjacent-sym
calculation f)
          moreover have adjacent x (f x)
              using someI-ex[OF f-ex,of x] by (metis ‹adjacent v x› adjacent-V(2)
adjacent-no-loop f)
          moreover have v≠f x
            by (metis ‹f x ∈ {n. adjacent v n}› adjacent-no-loop mem-Collect-eq)
          ultimately show False
            using no-quad[OF friend-assm] using ‹adjacent v x› ‹f (f x)≠x›
            by metis
        qed
      ultimately show f x ∈ {n. adjacent v n} ∧ f x ≠ x ∧ f (f x) = x by auto
    qed
  moreover have finite {n. adjacent v n} by (metis adjacent-finite assms(2))
  ultimately have even (card {n. adjacent v n})
    using even-card-set[of {n. adjacent v n} f] by auto
  thus even(degree v G) by (metis assms(2) degree-adjacent)
qed

lemma (in valid-unSimpGraph) degree-two-windmill:
  assumes friend-assm:⋀v u. v∈V ⟹ u∈V ⟹ v≠u ⟹ ∃! n. adjacent v n ∧
adjacent u n
      and finite E and card V≥2
  shows (∃ v∈V. degree v G = 2) ⟷(∃ v. ∀ n∈V. n≠v ⟶ adjacent v n)
proof
  assume ∃ v∈V. degree v G = 2
  then obtain v where degree v G=2 by auto
  hence card {n. adjacent v n}=2 using degree-adjacent[OF ‹finite E›,of v] by
auto
  then obtain v1 v2 where v1v2:{n. adjacent v n}={v1,v2} and v1≠v2
    proof −
      obtain v1 S where {n. adjacent v n} = insert v1 S and  v1 ∉ S and card
S = 1
         using ‹card {n. adjacent v n}=2› card-Suc-eq[of {n. adjacent v n} 1] by
auto
      then obtain v2 where S=insert v2 {}
        using card-Suc-eq[of S 0] by auto
      hence {n. adjacent v n}={v1,v2} and v1≠v2
        using ‹{n. adjacent v n} = insert v1 S› ‹v1 ∉ S› by auto
      thus ?thesis using that[of v1 v2] by auto
```

qed
have *adjacent v1 v2*
proof −
obtain *n* **where** *adjacent v n adjacent v1 n* **using** *friend-assm*[*of v v1*]
by (*metis* (*full-types*) *adjacent-V*(*2*) *adjacent-sym insertI1 mem-Collect-eq v1v2*)
hence $n \in \{n.\ adjacent\ v\ n\}$ **by** *auto*
moreover have $n \neq v1$ **by** (*metis* ‹*adjacent v1 n*› *adjacent-no-loop*)
ultimately have $n = v2$ **using** *v1v2* **by** *auto*
thus *?thesis* **by** (*metis* ‹*adjacent v1 n*›)
qed
have *v1v2-adj*: $\forall x \in V.\ x \in \{n.\ adjacent\ v1\ n\} \cup \{n.\ adjacent\ v2\ n\}$
proof
fix *x* **assume** $x \in V$
have $x = v \Longrightarrow x \in \{n.\ adjacent\ v1\ n\} \cup \{n.\ adjacent\ v2\ n\}$
by (*metis Un-iff adjacent-sym insertI1 mem-Collect-eq v1v2*)
moreover have $x \neq v \Longrightarrow x \in \{n.\ adjacent\ v1\ n\} \cup \{n.\ adjacent\ v2\ n\}$
proof −
assume $x \neq v$
then obtain *y* **where** *adjacent v y adjacent x y*
using *friend-assm*[*of v x*]
by (*metis Collect-empty-eq* ‹$x \in V$› *adjacent-V*(*1*) *all-not-in-conv insertCI v1v2*)
hence $y = v1 \vee y = v2$ **using** *v1v2* **by** *auto*
thus $x \in \{n.\ adjacent\ v1\ n\} \cup \{n.\ adjacent\ v2\ n\}$ **using** ‹*adjacent x y*›
by (*metis UnI1 UnI2 adjacent-sym mem-Collect-eq*)
qed
ultimately show $x \in \{n.\ adjacent\ v1\ n\} \cup \{n.\ adjacent\ v2\ n\}$ **by** *auto*
qed
have $\{n.\ adjacent\ v1\ n\} - \{v2, v\} = \{\} \Longrightarrow \exists v.\ \forall n \in V.\ n \neq v \longrightarrow adjacent\ v\ n$
proof (*rule exI*[*of - v2*],*rule*,*rule*)
fix *n* **assume** *v1-adj*: $\{n.\ adjacent\ v1\ n\} - \{v2, v\} = \{\}$ **and** $n \in V$ **and** $n \neq v2$
have $n \in \{n.\ adjacent\ v2\ n\}$
proof (*cases* $n = v$)
case *True*
show *?thesis* **by** (*metis True adjacent-sym insertI1 insert-commute mem-Collect-eq v1v2*)
next
case *False*
have $n \notin \{n.\ adjacent\ v1\ n\}$ **by** (*metis DiffI False* ‹$n \neq v2$› *empty-iff insert-iff v1-adj*)
thus *?thesis* **by** (*metis Un-iff* ‹$n \in V$› *v1v2-adj*)
qed
thus *adjacent v2 n* **by** *auto*
qed
moreover have $\{n.\ adjacent\ v2\ n\} - \{v1, v\} = \{\} \Longrightarrow \exists v.\ \forall n \in V.\ n \neq v \longrightarrow adjacent\ v\ n$
proof (*rule exI*[*of - v1*],*rule*,*rule*)

fix n **assume** *v2-adj*:$\{n.\ adjacent\ v2\ n\} - \{v1,\ v\} = \{\}$ **and** $n \in V$ **and** $n \neq v1$
have $n \in \{n.\ adjacent\ v1\ n\}$
proof (*cases* $n=v$)
case *True*
show *?thesis* **by** (*metis True adjacent-sym insertI1 mem-Collect-eq v1v2*)
next
case *False*
have $n \notin \{n.\ adjacent\ v2\ n\}$ **by** (*metis DiffI False* ‹$n \neq v1$› *empty-iff insert-iff v2-adj*)
thus *?thesis* **by** (*metis Un-iff* ‹$n \in V$› *v1v2-adj*)
qed
thus *adjacent v1 n* **by** *auto*
qed
moreover have $\{n.\ adjacent\ v1\ n\}-\{v2,v\}\neq\{\} \Longrightarrow \{n.\ adjacent\ v2\ n\}-\{v1,v\}\neq\{\} \Longrightarrow False$
proof −
assume $\{n.\ adjacent\ v1\ n\} - \{v2,\ v\} \neq \{\}$ $\{n.\ adjacent\ v2\ n\} - \{v1,\ v\} \neq \{\}$
then obtain a b **where** a:$a \in \{n.\ adjacent\ v1\ n\} - \{v2,\ v\}$
and b:$b \in \{n.\ adjacent\ v2\ n\} - \{v1,\ v\}$
by *auto*
have $a=b \Longrightarrow False$
proof −
assume $a=b$
have *adjacent v1 a* **using** a **by** *auto*
moreover have *adjacent a v2* **using** b ‹$a=b$› *adjacent-sym* **by** *auto*
moreover have $a \neq v$ **by** (*metis DiffD2* ‹$a = b$› *b doubleton-eq-iff insertI1*)
moreover have *adjacent v2 v*
by (*metis* (*full-types*) *adjacent-sym inf-sup-aci*(*5*) *insertI1 insert-is-Un mem-Collect-eq v1v2*)
moreover have *adjacent v v1* **by** (*metis* (*full-types*) *insertI1 mem-Collect-eq v1v2*)
ultimately show *False* **using** *no-quad*[*OF friend-assm*]
using ‹$v1 \neq v2$› **by** *auto*
qed
moreover have $a \neq b \Longrightarrow False$
proof −
assume $a \neq b$
moreover have $a \in V$ **using** a **by** (*metis DiffD1 adjacent-V*(*2*) *mem-Collect-eq*)
moreover have $b \in V$ **using** b **by** (*metis DiffD1 adjacent-V*(*2*) *mem-Collect-eq*)
ultimately obtain c **where** *adjacent a c adjacent b c*
using *friend-assm*[*of a b*] **by** *auto*
hence $c \in \{n.\ adjacent\ v1\ n\} \cup \{n.\ adjacent\ v2\ n\}$
by (*metis* (*full-types*) *adjacent-V*(*2*) *v1v2-adj*)
moreover have $c \in \{n.\ adjacent\ v1\ n\} \Longrightarrow False$
proof −
assume $c \in \{n.\ adjacent\ v1\ n\}$

hence *adjacent v1 c* **by** *auto*
moreover have *adjacent c b* **by** (*metis* ‹*adjacent b c*› *adjacent-sym*)
moreover have *adjacent b v2*
by (*metis* (*full-types*) *Diff-iff adjacent-sym b mem-Collect-eq*)
moreover have *adjacent v2 v1* **by** (*metis* ‹*adjacent v1 v2*› *adjacent-sym*)
moreover have $c \neq v2$
proof (*rule ccontr*)
assume $\neg\ c \neq v2$
hence $c = v2$ **by** *auto*
hence *adjacent v2 a* **by** (*metis* ‹*adjacent a c*› *adjacent-sym*)
moreover have *adjacent v2 v*
by (*metis adjacent-sym insert-iff mem-Collect-eq v1v2*)
moreover have *adjacent v1 v*
using *adjacent-sym v1v2* **by** *auto*
moreover have *adjacent v1 a* **by** (*metis* (*full-types*) *Diff-iff a mem-Collect-eq*)
ultimately have $a = v$ **using** *friend-assm*[*of v1 v2*]
by (*metis* ‹$v1 \neq v2$› *adjacent-V*(*1*))
thus *False* **using** *a* **by** *auto*
qed
moreover have $b \neq v1$ **by** (*metis DiffD2 b insertI1*)
ultimately show *False* **using** *no-quad*[*OF friend-assm*] **by** *auto*
qed
moreover have $c \in \{n.\ adjacent\ v2\ n\} \Longrightarrow False$
proof −
assume $c \in \{n.\ adjacent\ v2\ n\}$
hence *adjacent c v2* **by** (*metis adjacent-sym mem-Collect-eq*)
moreover have *adjacent a c* **using** ‹*adjacent a c*› **.**
moreover have *adjacent v1 a* **by** (*metis* (*full-types*) *Diff-iff a mem-Collect-eq*)
moreover have *adjacent v2 v1* **by** (*metis* ‹*adjacent v1 v2*› *adjacent-sym*)
moreover have $c \neq v1$
proof (*rule ccontr*)
assume $\neg\ c \neq v1$
hence $c = v1$ **by** *auto*
hence *adjacent v1 b* **by** (*metis* ‹*adjacent b c*› *adjacent-sym*)
moreover have *adjacent v2 v*
by (*metis adjacent-sym insert-iff mem-Collect-eq v1v2*)
moreover have *adjacent v1 v*
using *adjacent-sym v1v2* **by** *auto*
moreover have *adjacent v2 b* **by** (*metis Diff-iff b mem-Collect-eq*)
ultimately have $b = v$ **using** *friend-assm*[*of v1 v2*]
by (*metis* ‹$v1 \neq v2$› *adjacent-V*(*1*))
thus *False* **using** *b* **by** *auto*
qed
moreover have $a \neq v2$ **by** (*metis DiffD2 a insertI1*)
ultimately show *False* **using** *no-quad*[*OF friend-assm*] **by** *auto*
qed
ultimately show *False* **by** *auto*

```
        qed
      ultimately show False by auto
    qed
  ultimately show ∃ v. ∀ n∈V. n ≠ v ⟶ adjacent v n by auto
next
  assume ∃ v. ∀ n∈V. n ≠ v ⟶ adjacent v n
  then obtain v where v:∀ n∈V. n ≠ v ⟶ adjacent v n by auto
  obtain v1 where v1∈V v1≠v
    proof (cases v∈V)
      case False
      have V≠{} using ‹2≤card V› by auto
      then obtain v1 where v1∈V by auto
      thus ?thesis using False that[of v1] by auto
    next
      case True
      then obtain S where  V = insert v S v ∉ S
        using  mk-disjoint-insert[OF True] by auto
      moreover have finite V using ‹2≤card V›
        by (metis add-leE card-infinite not-one-le-zero numeral-Bit0 numeral-One)
      ultimately have 1≤card S
        using ‹2≤card V›  card.insert[of S v]  finite-insert[of v S] by auto
      hence S≠{} by auto
      then obtain v1 where v1∈S by auto
      hence v1≠v using ‹v∉S› by auto
      thus thesis using that[of v1] ‹v1∈S› ‹V=insert v S› by auto
    qed
  hence v∈V using v by (metis adjacent-V(1))
  then obtain v2 where adjacent v1 v2 adjacent v v2 using friend-assm[of v v1]

    by (metis ‹v1 ∈ V› ‹v1 ≠ v›)
  have degree v1 G≠2 ⟹ False
    proof –
      assume degree v1 G≠2
      hence card {n. adjacent v1 n}≠2 by (metis assms(2) degree-adjacent)
      have {v,v2} ⊆ {n. adjacent v1 n}
         by (metis ‹ adjacent v1 v2 › ‹ v1 ∈ V › ‹ v1 ≠ v › adjacent-sym bot-least
insert-subset
          mem-Collect-eq v)
      moreover have v≠v2 using ‹adjacent v v2› adjacent-no-loop by auto
      hence card {v,v2} = 2 by auto
      ultimately have card {n. adjacent v1 n} ≥2
        using adjacent-finite[OF ‹finite E›, of v1] by (metis card-mono)
       hence card {n. adjacent v1 n} ≥3 using ‹card {n. adjacent v1 n}≠2› by
auto
      then obtain v3 where v3∈{n. adjacent v1 n} and v3∉{v,v2}
        using ‹{v,v2} ⊆ {n. adjacent v1 n}› ‹card {v, v2} = 2›
        by (metis ‹card {n. adjacent v1 n} ≠ 2› subsetI subset-antisym)
      hence adjacent v1 v3 by auto
      moreover have adjacent v3 v using v
```

by (*metis* ‹$v3 \notin \{v, v2\}$› *adjacent-V*(*2*) *adjacent-sym calculation insertCI*)
moreover have *adjacent v v2* **using** ‹*adjacent v v2*› **.**
moreover have *adjacent v2 v1* **using** ‹*adjacent v1 v2*› *adjacent-sym* **by** *auto*
moreover have $v1 \neq v$ **using** ‹$v1 \neq v$› **.**
moreover have $v3 \neq v2$ **by** (*metis* ‹$v3 \notin \{v, v2\}$› *insert-subset subset-insertI*)
ultimately show *False* **using** *no-quad*[*OF friend-assm*] **by** *auto*
qed
thus $\exists\, v \in V.$ *degree v G* = *2* **using** ‹$v1 \in V$› **by** *auto*
qed

lemma (**in** *valid-unSimpGraph*) *regular*:
assumes *friend-assm*:$\bigwedge v\ u.\ v \in V \Longrightarrow u \in V \Longrightarrow v \neq u \Longrightarrow \exists!\ n.$ *adjacent v n* $\wedge$ *adjacent u n*
and *finite E* **and** *finite V* **and** $\neg(\exists\, v \in V.$ *degree v G* = *2*)
shows $\exists\, k.\ \forall\, v \in V.$ *degree v G* = *k*
proof –
{ fix *v u* **assume** *non-adj v u*
obtain *v-adj* **where** *v-adj*:*v-adj*=$\{n.$ *adjacent v n*$\}$ **by** *auto*
obtain *u-adj* **where** *u-adj*:*u-adj*=$\{n.$ *adjacent u n*$\}$ **by** *auto*
obtain *f* **where** *f*:*f* = ($\lambda n.$ (*SOME* $v'.\ n \in V \longrightarrow n \neq u \longrightarrow$ *adjacent n* $v' \wedge$ *adjacent u* v')) **by** *auto*
have $\bigwedge n.\ n \in V \longrightarrow n \neq u \longrightarrow (\exists\, v'.$ *adjacent n* $v' \wedge$ *adjacent u* v')
proof (*rule*,*rule*)
fix *n* **assume** $n \in V\ n \neq u$
hence $\exists! v'.$ *adjacent n* $v' \wedge$ *adjacent u* v'
using *friend-assm*[*of n u*] ‹*non-adj v u*› **unfolding** *non-adj-def* **by** *auto*
thus $\exists\, v'.$ *adjacent n* $v' \wedge$ *adjacent u* v' **by** *auto*
qed
hence *f-ex*:$\bigwedge n.\ (\exists\, v'.\ n \in V \longrightarrow n \neq u \longrightarrow$ *adjacent n* $v' \wedge$ *adjacent u* v') **by** *auto*
obtain *v-adj-u* **where** *v-adj-u*:*v-adj-u*= *f* ‘ *v-adj* **by** *auto*
have *finite u-adj* **using** *u-adj adjacent-finite*[*OF* ‹*finite E*›] **by** *auto*
have *finite v-adj* **using** *v-adj adjacent-finite*[*OF* ‹*finite E*›] **by** *auto*
hence *finite v-adj-u* **using** *v-adj-u adjacent-finite*[*OF* ‹*finite E*›] **by** *auto*
have *inj-on f v-adj* **unfolding** *inj-on-def*
proof (*rule ccontr*)
assume $\neg$ ($\forall\, x \in$*v-adj*. $\forall\, y \in$*v-adj*. *f x* = *f y* $\longrightarrow$ *x* = *y*)
then obtain *x y* **where** $x \in$*v-adj* $y \in$*v-adj* *f x*=*f y* $x \neq y$ **by** *auto*
have $x \in V$ **by** (*metis* ‹$x \in$ *v-adj*› *adjacent-V*(*2*) *mem-Collect-eq v-adj*)
moreover have $x \neq u$ **by** (*metis* ‹*non-adj v u*› ‹$x \in$ *v-adj*› *mem-Collect-eq non-adj-def v-adj*)
ultimately have *adjacent* (*f x*) *u* **and** *adjacent x* (*f x*)
using *someI-ex*[*OF f-ex*[*of x*]] *adjacent-sym* **by** (*metis f*)+
hence *f x* $\neq$ *v* **by** (*metis* ‹*non-adj v u*› *non-adj-def*)
have $y \in V$ **by** (*metis* ‹$y \in$ *v-adj*› *adjacent-V*(*2*) *mem-Collect-eq v-adj*)
moreover have $y \neq u$ **by** (*metis* ‹*non-adj v u*› ‹$y \in$ *v-adj*› *mem-Collect-eq non-adj-def v-adj*)
ultimately have *adjacent y* (*f y*) **using** *someI-ex*[*OF f-ex*[*of y*]] **by** (*metis f*)

hence $x \neq y \wedge v \neq f\,x \wedge$ *adjacent v x* $\wedge$ *adjacent x* $(f\,x) \wedge$ *adjacent* $(f\,x)$ y
$\wedge$ *adjacent y v*
using ‹$x \in$*v-adj*› ‹$y \in$*v-adj*› ‹$f\ x=f\ y$› ‹$x \neq y$› ‹*adjacent x* $(f\ x)$› *v-adj* *adjacent-sym* ‹$f\,x \neq v$›
by *auto*
thus *False* **using** *no-quad*[*OF friend-assm*] **by** *auto*
qed
then have *card v-adj =card v-adj-u* **by** (*metis card-image v-adj-u*)
moreover have *v-adj-u* $\subseteq$ *u-adj*
proof
fix x **assume** $x \in$*v-adj-u*
then obtain y **where** $y \in$*v-adj*
and $x = ($*SOME* $v'.\ y \in V \longrightarrow y \neq u \longrightarrow$ *adjacent y* $v' \wedge$ *adjacent u* $v')$
using *f image-def v-adj-u* **by** *auto*
hence $y \in V \longrightarrow y \neq u \longrightarrow$ *adjacent y x* $\wedge$ *adjacent u x* **using** *someI-ex*[*OF f-ex*[*of y*]]
by *auto*
moreover have $y \in V$ **by** (*metis* ‹$y \in$ *v-adj*› *adjacent-V*(*2*) *mem-Collect-eq v-adj*)
moreover have $y \neq u$ **by** (*metis* ‹*non-adj v u*› ‹$y \in$ *v-adj*› *mem-Collect-eq non-adj-def v-adj*)
ultimately have *adjacent u x* **by** *auto*
thus $x \in$*u-adj* **unfolding** *u-adj* **by** *auto*
qed
moreover have *card v-adj=degree v G* **using** *degree-adjacent*[*OF* ‹*finite E*›, *of v*] *v-adj* **by** *auto*
moreover have *card u-adj=degree u G* **using** *degree-adjacent*[*OF* ‹*finite E*›, *of u*] *u-adj* **by** *auto*
ultimately have *degree v G* $\leq$ *degree u G* **using** ‹*finite u-adj*›
by (*metis* ‹*inj-on f v-adj*› *card-inj-on-le v-adj-u*) **}**
hence *non-adj-degree*:$\bigwedge v\ u.$ *non-adj v u* $\Longrightarrow$ *degree v G* $=$ *degree u G*
by (*metis adjacent-sym antisym non-adj-def*)
have *card V=3* $\Longrightarrow$ *?thesis*
proof
assume *card V=3*
then obtain *v1 v2 v3* **where** $V=\{v1,v2,v3\}$ $v1 \neq v2$ $v2 \neq v3$ $v1 \neq v3$
proof –
obtain *v1 S1* **where** *VS1*:$V =$ *insert v1 S1* **and** $v1 \notin S1$ **and** *card S1* $= 2$
using *card-Suc-eq*[*of V 2*] ‹*card V=3*› **by** *auto*
then obtain *v2 S2* **where** *S1S2*:$S1 =$ *insert v2 S2* **and** $v2 \notin S2$ **and** *card S2* $= 1$
using *card-Suc-eq*[*of S1 1*] **by** *auto*
then obtain *v3* **where** $S2=\{v3\}$
using *card-Suc-eq*[*of S2 0*] **by** *auto*
hence $V=\{v1,v2,v3\}$ **using** *VS1 S1S2* **by** *auto*
moreover have $v1 \neq v2$ $v2 \neq v3$ $v1 \neq v3$**using** *VS1 S1S2* ‹$v1 \notin S1$› ‹$v2 \notin S2$› ‹$S2=\{v3\}$› **by** *auto*

ultimately show *?thesis* **using** *that* **by** *auto*
qed
obtain *n* **where** *adjacent v1 n adjacent v2 n*
using *friend-assm[of v1 v2]* **by** (*metis* ‹$V = \{v1, v2, v3\}$› ‹$v1 \neq v2$› *insertI1 insertI2*)
moreover hence *n=v3*
using ‹$V = \{v1, v2, v3\}$› *adjacent-V(2) adjacent-no-loop*
by (*metis* (*mono-tags*) *empty-iff insertE*)
moreover obtain n' **where** *adjacent v2 n' adjacent v3 n'*
using *friend-assm[of v2 v3]* **by** (*metis* ‹$V = \{v1, v2, v3\}$› ‹$v2 \neq v3$› *insertI1 insertI2*)
moreover hence $n'=v1$
using ‹$V = \{v1, v2, v3\}$› *adjacent-V(2) adjacent-no-loop*
by (*metis* (*mono-tags*) *empty-iff insertE*)
ultimately have *adjacent v1 v2* **and** *adjacent v2 v3* **and** *adjacent v3 v1*
using *adjacent-sym* **by** *auto*
have *degree v1 G=2*
proof –
have $v2 \in \{n.\ adjacent\ v1\ n\}$ **and** $v3 \in \{n.\ adjacent\ v1\ n\}$ **and** $v1 \notin \{n.\ adjacent\ v1\ n\}$
using ‹*adjacent v1 v2*› ‹*adjacent v3 v1*› *adjacent-sym*
by (*auto,metis adjacent-no-loop*)
hence $\{n.\ adjacent\ v1\ n\}=\{v2,v3\}$ **using** ‹$V=\{v1,v2,v3\}$› **by** *auto*
thus *?thesis* **using** *degree-adjacent[OF* ‹*finite E*›*,of v1]* ‹$v2 \neq v3$› **by** *auto*
qed
moreover have *degree v2 G=2*
proof –
have $v1 \in \{n.\ adjacent\ v2\ n\}$ **and** $v3 \in \{n.\ adjacent\ v2\ n\}$ **and** $v2 \notin \{n.\ adjacent\ v2\ n\}$
using ‹*adjacent v1 v2*› ‹*adjacent v2 v3*› *adjacent-sym*
by (*auto,metis adjacent-no-loop*)
hence $\{n.\ adjacent\ v2\ n\}=\{v1,v3\}$ **using** ‹$V=\{v1,v2,v3\}$› **by** *force*
thus *?thesis* **using** *degree-adjacent[OF* ‹*finite E*›*,of v2]* ‹$v1 \neq v3$› **by** *auto*

qed
moreover have *degree v3 G=2*
proof –
have $v1 \in \{n.\ adjacent\ v3\ n\}$ **and** $v2 \in \{n.\ adjacent\ v3\ n\}$ **and** $v3 \notin \{n.\ adjacent\ v3\ n\}$
using ‹*adjacent v3 v1*› ‹*adjacent v2 v3*› *adjacent-sym*
by (*auto,metis adjacent-no-loop*)
hence $\{n.\ adjacent\ v3\ n\}=\{v1,v2\}$ **using** ‹$V=\{v1,v2,v3\}$› **by** *force*
thus *?thesis* **using** *degree-adjacent[OF* ‹*finite E*›*,of v3]* ‹$v1 \neq v2$› **by** *auto*
qed
ultimately show $\forall v \in V.\ degree\ v\ G = 2$ **using** ‹$V=\{v1,v2,v3\}$› **by** *auto*
qed
moreover have $card\ V=2 \Longrightarrow False$
proof –
assume $card\ V=2$

```
    obtain v1 v2 where V={v1,v2} v1≠v2
      proof –
        obtain v1 S1 where VS1:V = insert v1 S1 and v1 ∉ S1 and card S1
= 1
          using card-Suc-eq[of V 1] ‹card V=2› by auto
        then obtain v2 where S1={v2}
          using card-Suc-eq[of S1 0] by auto
        hence V={v1,v2} using VS1 by auto
        moreover have v1≠v2 using ‹v1∉S1› ‹S1={v2}› by auto
        ultimately show ?thesis using that by auto
      qed
    then obtain v3 where adjacent v1 v3 adjacent v2 v3
      using friend-assm[of v1 v2] by auto
    hence v3≠v2 and v3≠v1 by (metis adjacent-no-loop)+
    hence v3∉V using ‹V={v1,v2}› by auto
    thus False using ‹adjacent v1 v3› by (metis (full-types) adjacent-V(2))
  qed
moreover have card V=1 ⟹ ?thesis
  proof
    assume card V=1
    then obtain v1 where V={v1} using card-eq-SucD[of V 0] by auto
    have E={}
      proof (rule ccontr)
        assume E≠{}
        then obtain x1 x2 x3 where x:(x1,x2,x3)∈E by auto
        hence x1=v1 and x3=v1 using ‹V={v1}› E-validD by auto
        thus False using no-id x by auto
      qed
    hence degree v1 G=0 unfolding degree-def by auto
    thus ∀v∈V. degree v G =0 using ‹V={v1}›by auto
  qed
moreover have card V=0 ⟹ ?thesis
  proof –
    assume card V=0
    hence V={} using ‹finite V› by auto
    thus ?thesis by auto
  qed
moreover have card V ≥4 ⟹ ¬(∃v u. non-adj v u) ⟹ False
  proof –
    assume ¬(∃v u. non-adj v u) card V≥4
     hence non-non-adj:⋀v u. v∉V ∨ u∉V ∨ v=u ∨ adjacent v u unfolding
non-adj-def by auto
    obtain v1 v2 v3 v4 where v1∈V v2∈V v3∈V v4∈V v1≠v2 v1≠v3 v1≠v4
            v2≠v3 v2≠v4 v3≠v4
      proof –
        obtain v1 B1 where V = insert v1 B1  v1 ∉ B1  card B1 ≥3
          using ‹card V≥4› card-le-Suc-iff[of 3 V] by auto
        then obtain v2 B2 where B1 = insert v2 B2  v2 ∉ B2  card B2 ≥2
          using card-le-Suc-iff[of 2 B1] by auto
```

then obtain *v3 B3* **where** *B2= insert v3 B3 v3∉B3 card B3≥1*
using *card-le-Suc-iff[of 1 B2]* **by** *auto*
then obtain *v4 B4* **where** *B3=insert v4 B4 v4∉B4*
using *card-le-Suc-iff[of 0 B3]* **by** *auto*
have *v1∈V* **by** (*metis* ‹*V = insert v1 B1*› *insert-subset order-refl*)
moreover have *v2∈V*
by (*metis* ‹*B1 = insert v2 B2*› ‹*V = insert v1 B1*› *insert-subset subset-insertI*)
moreover have *v3∈V*
by (*metis* ‹*B1 = insert v2 B2*› ‹*B2 = insert v3 B3*› ‹*V = insert v1 B1*› *insert-iff*)
moreover have *v4∈V*
by (*metis* ‹*B1 = insert v2 B2*› ‹*B2 = insert v3 B3*› ‹*B3 = insert v4 B4*›
‹*V = insert v1 B1*› *insert-iff*)
moreover have *v1≠v2*
by (*metis* (*full-types*) ‹*B1 = insert v2 B2*› ‹*v1 ∉ B1*› *insertI1*)
moreover have *v1≠v3*
by (*metis* ‹*B1 = insert v2 B2*› ‹*B2 = insert v3 B3*› ‹*v1 ∉ B1*› *insert-iff*)
moreover have *v1≠v4*
by (*metis* ‹*B1 = insert v2 B2*› ‹*B2 = insert v3 B3*› ‹*B3 = insert v4 B4*› ‹*v1 ∉ B1*›
insert-iff)
moreover have *v2≠v3*
by (*metis* (*full-types*) ‹*B2 = insert v3 B3*› ‹*v2 ∉ B2*› *insertI1*)
moreover have *v2≠v4*
by (*metis* ‹*B2 = insert v3 B3*› ‹*B3 = insert v4 B4*› ‹*v2 ∉ B2*› *insert-iff*)
moreover have *v3≠v4*
by (*metis* (*full-types*) ‹*B3 = insert v4 B4*› ‹*v3 ∉ B3*› *insertI1*)
ultimately show *?thesis* **using** *that* **by** *auto*
qed
hence *adjacent v1 v2* **using** *non-non-adj* **by** *auto*
moreover have *adjacent v2 v3* **using** *non-non-adj* **by** (*metis* ‹*v2 ∈ V*› ‹*v2 ≠ v3*› ‹*v3 ∈ V*›)
moreover have *adjacent v3 v4* **using** *non-non-adj* **by** (*metis* ‹*v3 ∈ V*› ‹*v3 ≠ v4*› ‹*v4 ∈ V*›)
moreover have *adjacent v4 v1* **using** *non-non-adj* **by** (*metis* ‹*v1 ∈ V*› ‹*v1 ≠ v4*› ‹*v4 ∈ V*›)
ultimately show *False* **using** *no-quad[OF friend-assm]*
by (*metis* ‹*v1 ≠ v3*› ‹*v2 ≠ v4*›)
qed
moreover have *card V≥4* ⟹ (∃ *v u. non-adj v u*) ⟹ *?thesis*
proof −
assume (∃ *v u. non-adj v u*) *card V≥4*
then obtain *v u* **where** *non-adj v u* **by** *auto*
then obtain *w* **where** *adjacent v w* **and** *adjacent u w*
and *unique*:∀ *n. adjacent v n* ∧ *adjacent u n* ⟶ *n=w*
using *friend-assm[of v u]* **unfolding** *non-adj-def* **by** *auto*
have ∀ *n∈V. degree n G = degree v G*
proof

```
        fix n assume n∈V
        moreover have n=v ⟹ degree n G = degree v G by auto
        moreover have n=u ⟹ degree n G = degree v G
          using non-adj-degree ‹non-adj v u› by auto
        moreover have n≠v ⟹ n≠u ⟹ n≠w ⟹ degree n G = degree v G
          proof −
            assume n≠v n≠u n≠w
          have non-adj v n ⟹ degree n G = degree v G by (metis non-adj-degree)
            moreover have non-adj u n ⟹ degree n G = degree v G
              by (metis ‹non-adj v u› non-adj-degree)
              moreover have ¬non-adj u n ⟹ ¬non-adj v n ⟹ degree n G =
degree v G
              by (metis ‹n ∈ V› ‹n ≠ w› ‹non-adj v u› non-adj-def unique)
            ultimately show degree n G = degree v G by auto
          qed
        moreover have n=w ⟹ degree n G = degree v G
          proof −
            assume n=w
            moreover have ¬(∃ v. ∀ n∈V. n≠v ⟶ adjacent v n)
            using ‹card V≥4› degree-two-windmill assms(2) assms(4) friend-assm
              by auto
            ultimately obtain w1 where w1∈V w1≠w non-adj w w1
              by (metis ‹n∈V› non-adj-def)
            have w1=v ⟹ degree n G = degree v G
              by (metis ‹n = w› ‹non-adj w w1› non-adj-degree)
            moreover have w1=u ⟹ degree n G = degree v G
              by (metis ‹adjacent u w› ‹non-adj w w1› adjacent-sym non-adj-def)
            moreover have w1≠u ⟹ w1≠v ⟹ degree n G = degree v G
                  by (metis ‹n = w› ‹non-adj v u› ‹non-adj w w1› non-adj-def
non-adj-degree unique)
            ultimately show degree n G = degree v G by auto
          qed
        ultimately show degree n G = degree v G by auto
      qed
    thus ?thesis by auto
  qed
 ultimately show ?thesis by force
qed
```

11 Exclusive steps for combinatorial proofs

```
fun (in valid-unSimpGraph) adj-path:: 'v ⇒ 'v list ⇒bool where
  adj-path v [] = (v∈V)
 | adj-path v (u#us)= (adjacent v u ∧ adj-path u us)

lemma (in valid-unSimpGraph) adj-path-butlast:
  adj-path v ps ⟹ adj-path v (butlast ps)
by (induct ps arbitrary:v,auto)
```

lemma (**in** *valid-unSimpGraph*) *adj-path-V*:
  adj-path v ps $\Longrightarrow$ *set ps* $\subseteq$ *V*
by (*induct ps arbitrary*:*v*, *auto*)

lemma (**in** *valid-unSimpGraph*) *adj-path-V'*:
  adj-path v ps $\Longrightarrow$ *v*$\in$ *V*
by (*induct ps arbitrary*:*v*, *auto*)

lemma (**in** *valid-unSimpGraph*) *adj-path-app*:
  adj-path v ps $\Longrightarrow$ *ps*$\neq$[] $\Longrightarrow$ *adjacent* (*last ps*) *u* $\Longrightarrow$ *adj-path v* (*ps*@[*u*])
proof (*induct ps arbitrary*:*v*)
  case *Nil*
  thus *?case* **by** *auto*
next
  case (*Cons x xs*)
  thus *?case* **by** (*cases xs*,*auto*)
qed

lemma (**in** *valid-unSimpGraph*) *adj-path-app'*:
  adj-path v (*ps* @ [*q*]) $\Longrightarrow$ *ps* $\neq$ [] $\Longrightarrow$ *adjacent* (*last ps*) *q*
proof (*induct ps arbitrary*:*v*)
  case *Nil*
  thus *?case* **by** *auto*
next
  case (*Cons x xs*)
  thus *?case* **by** (*cases xs*,*auto*)
qed

lemma *card-partition'*:
  assumes $\forall$ *v*$\in$*A*. *card* {*n*. *R v n*} = *k k*>*0 finite A*
    $\forall$ *v1 v2*. *v1*$\neq$*v2* $\longrightarrow$ {*n*. *R v1 n*} $\cap$ {*n*. *R v2 n*}={}
  shows *card* ($\bigcup$ *v*$\in$*A*. {*n*. *R v n*}) = *k* $*$ *card A*
proof −
  have $\bigwedge$*C*. *C* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A* $\Longrightarrow$ *card C* = *k*
    proof −
      fix *C* **assume** *C* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A*
      show *card C*=*k* **by** (*metis* (*mono-tags*) ‹*C* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A*› *assms*(*1*) *imageE*)
    qed
  moreover have $\bigwedge$*C1 C2*. *C1* $\in$(λ*x*. {*n*. *R x n*}) ‘ *A* $\Longrightarrow$ *C2* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A* $\Longrightarrow$ *C1* $\neq$ *C2*
      $\Longrightarrow$ *C1* $\cap$ *C2* = {}
    proof −
      fix *C1 C2* **assume** *C1* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A* *C2* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A* *C1* $\neq$ *C2*
      obtain *v1* **where** *v1*$\in$*A* *C1*={*n*. *R v1 n*} **by** (*metis* ‹*C1* $\in$ (λ*x*. {*n*. *R x n*}) ‘ *A*› *imageE*)
      obtain *v2* **where** *v2*$\in$*A* *C2*={*n*. *R v2 n*} **by** (*metis* ‹*C2* $\in$ (λ*x*. {*n*. *R x n*})

‘ A› imageE)
have v1≠v2 by (metis ‹C1 = {n. R v1 n}› ‹C1 ≠ C2› ‹C2 = {n. R v2 n}›)
thus C1 ∩ C2 ={} by (metis ‹C1 = {n. R v1 n}› ‹C2 = {n. R v2 n}› assms(4))
qed
moreover have ⋃((λx. {n. R x n}) ‘ A) = (⋃x∈A. {n. R x n}) by auto
moreover have finite ((λx. {n. R x n}) ‘ A) by (metis assms(3) finite-imageI)
moreover have finite (⋃((λx. {n. R x n}) ‘ A)) by (metis (full-types) assms(1)

assms(2) assms(3) card-eq-0-iff finite-UN-I less-nat-zero-code)
moreover have card A = card ((λx. {n. R x n}) ‘ A)
proof –
have inj-on (λx. {n. R x n}) A unfolding inj-on-def
using ‹∀ v1 v2. v1≠v2 ⟶ {n. R v1 n} ∩ {n. R v2 n}={}›
by (metis assms(1) assms(2) card-empty inf.idem less-le)
thus ?thesis by (metis card-image)
qed
ultimately show ?thesis using card-partition[of (λx. {n. R x n}) ‘ A] by auto
qed

lemma (in valid-unSimpGraph) path-count:
assumes k-adj:⋀v. v∈V ⟹ card {n. adjacent v n} = k and v∈V and finite V and k>0
shows card {ps. length ps=l ∧ adj-path v ps}=k^l
proof (induct l rule:nat.induct)
case zero
have {ps. length ps=0 ∧ adj-path v ps}={[]} using ‹v∈V› by auto
thus ?case by auto
next
case (Suc n)
obtain ext where ext: ext=(λps ps'. ps'≠[] ∧ (butlast ps'=ps) ∧ adj-path v ps') by auto
have ∀ps∈{ps. length ps = n ∧ adj-path v ps}. card {ps'. ext ps ps'} = k
proof
fix ps assume ps∈{ps. length ps = n ∧ adj-path v ps}
hence adj-path v ps and length ps = n by auto
obtain qs where qs:qs = {n. if ps=[] then adjacent v n else adjacent (last ps) n} by auto
hence card qs = k
proof (cases ps=[])
case True
thus ?thesis using qs k-adj[OF ‹v∈V›] by auto
next
case False
have last ps ∈ V using adj-path-V by (metis False ‹adj-path v ps› last-in-set subsetD)
thus ?thesis using k-adj[of last ps] False qs by auto
qed
obtain app where app:app=(λq. ps@[q]) by auto

```
      have app ‘ qs = {ps'. ext ps ps'}
        proof –
          have ⋀xs. xs∈ app ‘ qs ⟹ xs ∈ {ps'. ext ps ps'}
            proof (rule,cases ps=[])
              case True
              fix xs assume xs∈ app ‘ qs
              then obtain q where q∈ qs app q=xs by (metis imageE)
              hence  adjacent v q and xs=ps@[q] using qs app True by auto
              hence adj-path v xs
                  by (metis True adj-path.simps(1) adj-path.simps(2) adjacent-V(2)
append-Nil)
             moreover have butlast xs = ps using ‹xs=ps@[q]› by auto
              ultimately show ext ps xs using ext ‹xs=ps@[q]› by auto
            next
              case False
              fix xs assume xs∈ app ‘ qs
              then obtain q where q∈ qs app q=xs by (metis imageE)
              hence  adjacent (last ps) q using qs app False by auto
             hence adj-path v (ps@[q]) using ‹adj-path v ps› False adj-path-app by
auto
              hence adj-path v xs by (metis ‹app q = xs› app)
             moreover have butlast xs=ps by (metis ‹app q = xs› app butlast-snoc)
              ultimately show ext ps xs by (metis False butlast.simps(1) ext)
            qed
          moreover have ⋀xs. xs∈{ps'. ext ps ps'} ⟹ xs∈ app ‘ qs
            proof (cases ps=[])
              case True
              hence qs = {n. adjacent v n } using qs by auto
              fix xs assume xs ∈ {ps'. ext ps ps'}
             hence xs≠[] and (butlast xs=ps) and adj-path v xs using ext by auto
              thus xs ∈ app ‘ qs
                using True app ‹qs = {n. adjacent v n}›
                by (metis  adj-path.simps(2) append-butlast-last-id append-self-conv2
image-iff
                  mem-Collect-eq)
            next
              case False
              fix xs assume xs ∈ {ps'. ext ps ps'}
             hence xs≠[] and (butlast xs=ps) and adj-path v xs using ext by auto
              then obtain q where xs=ps@[q] by (metis append-butlast-last-id)
             hence adjacent (last ps) q using ‹adj-path v xs› False adj-path-app' by
auto
              thus xs ∈ app ‘ qs using qs
                  by (metis (lifting, full-types) False ‹xs = ps @ [q]› app imageI
mem-Collect-eq)
            qed
          ultimately show ?thesis by auto
        qed
      moreover have inj-on app qs using app unfolding inj-on-def by auto
```

ultimately show *card* {*ps'*. *ext ps ps'*}=*k* **by** (*metis* ‹*card qs* = *k*› *card-image*)
qed
moreover have ∀ *ps1 ps2*. *ps1*≠*ps2* ⟶ {*n*. *ext ps1 n*} ∩ {*n*. *ext ps2 n*}={}
using *ext* **by** *auto*
moreover have *finite* {*ps*. *length ps* = *n* ∧ *adj-path v ps*}
using *Suc.hyps assms* **by** (*auto intro*: *card-ge-0-finite*)
ultimately have *card* (⋃ *v*∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}. {*n*. *ext v n*})
= *k* ∗ *card* {*ps*. *length ps* = *n* ∧ *adj-path v ps*}
using *card-partition'*[*of* {*ps*. *length ps* = *n* ∧ *adj-path v ps*} *ext k*] ‹*k*>*0*› **by**
auto
moreover have {*ps*. *length ps* = *n*+*1* ∧ *adj-path v ps*}
=(⋃ *ps*∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}. {*ps'*. *ext ps ps'*})
proof −
have ⋀*xs*. *xs* ∈ {*ps*. *length ps* = *n* + *1* ∧ *adj-path v ps*} ⟹
xs ∈ (⋃ *ps*∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}. {*ps'*. *ext ps ps'*})
proof −
fix *xs* **assume** *xs* ∈ {*ps*. *length ps* = *n* + *1* ∧ *adj-path v ps*}
hence *length xs* = *n* +*1* **and** *adj-path v xs* **by** *auto*
hence *butlast xs* ∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}
using *adj-path-butlast length-butlast mem-Collect-eq* **by** *auto*
thus *xs* ∈ (⋃ *ps*∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}. {*ps'*. *ext ps ps'*})
using ‹*adj-path v xs*› ‹*length xs* = *n* + *1*› *UN-iff ext length-greater-0-conv*

mem-Collect-eq
by *auto*
qed
moreover have ⋀*xs* . *xs*∈(⋃ *ps*∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}. {*ps'*.
ext ps ps'}) ⟹
xs ∈ {*ps*. *length ps* = *n* + *1* ∧ *adj-path v ps*}
proof −
fix *xs* **assume** *xs*∈(⋃ *ps*∈{*ps*. *length ps* = *n* ∧ *adj-path v ps*}. {*ps'*. *ext ps
ps'*})
then obtain *ys* **where** *length ys*=*n adj-path v ys ext ys xs* **by** *auto*
hence *length xs*=*n*+*1* **using** *ext* **by** *auto*
thus *xs*∈{*ps*. *length ps* = *n* + *1* ∧ *adj-path v ps*}
by (*metis* (*lifting*, *full-types*) ‹*ext ys xs*› *ext mem-Collect-eq*)
qed
ultimately show *?thesis* **by** *fast*
qed
ultimately show *card* {*ps*. *length ps* = (*Suc n*) ∧ *adj-path v ps*} = *k* ^ (*Suc n*)
using *Suc.hyps* **by** *auto*
qed

lemma (**in** *valid-unSimpGraph*) *total-v-num*:
assumes *friend-assm*:⋀*v u*. *v*∈*V* ⟹ *u*∈*V* ⟹ *v*≠*u* ⟹ ∃! *n*. *adjacent v n* ∧
adjacent u n
and *finite E* **and** *finite V* **and** *V*≠{} **and** ∀ *v*∈*V*. *degree v G* = *k* **and** *k*>*0*
shows *card V*= *k*∗*k* − *k* +*1*
proof −

have *k-adj*:$\bigwedge v.\ v\in V \Longrightarrow$*card* ({*n. adjacent v n*})=*k* **by** (*metis assms*(*2*) *assms*(*5*) *degree-adjacent*)
obtain *v* **where** $v\in V$ **using** ‹$V\neq${}› **by** *auto*
obtain *l2-eq-v* **where** *l2-eq-v*: *l2-eq-v*={*ps. length ps=2* $\wedge$ *adj-path v ps* $\wedge$ *last ps=v*} **by** *auto*
have *card l2-eq-v=k*
proof −
obtain *hds* **where** *hds*:*hds= hd‘ l2-eq-v* **by** *auto*
moreover have *hds*={*n. adjacent v n*}
proof −
have $\bigwedge x.\ x\in hds \Longrightarrow x\in$ {*n. adjacent v n*}
proof
fix *x* **assume** $x\in hds$
then obtain *ps* **where** *hd ps=x length ps=2 adj-path v ps last ps=v*
using *hds l2-eq-v* **by** *auto*
thus *adjacent v x*
by (*metis* (*full-types*) *adj-path.simps*(*2*) *list.sel*(*1*) *length-0-conv neq-Nil-conv*
zero-neq-numeral)
qed
moreover have $\bigwedge x.\ x\in${*n. adjacent v n*} $\Longrightarrow x\in hds$
proof −
fix *x* **assume** $x\in${*n. adjacent v n*}
obtain *ps* **where** *ps*=[*x*,*v*] **by** *auto*
hence *hd ps=x* **and** *length ps=2* **and** *adj-path v ps* **and** *last ps=v*
using ‹$x\in${*n. adjacent v n*}› *adjacent-sym* **by** *auto*
thus $x\in hds$ **by** (*metis* (*lifting*, *mono-tags*) *hds image-eqI l2-eq-v mem-Collect-eq*)
qed
ultimately show *hds*={*n. adjacent v n*} **by** *auto*
qed
moreover have *inj-on hd l2-eq-v* **unfolding** *inj-on-def*
proof (*rule*+)
fix *x y* **assume** $x \in$ *l2-eq-v* $y \in$ *l2-eq-v hd x = hd y*
hence *length x=2* **and** *last x=last y* **and** *length y=2*
using *l2-eq-v* **by** *auto*
hence *x*!*1*=*y*!*1*
using *last-conv-nth*[*of x*] *last-conv-nth*[*of y*] **by** *force*
moreover have *x*!*0*=*y*!*0*
using ‹*hd x=hd y*› ‹*length x=2*› ‹*length y=2*›
by(*metis hd-conv-nth length-greater-0-conv*)
ultimately show *x=y* **using** ‹*length x=2*› ‹*length y=2*›
using *nth-equalityI*[*of x y*]
by (*metis One-nat-def less-2-cases*)
qed
ultimately show *card l2-eq-v=k* **using** *k-adj*[*OF* ‹$v\in V$›] **by** (*metis card-image*)
qed
obtain *l2-neq-v* **where** *l2-neq-v*:*l2-neq-v*={*ps. length ps=2* $\wedge$ *adj-path v ps* $\wedge$ *last ps*$\neq$*v*} **by** *auto*

have *card l2-neq-v = k∗k−k*
proof −
obtain *l2-v* **where** *l2-v*:*l2-v*={*ps. length ps=2∧ adj-path v ps*} **by** *auto*
hence *card l2-v=k∗k* **using** *path-count*[*OF k-adj*,*of v 2*] ‹*0<k*› ‹*finite V*› ‹*v∈V*›
by (*simp add*: *power2-eq-square*)
hence *finite l2-v* **using** ‹*k>0*› **by** (*metis card-infinite mult-is-0 neq0-conv*)
moreover have *l2-v=l2-neq-v ∪ l2-eq-v* **using** *l2-v l2-neq-v l2-eq-v* **by** *auto*
moreover have *l2-neq-v ∩ l2-eq-v* ={} **using** *l2-neq-v l2-eq-v* **by** *auto*
ultimately have *card l2-neq-v = card l2-v − card l2-eq-v*
by (*metis Int-commute Nat.add-0-right Un-commute card-Diff-subset-Int card-Un-Int*
card-gt-0-iff diff-add-inverse finite-Diff finite-Un inf-sup-absorb
less-nat-zero-code)
thus *card l2-neq-v = k∗k−k* **using** ‹*card l2-eq-v=k*› **using** ‹*card l2-v=k∗k*› **by** *auto*
qed
moreover have *bij-betw last l2-neq-v* {*n. n∈V ∧ n≠v*}
proof −
have *last ' l2-neq-v =* {*n. n∈V ∧ n≠v*}
proof −
have ⋀*x. x∈ last' l2-neq-v ⟹ x∈*{*n. n∈V ∧ n≠v*}
proof
fix *x* **assume** *x∈last' l2-neq-v*
then obtain *ps* **where** *length ps = 2 adj-path v ps last ps=x last ps≠v*
using *l2-neq-v* **by** *auto*
hence (*last ps*)*∈V*
by (*metis* (*full-types*) *adj-path-V last-in-set length-0-conv rev-subsetD*
zero-neq-numeral)
thus *x ∈ V ∧ x ≠ v* **using** ‹*last ps=x*› ‹*last ps≠v*› **by** *auto*
qed
moreover have ⋀*x. x∈*{*n. n∈V ∧ n≠v*} *⟹ x∈ last' l2-neq-v*
proof −
fix *x* **assume** *x*:*x ∈* {*n ∈ V. n ≠ v*}
then obtain *y* **where** *adjacent v y adjacent x y*
using *friend-assm*[*of v x*] ‹*v∈V*› **by** *auto*
hence *adj-path v* [*y*,*x*] **using** *adjacent-sym*[*of x y*]**by** *auto*
hence [*y*,*x*]*∈l2-neq-v* **using** *l2-neq-v x* **by** *auto*
thus *x∈ last' l2-neq-v* **by** (*metis imageI last.simps not-Cons-self2*)
qed
ultimately show *?thesis* **by** *fast*
qed
moreover have *inj-on last l2-neq-v* **unfolding** *inj-on-def*
proof (*rule*,*rule*,*rule*)
fix *x y* **assume** *x ∈ l2-neq-v y ∈ l2-neq-v last x = last y*
hence *length x=2* **and** *adj-path v x* **and** *last x≠v* **and** *length y=2* **and** *adj-path v y*
and *last y≠v*
using *l2-neq-v* **by** *auto*

obtain *x1 x2 y1 y2* **where** *x*:*x*=[*x1*,*x2*] **and** *y*:*y*=[*y1*,*y2*]
  proof −
    { **fix** *l* **assume** *length l=2*
      obtain *h1 t* **where** *l=h1#t* **and** *length t=1*
        using ‹*length l=2*› *Suc-length-conv*[*of 1 l*] **by** *auto*
      then obtain *h2* **where** *t*=[*h2*]
        using *Suc-length-conv*[*of 0 t*] **by** *auto*
      have ∃ *h1 h2*. *l*=[*h1*,*h2*] **using** ‹*l=h1#t*› ‹*t*=[*h2*]› **by** *auto* }
    thus *?thesis* **using** *that* ‹*length x=2*› ‹*length y=2*› **by** *metis*
  qed
hence *x2≠v* **and** *y2≠v* **using** ‹*last x≠v*› ‹*last y≠v*› **by** *auto*
moreover have *adjacent v x1* **and** *adjacent x2 x1* **and** *x2∈V*
  using ‹*adj-path v x*› *x adjacent-sym* **by** *auto*
moreover have *adjacent v y1* **and** *adjacent y2 y1* **and** *y2∈V*
  using ‹*adj-path v y*› *y adjacent-sym* **by** *auto*
ultimately have *x1=y1* **using** *friend-assm* ‹*v∈V*›
  by (*metis* ‹*last x = last y*› *last-ConsL last-ConsR not-Cons-self2 x y*)
thus *x=y* **using** *x y* ‹*last x = last y*› **by** *auto*
qed
ultimately show *?thesis* **unfolding** *bij-betw-def* **by** *auto*
qed
hence *card l2-neq-v = card* {*n*. *n∈V* ∧ *n≠v*} **by** (*metis bij-betw-same-card*)
ultimately have *card* {*n*. *n∈V* ∧ *n≠v*}=*k*∗*k*−*k* **by** *auto*
moreover have *card V = card* {*n*. *n∈V*∧*n≠v*} + *card* {*v*}
  proof −
    have *V*={*n*. *n∈V* ∧ *n≠v*} ∪ {*v*} **using** ‹*v∈V*› **by** *auto*
    moreover have {*n*. *n∈V* ∧ *n≠v*} ∩ {*v*}={} **by** *auto*
    ultimately show *?thesis*
      using ‹*finite V*› *card-Un-disjoint*[*of* {*n* ∈ *V*. *n* ≠ *v*} {*v*}] *finite-Un*
      by *auto*
  qed
ultimately show *card V = k*∗*k*−*k*+*1* **by** *auto*
qed

lemma *rotate-eq*:*rotate1 xs=rotate1 ys* ⟹ *xs=ys*
proof (*induct xs arbitrary*:*ys*)
  case *Nil*
  thus *?case* **by** (*metis rotate1-is-Nil-conv*)
next
  case (*Cons n ns*)
  hence *ys≠*[] **by** (*metis list.distinct*(*1*) *rotate1-is-Nil-conv*)
  thus *?case* **using** *Cons* **by** (*metis butlast-snoc last-snoc list.exhaust rotate1.simps*(*2*))
qed

lemma *rotate-diff*:*rotate m xs=rotate n xs* ⟹*rotate* (*m*−*n*) *xs = xs*
proof (*induct m arbitrary*:*n*)
  case *0*
  thus *?case* **by** *auto*

```
next
  case (Suc m')
  hence n=0 ⟹ ?case by auto
  moreover have n≠0 ⟹?case
    proof −
      assume n≠0
      then obtain n' where n': n = Suc n' by (metis nat.exhaust)
      hence rotate m' xs = rotate n' xs
        using ‹rotate (Suc m') xs = rotate n xs› rotate-eq rotate-Suc
        by auto
      hence rotate (m' − n') xs = xs by (metis Suc.hyps)
      moreover have Suc m' − n = m'−n'
        by (metis n' diff-Suc-Suc)
      ultimately show ?case by auto
    qed
  ultimately show ?case by fast
qed

lemma (in valid-unSimpGraph) exist-degree-two:
  assumes friend-assm:⋀v u. v∈V ⟹ u∈V ⟹ v≠u ⟹ ∃! n. adjacent v n ∧
adjacent u n
      and finite E and finite V and card V≥2
  shows ∃ v∈V. degree v G = 2
proof (rule ccontr)
  assume ¬ (∃ v∈V. degree v G = 2)
  hence ⋀v. v∈V ⟹ degree v G≠2 by auto
 obtain k where k-adj: ⋀v. v∈V⟹ card {n. adjacent v n}=k using regular[OF
friend-assm]
    by (metis ‹¬ (∃ v∈V. degree v G = 2)› assms(2) assms(3) degree-adjacent)
  have k≥4
    proof −
      obtain v1 v2 where v1∈V v2∈V v1≠v2
           using ‹card V≥2› by (metis ‹¬(∃ v∈V. degree v G = 2)› assms(2)
degree-two-windmill)
      have k≠0
        proof
          assume k=0
          obtain v3 where adjacent v1 v3 using friend-assm[OF ‹v1∈V› ‹v2∈V›
‹v1≠v2›] by auto
           hence card {n. adjacent v1 n} ≠ 0 using adjacent-finite[OF ‹finite E›]
by auto
          moreover have card {n. adjacent v1 n} = 0 using k-adj[OF ‹v1∈V›]
            by (metis ‹k = 0›)
          ultimately show False by simp
        qed
      moreover have even k using even-degree[OF friend-assm]
        by (metis ‹v1 ∈ V› assms(2) degree-adjacent k-adj)
      hence k≠1 and k≠3 by auto
       moreover have k≠2 using ‹⋀v. v∈V ⟹ degree v G≠2› degree-adjacent
```

```
k-adj
        by (metis ‹v1 ∈ V› assms(2))
      ultimately show ?thesis by auto
    qed
  obtain T where T:T=(λl::nat. {ps. length ps = l+1 ∧ adj-path (hd ps) (tl ps)}) by auto
  have T-count:⋀l::nat. card (T l) = (k*k−k+1)*k^l using card-partition'
    proof −
      fix l::nat
      obtain ext where ext:ext=(λv ps. adj-path v (tl ps) ∧ hd ps=v ∧ length ps=l+1) by auto
      have ∀v∈V. card {ps. ext v ps} = k^l
        proof
          fix v assume v ∈ V
          have ⋀ps. ps∈tl ` {ps. ext v ps} ⟹ ps∈{ps. length ps=l ∧ adj-path v ps}
            proof −
              fix ps assume ps ∈ tl ` {ps. ext v ps}
              then obtain ps' where adj-path v (tl ps') hd ps'=v length ps'=l+1 ps=tl ps'
                using ext by auto
              hence adj-path v ps and length ps=l by auto
              thus ps∈{ps. length ps=l ∧ adj-path v ps} by auto
            qed
          moreover have ⋀ps. ps∈{ps. length ps=l ∧ adj-path v ps} ⟹ ps∈ tl ` {ps. ext v ps}
            proof −
              fix ps assume ps ∈ {ps. length ps = l ∧ adj-path v ps}
              hence length ps=l and adj-path v ps by auto
              moreover obtain ps' where ps'=v#ps by auto
              ultimately have adj-path v (tl ps') and hd ps'=v and length ps'=l+1 by auto
              thus ps ∈ tl ` {ps. ext v ps}
                by (metis ‹ps' = v # ps› ext imageI mem-Collect-eq list.sel(3))
            qed
          ultimately have tl ` {ps. ext v ps} = {ps. length ps=l ∧ adj-path v ps} by fast
          moreover have inj-on tl {ps. ext v ps} unfolding inj-on-def
            proof (rule,rule,rule)
              fix x y assume x ∈ Collect (ext v) y ∈ Collect (ext v) tl x = tl y
              hence hd x=hd y and x≠[] and y≠[]using ext by auto
              thus x=y using ‹tl x= tl y› by (metis list.sel(1,3) list.exhaust)
            qed
          moreover have card {ps. length ps=l ∧ adj-path v ps} = k^l
            using path-count[OF k-adj,of v l] ‹4 ≤ k› ‹v ∈ V› assms(3)
            by auto
          ultimately show card {ps. ext v ps} = k ^ l by (metis card-image)
        qed
      moreover have ∀v1 v2. v1 ≠ v2 ⟶ {n. ext v1 n} ∩ {n. ext v2 n} = {}
```

using *ext* **by** *auto*
moreover have $(\bigcup v\in V.\ \{n.\ ext\ v\ n\})=T\ l$
proof −
have $\bigwedge ps.\ ps\in(\bigcup v\in V.\ \{n.\ ext\ v\ n\}) \Longrightarrow ps\in T\ l$ **using** *T*
proof −
fix *ps* **assume** $ps\in(\bigcup v\in V.\ \{n.\ ext\ v\ n\})$
then obtain *v* **where** $v\in V$ *adj-path v* (*tl ps*) *hd ps* = *v length ps* = *l*
+ *1*
using *ext* **by** *auto*
hence *length ps* = *l* + *1* **and** *adj-path* (*hd ps*) (*tl ps*) **by** *auto*
thus $ps\in T\ l$ **using** *T* **by** *auto*
qed
moreover have $\bigwedge ps.\ ps\in T\ l \Longrightarrow ps\in(\bigcup v\in V.\ \{n.\ ext\ v\ n\})$
proof −
fix *ps* **assume** $ps\in T\ l$
hence *length ps* = *l* + *1* **and** *adj-path* (*hd ps*) (*tl ps*) **using** *T* **by**
auto
moreover then obtain *v* **where** *v*=*hd ps* $v\in V$
by (*metis adj-path.simps*(*1*) *adj-path.simps*(*2*) *adjacent-V*(*1*)
list.exhaust)
ultimately show $ps\in(\bigcup v\in V.\ \{n.\ ext\ v\ n\})$ **using** *ext* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately have *card* (*T l*) = *card V* ∗ *k*ˆ*l*
using *card-partition′*[*of V ext k*ˆ*l*] ‹ *4* ≤ *k* › *assms*(*3*) *mult.commute*
nat-one-le-power
by *auto*
moreover have *card V*=(*k* ∗ *k* − *k* + *1*)
using *total-v-num*[*OF friend-assm*,*of k*] *k-adj degree-adjacent* ‹*finite E*›
‹*finite V*›
‹*card V*≥*2*› ‹*4* ≤ *k*› *card-gt-0-iff*
by *force*
ultimately show *card* (*T l*) = (*k* ∗ *k* − *k* + *1*) ∗ *k* ˆ *l* **by** *auto*
qed
obtain *C* **where** *C*:*C*=(λ*l*::*nat*. {*ps*. *length ps* = *l*+*1* ∧ *adj-path* (*hd ps*) (*tl ps*)

∧ *adjacent* (*last ps*) (*hd ps*)}) **by** *auto*
obtain *C-star* **where** *C-star*:*C-star*=(λ*l*::*nat*. {*ps*. *length ps* = *l*+*1* ∧ *adj-path*
(*hd ps*) (*tl ps*)
∧ (*last ps*) = (*hd ps*)}) **by** *auto*
have $\bigwedge$*l*::*nat*. *card* (*C* (*l*+*1*)) = *k*∗ *card* (*C-star l*) + *card* (*T l* − *C-star l*)
proof −
fix *l*::*nat*
have *C* (*l*+*1*) = {*ps*. *length ps* = *l*+*2* ∧ *adj-path* (*hd ps*) (*tl ps*) ∧ *adjacent*
(*last ps*) (*hd ps*)
∧ *last* (*butlast ps*)=*hd ps*} ∪ {*ps*. *length ps* = *l*+*2* ∧ *adj-path* (*hd ps*) (*tl*
ps) ∧
adjacent (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)≠*hd ps*} **using** *C* **by** *auto*

moreover have $\{ps.\ length\ ps = l+2 \wedge adj\text{-}path\ (hd\ ps)\ (tl\ ps) \wedge adjacent\ (last\ ps)\ (hd\ ps)$
$\wedge\ last\ (butlast\ ps)=hd\ ps\} \cap \{ps.\ length\ ps = l+2 \wedge adj\text{-}path\ (hd\ ps)\ (tl\ ps) \wedge$
$adjacent\ (last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps)\neq hd\ ps\} =\{\}$ **by** *auto*
moreover have *finite* $(C\ (l+1))$
proof −
have $C\ (l+1) \subseteq T\ (l+1)$ **using** *C T* **by** *auto*
moreover have $(k * k - k + 1) * k\ \hat{}\ (l + 1)\neq 0$ **using** $\langle k\geq 4\rangle$ **by** *auto*
hence *finite* $(T\ (l+1))$ **using** *T-count*[*of* $l+1$] **by** (*metis card-infinite*)
ultimately show *?thesis* **by** (*metis finite-subset*)
qed
ultimately have $card\ (C\ (l+1)) = card\ \{ps.\ length\ ps = l+2 \wedge adj\text{-}path\ (hd\ ps)\ (tl\ ps)$
$\wedge\ adjacent\ (last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps)=hd\ ps\} + card\ \{ps.\ length\ ps = l+2 \wedge$
$adj\text{-}path\ (hd\ ps)\ (tl\ ps) \wedge adjacent\ (last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps)\neq hd\ ps\}$
using *card-Un-disjoint*[*of* $\{ps.\ length\ ps = l + 2 \wedge adj\text{-}path\ (hd\ ps)\ (tl\ ps) \wedge adjacent$
$(last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps) = hd\ ps\}\ \{ps.\ length\ ps = l + 2 \wedge adj\text{-}path\ (hd\ ps)$
$(tl\ ps) \wedge adjacent\ (last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps) \neq hd\ ps\}$] *finite-Un*
by *auto*
moreover have $card\ \{ps.\ length\ ps = l+2 \wedge adj\text{-}path\ (hd\ ps)\ (tl\ ps)$
$\wedge\ adjacent\ (last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps)=hd\ ps\}=k * card\ (C\text{-}star\ l)$
proof −
obtain *ext* **where** *ext*: $ext=(\lambda ps\ ps'.\ ps'\neq[] \wedge (butlast\ ps'=ps)$
$\wedge\ adj\text{-}path\ (hd\ ps')\ (tl\ ps'))$ **by** *auto*
have $\forall ps\in(C\text{-}star\ l).\ card\ \{ps'.\ ext\ ps\ ps'\} = k$
proof
fix *ps* **assume** $ps\in C\text{-}star\ l$
hence $length\ ps = l + 1$ **and** $adj\text{-}path\ (hd\ ps)\ (tl\ ps)$ **and** $last\ ps = hd\ ps$
using *C-star* **by** *auto*
obtain *qs* **where** *qs*:$qs=\{v.\ adjacent\ (last\ ps)\ v\}$ **by** *auto*
obtain *app* **where** *app*:$app=(\lambda v.\ ps@[v])$ **by** *auto*
have $app\ `\ qs = \{ps'.\ ext\ ps\ ps'\}$
proof −
have $\bigwedge x.\ x\in app`qs \Longrightarrow x\in\{ps'.\ ext\ ps\ ps'\}$
proof
fix *x* **assume** $x \in app\ `\ qs$
then obtain *y* **where** $adjacent\ (last\ ps)\ y\ x=ps@[y]$ **using** *qs app* **by** *auto*
moreover hence $adj\text{-}path\ (hd\ x)\ (tl\ x)$
by (*cases* $tl\ ps = []$, *metis adj-path.simps*(*1*) *adj-path.simps*(*2*)
adjacent-V(*2*) *append-Nil list.sel*(*1*,*3*) *hd-append snoc-eq-iff-butlast*

tl-append2, *metis* ‹*adj-path* (*hd ps*) (*tl ps*)› *adj-path-app hd-append*
last-tl list.sel(*2*) *tl-append2*)
ultimately show *ext ps x* **using** *ext* **by** (*metis snoc-eq-iff-butlast*)
qed
moreover have $\bigwedge$*x*. *x*∈{*ps'*. *ext ps ps'*}⟹ *x*∈ *app*‘*qs*
proof −
fix *x* **assume** *x* ∈ {*ps'*. *ext ps ps'*}
hence *x*≠[] **and** *butlast x*=*ps* **and** *adj-path* (*hd x*) (*tl x*)
using *ext* **by** *auto*
have *adjacent* (*last ps*) (*last x*)
proof (*cases length ps*=*1*)
case *True*
hence *length x*=*2* **using** ‹*butlast x*=*ps*› **by** *auto*
then obtain *x1 t1* **where** *x*=*x1*#*t1* **and** *length t1*=*1*
using *Suc-length-conv*[*of 1 x*] **by** *auto*
then obtain *x2* **where** *t1*=[*x2*]
using *Suc-length-conv*[*of 0 t1*] **by** *auto*
have *x*=[*x1*,*x2*] **using** ‹*x*=*x1*#*t1*› ‹*t1*=[*x2*]› **by** *auto*
thus *adjacent* (*last ps*) (*last x*)
using ‹*adj-path* (*hd x*) (*tl x*)› ‹*butlast x*=*ps*› **by** *auto*
next
case *False*
hence *tl ps*≠[]
by (*metis* ‹*length ps* = *l* + *1*› *add-0-iff add-diff-cancel-left'*
length-0-conv length-tl add.commute)
moreover have *adj-path* (*hd x*) (*tl ps* @ [*last x*])
using ‹*adj-path* (*hd x*) (*tl x*)› ‹*butlast x*=*ps*› ‹*x* ≠ []›
by (*metis append-butlast-last-id calculation list.sel*(*2*) *tl-append2*)
ultimately have *adjacent* (*last* (*tl ps*)) (*last x*)
using *adj-path-app'*[*of hd x tl ps last x*]
by *auto*
thus *adjacent* (*last ps*) (*last x*) **by** (*metis* ‹*tl ps* ≠ []› *last-tl*)
qed
thus *x* ∈ *app* ‘ *qs* **using** *app qs*
by (*metis* ‹*butlast x* = *ps*› ‹*x* ≠ []› *append-butlast-last-id mem-Collect-eq*
rev-image-eqI)
qed
ultimately show *?thesis* **by** *auto*
qed
moreover have *inj-on app qs* **using** *app* **unfolding** *inj-on-def* **by** *auto*
moreover have *last ps*∈*V*
using ‹*length ps* = *l* + *1*› ‹*adj-path* (*hd ps*) (*tl ps*)› *adj-path-V*
by (*metis* ‹*last ps* = *hd ps*› *adj-path.simps*(*1*) *last-in-set last-tl subset-code*(*1*))
hence *card qs*=*k* **using** *qs k-adj* **by** *auto*

ultimately show *card* {*ps'*. *ext ps ps'*} = *k* **by** (*metis card-image*)
qed
moreover have *finite* (*C-star l*)
proof −
have *C-star l* ⊆ *T l* **using** *C-star T* **by** *auto*
moreover have (*k* ∗ *k* − *k* + *1*) ∗ *k* ^ *l*≠*0* **using** ‹*k*≥*4*› **by** *auto*
hence *finite* (*T l*) **using** *T-count*[*of l*] **by** (*metis card-infinite*)
ultimately show *?thesis* **by** (*metis finite-subset*)
qed
moreover have ∀ *ps1 ps2*. *ps1* ≠ *ps2* ⟶ {*ps'*. *ext ps1 ps'*} ∩ {*ps'*. *ext ps2 ps'*} = {}
using *ext* **by** *auto*
moreover have (⋃*ps*∈(*C-star l*). {*ps'*. *ext ps ps'*}) = {*ps*. *length ps* = *l*+*2*
∧ *adj-path* (*hd ps*) (*tl ps*) ∧ *adjacent* (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)=*hd ps*}
proof −
have ⋀*x*. *x*∈(⋃*ps*∈(*C-star l*). {*ps'*. *ext ps ps'*}) ⟹ *x*∈{*ps*. *length ps* = *l*+*2*
∧ *adj-path* (*hd ps*) (*tl ps*) ∧ *adjacent* (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)=*hd ps*}
proof
fix *x* **assume** *x* ∈ (⋃*ps*∈*C-star l*. {*ps'*. *ext ps ps'*})
then obtain *ps* **where** *ps*∈*C-star l ext ps x* **by** *auto*
hence *length ps* = *l* + *1* **and** *adj-path* (*hd ps*) (*tl ps*) **and** *last ps* = *hd ps*
and *x* ≠ [] **and** *butlast x* = *ps adj-path* (*hd x*) (*tl x*)
using *C-star ext* **by** *auto*
have *length x* = *l* + *2*
using ‹ *butlast x* = *ps* › ‹ *length ps* = *l* + *1* › *length-butlast* **by** *auto*
moreover have *adj-path* (*hd x*) (*tl x*) **by** (*metis* ‹*adj-path* (*hd x*) (*tl x*)›)
moreover have *adjacent* (*last x*) (*hd x*)
proof −
have *length x*≥*2* **using** ‹*length x*=*l*+*2*› **by** *auto*
hence *adjacent* (*last* (*butlast x*)) (*last x*) **using** ‹*adj-path* (*hd x*) (*tl x*)›
by (*induct x*,*auto*, *metis adj-path*.*simps*(*2*) *append-butlast-last-id*
append-eq-Cons-conv, *metis adj-path-app' append-butlast-last-id*)
hence *adjacent* (*last ps*) (*last x*) **using** ‹*butlast x*=*ps*› **by** *auto*
hence *adjacent* (*hd ps*) (*last x*) **using** ‹*last ps*=*hd ps*› **by** *auto*
hence *adjacent* (*hd x*) (*last x*)
using ‹*butlast x*=*ps*› ‹*length ps*=*l*+*1*›
by (*cases x*) *auto*
thus *?thesis* **using** *adjacent-sym* **by** *auto*
qed
moreover have *last* (*butlast x*) = *hd x*

by (*metis* ‹*butlast x* = *ps*› ‹*last ps* = *hd ps*› ‹*x* ≠ []› *adjacent-no-loop*

butlast.simps(*2*) *calculation*(*3*) *list.sel*(*1*) *last-ConsL neq-Nil-conv*)
ultimately show *length x* = *l* + *2* ∧ *adj-path* (*hd x*) (*tl x*)
∧ *adjacent* (*last x*) (*hd x*) ∧ *last* (*butlast x*) = *hd x*
by *auto*
qed
moreover have ⋀*x*. *x*∈{*ps*. *length ps* = *l*+*2* ∧ *adj-path* (*hd ps*) (*tl ps*)
∧ *adjacent* (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)=*hd ps*} ⟹
x∈(⋃*ps*∈(*C-star l*). {*ps*′. *ext ps ps*′})
proof −
fix *x* **assume** *x*∈{*ps*. *length ps* = *l*+*2* ∧ *adj-path* (*hd ps*) (*tl ps*)
∧ *adjacent* (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)=*hd ps*}
hence *length x*=*l*+*2* **and** *adj-path* (*hd x*) (*tl x*) **and** *adjacent* (*last x*) (*hd x*)
and *last* (*butlast x*)=*hd x* **by** *auto*
obtain *ps* **where** *ps*:*ps*=*butlast x* **by** *auto*
have *ps*∈*C-star l*
proof −
have *length ps* = *l* + *1* **using** *ps* ‹*length x*=*l*+*2*› **by** *auto*
moreover have *hd ps*=*hd x*
using *ps* ‹*length x*=*l*+*2*›
by (*metis* (*full-types*) ‹ *adjacent* (*last x*) (*hd x*) › *adjacent-no-loop*

append-Nil append-butlast-last-id butlast.simps(*1*) *list.sel*(*1*) *hd-append2*)
hence *adj-path* (*hd ps*) (*tl ps*) **using** *adj-path-butlast*
by (*metis* ‹*adj-path* (*hd x*) (*tl x*)› *butlast-tl ps*)
moreover have *last ps* = *hd ps*
by (*metis* ‹*hd ps* = *hd x*› ‹*last* (*butlast x*) = *hd x*› *ps*)
ultimately show *?thesis* **using** *C-star* **by** *auto*
qed
moreover have *ext ps x* **using** *ext*
by (*metis* ‹*adj-path* (*hd x*) (*tl x*)› ‹*adjacent* (*last x*) (*hd x*)›
‹*last* (*butlast x*) = *hd x*› *adjacent-no-loop butlast.simps*(*1*) *ps*)
ultimately show *x*∈(⋃*ps*∈(*C-star l*). {*ps*′. *ext ps ps*′}) **by** *auto*
qed
ultimately show *?thesis* **by** *fast*
qed
ultimately show *?thesis* **using** *card-partition*′[*of C-star l ext k*] ‹*k*≥*4*› **by** *auto*
qed
moreover have *card* {*ps*. *length ps* = *l*+*2* ∧ *adj-path* (*hd ps*) (*tl ps*) ∧
adjacent (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)≠*hd ps*}=*card* (*T l* − *C-star l*)
proof −
obtain *app* **where** *app*:*app*=(λ*ps*. *ps*@[*SOME n*. *adjacent* (*last ps*) *n* ∧ *adjacent* (*hd ps*) *n*])

by *auto*

have $\bigwedge x.\ x{\in}app`(T\ l\ -\ C\text{-}star\ l) \Longrightarrow x{\in}\{ps.\ length\ ps = l{+}2 \wedge adj\text{-}path\ (hd\ ps)\ (tl\ ps) \wedge$
$adjacent\ (last\ ps)\ (hd\ ps) \wedge last\ (butlast\ ps){\neq}hd\ ps\}$

proof

fix x **assume** $x \in app\ `\ (T\ l\ -\ C\text{-}star\ l)$

then obtain *ps* **where** $length\ ps = l + 1$ $adj\text{-}path\ (hd\ ps)\ (tl\ ps)$ $last\ ps \neq hd\ ps$
$x{=}app\ ps$

using *T C-star* **by** *auto*

hence $last\ ps{\in}V$

using *adj-path-V*[*OF* ‹*adj-path* (*hd ps*) (*tl ps*)›]

by (*cases ps*) *auto*

hence $\exists\, n.\ adjacent\ (last\ ps)\ n \wedge adjacent\ (hd\ ps)\ n$

using *adj-path-V'*[*OF* ‹*adj-path* (*hd ps*) (*tl ps*)›] ‹$last\ ps{\neq}hd\ ps$›
friend-assm[*of last ps hd ps*]

by *auto*

moreover have $last\ x{=}(SOME\ n.\ adjacent\ (last\ ps)\ n \wedge adjacent\ (hd\ ps)\ n)$

using *app* ‹$x{=}app\ ps$› **by** *auto*

ultimately have *adjacent* (*last ps*) (*last x*) **and** *adjacent* (*hd ps*) (*last x*)

using *someI-ex* **by** (*metis* (*lifting*))+

have $hd\ x{=}hd\ ps$ **using** ‹$x{=}app\ ps$› ‹$length\ ps{=}l{+}1$› *app*

by (*cases ps*) *auto*

have $length\ x = l + 2$ **using** ‹$x{=}app\ ps$› ‹$length\ ps{=}l{+}1$› *app* **by** *auto*

moreover have *adj-path* (*hd x*) (*tl x*)

proof −

have $last\ (tl\ ps){=}last\ ps$ **using** ‹$length\ ps{=}l{+}1$›
by (*metis* ‹$last\ ps \neq hd\ ps$› *list.sel*(*1*,*3*) *last-ConsL last-tl neq-Nil-conv*)

moreover have $length\ ps{\neq}1$ **using** ‹$last\ ps \neq hd\ ps$›
by (*metis Suc-eq-plus1-left gen-length-code*(*1*) *gen-length-def list.sel*(*1*)
last-ConsL length-Suc-conv neq-Nil-conv)

hence $tl\ ps{\neq}[]$ **using** ‹$length\ ps{=}l{+}1$›

by(*auto simp*: *length-Suc-conv*)

ultimately have *adj-path* (*hd ps*) (*tl ps* @ [*last x*])

using *adj-path-app*[*OF* ‹*adj-path* (*hd ps*) (*tl ps*)›,*of last x*]
‹*adjacent* (*last ps*) (*last x*)›

by *auto*

moreover have *tl ps* @ [*last x*]=*tl x*

using ‹$x{=}app\ ps$› *app*

by (*metis* ‹ $last\ x = (SOME\ n.\ adjacent\ (last\ ps)\ n \wedge adjacent\ (hd\ ps)\ n)$ ›
‹ $tl\ ps \neq []$ › *list.sel*(*2*) *tl-append2*)

ultimately show *?thesis* **using** ‹$hd\ x{=}hd\ ps$› **by** *auto*

qed

moreover have *adjacent* (*last x*) (*hd x*)

using ‹*hd x=hd ps*› ‹*adjacent* (*hd ps*) (*last x*)› *adjacent-sym* **by** *auto*
moreover have *last* (*butlast x*) $\neq$ *hd x*
using ‹*last ps* $\neq$ *hd ps*› ‹*hd x=hd ps*›
by (*metis* ‹*x* = *app ps*› *app butlast-snoc*)
ultimately show *length x* = *l* + *2* $\wedge$ *adj-path* (*hd x*) (*tl x*) $\wedge$ *adjacent* (*last x*) (*hd x*)
$\wedge$ *last* (*butlast x*) $\neq$ *hd x*
by *auto*
qed
moreover have $\bigwedge$*x*. *x*$\in${*ps*. *length ps* = *l*+*2* $\wedge$ *adj-path* (*hd ps*) (*tl ps*) $\wedge$
adjacent (*last ps*) (*hd ps*) $\wedge$ *last* (*butlast ps*)$\neq$*hd ps*}$\Longrightarrow$ *x*$\in$*app*‘(*T l* − *C-star l*)
proof −
fix *x* **assume** *x*$\in${*ps*. *length ps* = *l*+*2* $\wedge$ *adj-path* (*hd ps*) (*tl ps*) $\wedge$
adjacent (*last ps*) (*hd ps*) $\wedge$ *last* (*butlast ps*)$\neq$*hd ps*}
hence *length x*=*l*+*2* **and** *adj-path* (*hd x*) (*tl x*) **and** *adjacent* (*last x*) (*hd x*)
and *last* (*butlast x*)$\neq$*hd x*
by *auto*
hence *butlast x*$\in$*T l* − *C-star l*
proof −
have *length* (*butlast x*) = *l* + *1*
using ‹*length x* = *l* + *2*› *length-butlast* **by** *auto*
moreover have *hd* (*butlast x*)=*hd x*
using ‹*length x*=*l*+*2*›
by (*metis append-butlast-last-id butlast.simps*(*1*) *calculation diff-add-inverse*
diff-cancel2 hd-append length-butlast add.commute num.distinct(*1*)

one-eq-numeral-iff)
hence *adj-path* (*hd* (*butlast x*)) (*tl* (*butlast x*))
using ‹*adj-path* (*hd x*) (*tl x*)› **by** (*metis adj-path-butlast butlast-tl*)
moreover have *last* (*butlast x*) $\neq$ *hd* (*butlast x*)
using ‹*last* (*butlast x*)$\neq$*hd x*› ‹*hd* (*butlast x*)=*hd x*› **by** *auto*
ultimately show *?thesis* **using** *T C-star* **by** *auto*
qed
moreover have *app* (*butlast x*)=*x* **using** *app*
proof −
have *last* (*butlast x*)$\in$*V*
proof (*cases length x*$\geq$*3*)
case *True*
hence *last* (*butlast x*)$\in$*set* (*tl x*)
proof (*induct x*)
case *Nil*
thus *?case* **by** *auto*
next
case (*Cons x1 t1*)
have *length t1*<*3* $\Longrightarrow$*?case*
proof −

```
          assume length t1<3
         hence length t1=2 using ‹3 ≤ length (x1 # t1)› by auto
          then obtain x2 t2 where t1=x2#t2 length t2=1
            using Suc-length-conv[of 1 t1] by auto
          then obtain x3 where t2=[x3]
            using  Suc-length-conv[of 0 t2] by auto
          have t1=[x2,x3] using ‹t1=x2#t2› ‹t2=[x3]› by auto
          thus ?case by auto
        qed
      moreover have length t1≥3⟹?case
        proof –
          assume length t1≥3
          hence last (butlast t1) ∈ set (tl t1)
            using Cons.hyps by auto
          thus ?case
               by (metis butlast.simps(2) in-set-butlastD last.simps
last-in-set
             length-butlast length-greater-0-conv length-pos-if-in-set
             length-tl list.sel(3))
        qed
      ultimately show ?case by force
    qed
   thus ?thesis using adj-path-V[OF ‹adj-path (hd x) (tl x)›] by
auto
  next
    case False
    hence length x=2 using ‹length x=l+2›  by auto
    then obtain x1 x2 where x=[x1,x2]
      proof –
        obtain x1 t1 where x=x1#t1 length t1=1
          using Suc-length-conv[of 1 x] ‹length x=2› by auto
        then obtain x2 where t1=[x2]
          using  Suc-length-conv[of 0 t1] by auto
        have x=[x1,x2] using ‹x=x1#t1› ‹t1=[x2]› by auto
        thus ?thesis using that by auto
      qed
    hence last (butlast x)=hd x by auto
    thus ?thesis using adj-path-V'[OF ‹adj-path (hd x) (tl x)›] by
auto
  qed
moreover have hd (butlast x)=hd x using ‹length x=l+2›
  by (metis ‹adjacent (last x) (hd x)› adjacent-no-loop append-butlast-last-id

    butlast.simps(1) list.sel(1) hd-append)
hence  hd (butlast x)∈V using adj-path-V'[OF ‹adj-path (hd x) (tl
x)›] by auto
moreover have last (butlast x)≠hd (butlast x)
  using ‹last (butlast x)≠hd x› ‹hd (butlast x)=hd x› by auto
ultimately have ∃! n. adjacent (last (butlast x)) n ∧ adjacent (hd
```

(*butlast x*)) *n*
using *friend-assm* **by** *auto*
moreover have *length* $x \geq 2$ **using** ‹*length* $x=l+2$› **by** *auto*
hence *adjacent* (*last* (*butlast x*)) (*last x*)
using ‹*adj-path* (*hd x*) (*tl x*)›
by (*induct x*,*auto*, *metis* (*full-types*) *adj-path.simps*(*2*) *append-Nil*
append-butlast-last-id, *metis adj-path-app′ append-butlast-last-id*)
moreover have *adjacent* (*hd* (*butlast x*)) (*last x*)
using ‹*adjacent* (*last x*) (*hd x*)› ‹*hd* (*butlast x*)=*hd x*› *adjacent-sym*
by *auto*
ultimately have (*SOME n*. *adjacent* (*last* (*butlast x*)) *n*
∧ *adjacent* (*hd* (*butlast x*)) *n*) = *last x*
using *some1-equality* **by** *fast*
moreover have *x*=(*butlast x*)@[*last x*]
by (*metis* ‹*adjacent* (*last* (*butlast x*)) (*last x*)› *adjacent-no-loop*
append-butlast-last-id butlast.simps(*1*))
ultimately show *?thesis* **using** *app* **by** *auto*
qed
ultimately show *x*∈*app*‘(*T l* − *C-star l*) **by** (*metis image-iff*)
qed
ultimately have *app*‘(*T l* − *C-star l*)={*ps*. *length ps* = *l*+*2* ∧ *adj-path*
(*hd ps*) (*tl ps*) ∧
adjacent (*last ps*) (*hd ps*) ∧ *last* (*butlast ps*)≠*hd ps*} **by** *fast*
moreover have *inj-on app* (*T l* − *C-star l*) **using** *app* **unfolding** *inj-on-def*
by *auto*
ultimately show *?thesis* **by** (*metis card-image*)
qed
ultimately show *card* (*C* (*l* + *1*)) = *k* ∗ *card* (*C-star l*) + *card* (*T l* −
C-star l) **by** *auto*
qed
hence ⋀*l*::*nat*. *card* (*C* (*l*+*1*)) *mod* (*k*−(*1*::*nat*))=*1*
proof −
fix *l*::*nat*
have *C-star l* ⊆ *T l* **using** *C-star T* **by** *auto*
moreover have *card* (*T l*)≠*0* **using** *T-count* ‹*k*≥*4*› **by** *auto*
hence *finite* (*T l*) **using** ‹*k*≥*4*› **by** (*metis card-infinite*)
ultimately have *card* (*T l* − *C-star l*)=*card*(*T l*) − *card*(*C-star l*)
by (*metis card-Diff-subset rev-finite-subset*)
hence *card* (*C* (*l* + *1*))=*k*∗*card* (*C-star l*) + (*card* (*T l*) − *card* (*C-star l*))
using ‹⋀*l*::*nat*. *card* (*C* (*l*+*1*)) = *k*∗ *card* (*C-star l*) + *card* (*T l* − *C-star*
l)›
by *auto*
also have ...=*k*∗*card* (*C-star l*) + *card* (*T l*) − *card* (*C-star l*)
proof −
have *card* (*T l*) ≥ *card* (*C-star l*)
using ‹*C-star l* ⊆ *T l*› ‹*finite* (*T l*)› **by** (*metis card-mono*)
thus *?thesis* **by** *auto*
qed
also have ...=*k*∗*card* (*C-star l*) − *card* (*C-star l*) + *card* (*T l*)

proof −
have *card* (*T l*) ≥ *card* (*C-star l*)
using ‹*C-star l* ⊆ *T l*› ‹*finite* (*T l*)› **by** (*metis card-mono*)
moreover have *k*∗*card* (*C-star l*) ≥ *card* (*C-star l*) **using** ‹*k*≥*4*› **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
also have ...=(*k*−(*1*::*nat*))∗*card*(*C-star l*)+*card*(*T l*) **using** ‹*k*≥*4*›
by (*metis monoid-mult-class.mult.left-neutral diff-mult-distrib*)
finally have *card* (*C* (*l* + *1*))=(*k*−(*1*::*nat*))∗*card*(*C-star l*)+*card*(*T l*) **.**
hence *card* (*C* (*l*+*1*)) *mod* (*k*−(*1*::*nat*)) = *card*(*T l*) *mod* (*k*−(*1*::*nat*)) **using**
‹*k*>=*4*›
by (*metis mod-mult-self3 mult.commute*)
also have ...=((*k*∗*k*−*k*+*1*)∗*k*ˆ*l*) *mod* (*k*−(*1*::*nat*)) **using** *T-count* **by** *auto*
also have ...=((*k*−(*1*::*nat*))∗*k*+*1*)∗*k*ˆ*l* *mod* (*k*−(*1*::*nat*))
proof −
have *k*∗*k*−*k*+*1*=(*k*−(*1*::*nat*))∗*k*+*1* **using** ‹*k*≥*4*› **by** (*metis diff-mult-distrib*
nat-mult-1)
thus *?thesis* **by** *auto*
qed
also have ...=*1*∗*k*ˆ*l* *mod* (*k*−(*1*::*nat*))
by (*metis mod-mult-right-eq mod-mult-self1 add.commute mult.commute*)
also have ...=*k*ˆ*l* *mod* (*k*−(*1*::*nat*)) **by** *auto*
also have ...=(*k*−(*1*::*nat*)+*1*)ˆ*l* *mod* (*k*−(*1*::*nat*)) **using** ‹*k*≥*4*› **by** *auto*
also have ...=*1*ˆ*l* *mod* (*k*−(*1*::*nat*)) **by** (*metis mod-add-self2 add.commute*
power-mod)
also have ...=*1* *mod* (*k*−(*1*::*nat*)) **by** *auto*
also have ...=*1* **using** ‹*k*≥*4*› **by** *auto*
finally show *card* (*C* (*l*+*1*)) *mod* (*k*−(*1*::*nat*)) =*1* **.**
qed
obtain *p*::*nat* **where** *prime p p dvd* (*k*−(*1*::*nat*)) **using** ‹*k*≥*4*›
by (*metis Suc-eq-plus1 Suc-numeral add-One-commute eq-iff le-diff-conv numeral-le-iff*

one-le-numeral one-plus-BitM prime-factor-nat semiring-norm(*69*) *semiring-norm*(*71*))
hence *p-minus-1*:*p*−(*1*::*nat*)+*1*=*p*
by (*metis add-diff-inverse add.commute not-less-iff-gr-or-eq prime-nat-iff*)
hence ∗: ⋀*l*::*nat*. *card* (*C* (*l*+*1*)) *mod p*=*1*
using ‹⋀*l*::*nat*. *card* (*C* (*l*+*1*)) *mod* (*k*−(*1*::*nat*))=*1*› *mod-mod-cancel*[*OF* ‹*p*
dvd (*k*−(*1*::*nat*))›]
‹*prime p*›
by (*metis mod-if prime-gt-1-nat*)
have *card* (*C* (*p* − *1*)) *mod p* = *1*
proof (*cases 2* ≤ *p*)
case *True* **with** ∗ [*of p* − *2*] **show** *?thesis*
by (*metis Nat.add-diff-assoc2 add-le-cancel-right diff-diff-left one-add-one*
p-minus-1)
next
case *False* **with** ∗ [*of p* − *2*] ‹*prime p*› *prime-ge-2-nat* **show** *?thesis*
by *blast*
qed

moreover have *card* (*C* (*p*−(*1*::*nat*))) *mod p*=*0* **using** *C*
proof −
have *closure1*:⋀*x*. *x*∈*C* (*p*−(*1*::*nat*))⟹ *rotate1 x* ∈*C* (*p*−(*1*::*nat*))
proof −
fix *x* **assume** *x*∈*C* (*p*−(*1*::*nat*))
hence *length x* = *p* **and** *adj-path* (*hd x*) (*tl x*) **and** *adjacent* (*last x*) (*hd x*)
using *C p-minus-1* **by** *auto*
have *adjacent* (*last* (*rotate1 x*)) (*hd* (*rotate1 x*))
proof −
have *x*≠[] **using** ‹*length x*=*p*› ‹*prime p*› **by** *auto*
hence *adjacent* (*last* (*rotate1 x*)) (*hd* (*rotate1 x*))=*adjacent* (*hd x*) (*hd* (*tl x*))
by (*metis* ‹ *adjacent* (*last x*) (*hd x*) › *adjacent-no-loop append-Nil list.sel*(*1*,*3*)
hd-append2 last-snoc list.exhaust rotate1-hd-tl)
also have ...=*True* **using** ‹*adj-path* (*hd x*) (*tl x*)›
using ‹*adjacent* (*last x*) (*hd x*)› ‹*x* ≠ []›
by (*metis adj-path.simps*(*2*) *adjacent-no-loop append1-eq-conv append-Nil*
append-butlast-last-id list.sel(*1*,*3*) *list.exhaust*)
finally show *?thesis* **by** *auto*
qed
moreover have *adj-path* (*hd* (*rotate1 x*)) (*tl* (*rotate1 x*))
proof −
have *x*≠[] **using** ‹*length x*=*p*› ‹*prime p*› **by** *auto*
then obtain *y ys* **where** *y*=*hd x ys*=*tl x* **by** *auto*
hence *adj-path y ys* **and** *adjacent* (*last ys*) *y* **and** *ys*≠[]
by (*metis* ‹*adj-path* (*hd x*) (*tl x*)›, *metis* ‹*adjacent* (*last x*) (*hd x*)› ‹*y* = *hd x*›
‹*ys* = *tl x*› *adjacent-no-loop list.sel*(*1*,*3*) *last.simps last-tl list.exhaust*
, *metis* ‹*adjacent* (*last x*) (*hd x*)› ‹*x* ≠ []› ‹*ys* = *tl x*› *adjacent-no-loop list.sel*(*1*,*3*)
last-ConsL neq-Nil-conv)
hence *adj-path* (*hd* (*rotate1 x*)) (*tl* (*rotate1 x*))
=*adj-path* (*hd* (*ys*@[*y*])) (*tl* (*ys*@[*y*]))
using ‹*x*≠[]› ‹*y*=*hd x*› ‹*ys*=*tl x*› **by** (*metis rotate1-hd-tl*)
also have ...=*adj-path* (*hd ys*) ((*tl ys*)@[*y*])
by (*metis* ‹*ys* ≠ []› *hd-append tl-append2*)
also have ...=*True*
using *adj-path-app*[*OF* ‹*adj-path y ys*› ‹*ys*≠[]› ‹*adjacent* (*last ys*) *y*›] ‹*ys*≠[]›
by (*metis adj-path.simps*(*2*) *append-Cons list.sel*(*1*,*3*) *list.exhaust*)
finally show *?thesis* **by** *auto*
qed
moreover have *length* (*rotate1 x*) = *p* **using** ‹*length x*=*p*› **by** *auto*
ultimately show *rotate1 x* ∈ *C* (*p*−(*1*::*nat*)) **using** *C p-minus-1* **by** *auto*
qed
have *closure*:⋀*n x*. *x*∈*C* (*p*−(*1*::*nat*))⟹ *rotate n x* ∈*C* (*p*−(*1*::*nat*))

proof −
fix *n x* **assume** $x \in C\ (p-(1::nat))$
thus $rotate\ n\ x \in C\ (p-(1::nat))$
by (*induct n*,*auto*,*metis One-nat-def closure1*)
qed
obtain *r* **where** $r{:}r=\{(x,y).\ x \in C\ (p-(1::nat)) \wedge (\exists n<p.\ rotate\ n\ x=y)\}$
by *auto*
have $\bigwedge x.\ x \in C\ (p-(1::nat)) \Longrightarrow p\ dvd\ card\ \{y.(\exists n<p.\ rotate\ n\ x=y)\}$
proof −
fix *x* **assume** $x \in C\ (p-(1::nat))$
hence *length x=p* **using** *C p-minus-1* **by** *auto*
have $\{y.\ (\exists n<p.\ rotate\ n\ x=y)\}= (\lambda n.\ rotate\ n\ x)`\ \{0..<p\}$ **by** *auto*
moreover have $\bigwedge n1\ n2.\ n1 \in \{0..<p\} \Longrightarrow n2 \in \{0..<p\} \Longrightarrow n1 \neq n2 \Longrightarrow rotate\ n1\ x \neq rotate\ n2\ x$
proof
fix *n1 n2* **assume** $n1 \in \{0..<p\}\ n2 \in \{0..<p\}\ n1 \neq n2\ rotate\ n1\ x = rotate\ n2\ x$
{ fix *n1 n2*
assume $n1 \in \{0..<p\}\ n2 \in \{0..<p\}\ rotate\ n1\ x = rotate\ n2\ x\ n1>n2$
obtain *s::nat* **where** $s*(n1-n2)\ mod\ p=1\ s>0$
proof −
have $n1-n2>0$ **and** $n1-n2<p$
using ‹$n1 \in \{0..<p\}$› ‹$n2 \in \{0..<p\}$› ‹$n1>n2$› **by** *auto*
with ‹*prime p*› **have** *coprime* $(n1 - n2)\ p$
by (*simp add*: *prime-nat-iff''* *coprime-commute* [*of p*])
then have $\exists x.\ [(n1 - n2) * x = 1]\ (mod\ p)$
by (*simp add*: *cong-solve-coprime-nat*)
then obtain *s* **where** $s * (n1 - n2)\ mod\ p = 1$
using ‹*prime p*› *prime-gt-1-nat* [*of p*]
by (*auto simp add*: *cong-def ac-simps*)
moreover hence $s>0$ **by** (*metis mod-0 mult-0 neq0-conv zero-neq-one*)
ultimately show *?thesis* **using** *that* **by** *auto*
qed
have $rotate\ (s*n1)\ x=rotate\ (s*n2)\ x$
using ‹*rotate n1 x=rotate n2 x*›
apply (*induct s*)
apply (*auto simp add*: *algebra-simps*)
by (*metis add.commute rotate-rotate*)
hence $rotate\ (s*n1 - s*n2)\ x= x$
using *rotate-diff* **by** *auto*
hence $rotate\ (s*(n1-n2))\ x=x$ **by** (*metis diff-mult-distrib mult.commute*)
hence $rotate\ 1\ x = x$ **using** ‹$s*(n1-n2)\ mod\ p=1$› ‹*length x=p*›
by (*metis rotate-conv-mod*)
hence *rotate1 x=x* **by** *auto*
have *hd x=hd (tl x)* **using** ‹*prime p*› ‹*length x=p*›
proof −
have *length x≥2* **using** ‹*prime p*› ‹*length x=p*› **using** *prime-ge-2-nat*

```
by blast
                hence length (tl x)≥1 by force
                hence x≠[] and tl x≠[] by auto+
               hence x=(hd x)#(hd (tl x))#(tl (tl x)) using hd-Cons-tl by auto
              hence (hd (tl x))#(tl (tl x))@[hd x]=(hd x)#(hd (tl x))#(tl (tl x))
              using ‹rotate1 x = x› by (metis Cons-eq-appendI rotate1.simps(2))
                thus ?thesis by auto
              qed
            moreover have hd x≠hd (tl x)
              proof −
               have adj-path (hd x) (tl x) using ‹x ∈ C (p−(1::nat))› C by auto
                moreover have length x≥2 using ‹prime p› ‹length x=p› using
prime-ge-2-nat by blast
                hence length (tl x)≥1 by force
                hence tl x≠[] by force
                ultimately have adjacent (hd x) (hd (tl x))
                  by (metis adj-path.simps(2) list.sel(1) list.exhaust)
                thus ?thesis by (metis adjacent-no-loop)
              qed
            ultimately have False by auto }
          thus False
            by (metis ‹n1 ∈ {0..<p}› ‹n1 ≠ n2› ‹n2 ∈ {0..<p}› ‹rotate n1 x =
rotate n2 x›
              less-linear)
        qed
      hence inj-on (λn. rotate n x) {0..<p} unfolding inj-on-def by fast
        ultimately have card {y. (∃n<p. rotate n x=y)}=card {0..<p} by
(metis card-image)
      hence card {y. (∃n<p. rotate n x=y)}=p by auto
      thus p dvd card {y. (∃n<p. rotate n x=y)} by auto
    qed
    hence ∀X∈C (p−(1::nat)) // r. p dvd card X unfolding quotient-def
Image-def r by auto
   moreover have refl-on (C (p − 1)) r
    proof −
      have r ⊆ C (p − 1) × C (p − 1)
        proof
          fix x assume x∈r
           hence fst x∈C (p − 1) and ∃n. snd x=rotate n (fst x) using r by
auto
          moreover then obtain n where snd x=rotate n (fst x) by auto
          ultimately have snd x∈C (p − 1) using closure by auto
          moreover have x=(fst x,snd x) using ‹x∈r› r by auto
          ultimately show  x ∈ C (p − 1) × C (p − 1) using ‹fst x∈ C (p −
1)›
            by (metis SigmaI)
        qed
      moreover have ∀x∈C (p − 1). (x, x) ∈ r
        proof
```

```
        fix x assume x ∈ C (p − 1)
        hence rotate 0 x ∈ C (p − 1) using closure by auto
        moreover have 0<p using ‹prime p› by (auto intro: prime-gt-0-nat)
        ultimately have (x,rotate 0 x)∈ r using ‹x∈C (p − 1 )› r by auto
        moreover have rotate 0 x=x by auto
        ultimately show (x,x)∈r by auto
      qed
    ultimately show ?thesis using refl-on-def by auto
  qed
moreover have sym r unfolding sym-def
  proof (rule,rule,rule)
    fix x y assume (x, y) ∈ r
    hence x∈C (p − 1) using r by auto
    hence length x=p using C p-minus-1 by auto
    obtain n where n<p rotate n x = y using ‹(x,y)∈r› r by auto
    hence y∈ C (p − 1) using closure[OF ‹x∈ C (p − 1)›] by auto
    have n=0⟹(y, x) ∈ r
      proof −
        assume n=0
        hence x=y using ‹rotate n x=y› by auto
       thus (y,x)∈r using ‹refl-on (C (p − 1)) r› ‹y ∈ C (p − 1)› refl-on-def
by fast
      qed
    moreover have n≠0 ⟹ (y,x)∈r
      proof −
        assume n≠0
        have rotate (p−n) y = x
          proof −
            have rotate (p−n) y = rotate (p−n) (rotate n x)
              using ‹rotate n x=y› by auto
            also have rotate (p−n) (rotate n x)=rotate (p−n+n) x
              using rotate-rotate by auto
            also have ...=rotate p x using ‹n<p› by auto
            also have ...=rotate 0 x using ‹length x=p› by auto
            also have ...=x by auto
            finally show ?thesis .
          qed
        moreover have p−n<p using ‹n<p› ‹n≠0› by auto
        ultimately show (y,x)∈r using r ‹y∈ C (p − 1)› by auto
      qed
    ultimately show (y,x)∈r by auto
  qed
moreover have trans r unfolding trans-def
  proof (rule,rule,rule,rule,rule)
    fix x y z assume (x, y) ∈ r (y, z) ∈ r
    hence x∈C (p − 1) using r by auto
    hence length x=p using C p-minus-1 by auto
    obtain n1 n2 where n1<p n2<p y=rotate n1 x z=rotate n2 y
      using r ‹(x,y)∈r› ‹(y,z)∈r› by auto
```

```
        hence z=rotate (n2+n1) x by (metis rotate-rotate)
          hence z=rotate ((n2+n1) mod p) x using ‹length x=p› by (metis
rotate-conv-mod)
        moreover have (n2+n1) mod p < p by (metis ‹prime p› mod-less-divisor
prime-gt-0-nat)
        ultimately show (x,z)∈r using ‹x∈ C (p − 1)› r by auto
      qed
    moreover have finite (C (p − 1))
      by (metis ‹card (C (p − 1)) mod p = 1› card-eq-0-iff mod-0 zero-neq-one)
      ultimately have p dvd card (C (p−(1::nat))) using equiv-imp-dvd-card
equiv-def by fast
    thus card (C (p−(1::nat))) mod p=0 by (metis dvd-eq-mod-eq-0)
  qed
  ultimately show False by auto
qed

theorem (in valid-unSimpGraph) friendship-thm:
  assumes friend-assm:⋀v u. v∈V ⟹ u∈V ⟹ v≠u ⟹ ∃! n. adjacent v n ∧
adjacent u n
    and finite V
  shows ∃ v. ∀ n∈V. n≠v ⟶ adjacent v n
proof −
  have card V=0 ⟹ ?thesis
    using ‹finite V›
    by (metis all-not-in-conv card-seteq empty-subsetI le0)
  moreover have card V=1 ⟹ ?thesis
    proof −
      assume card V=1
      then obtain v where V={v}
        using card-eq-SucD[of V 0] by auto
      hence ∀ n∈V. n=v by auto
      thus ∃ v. ∀ n∈V. n ≠ v ⟶ adjacent v n by auto
    qed
  moreover have card V≥2 ⟹ ?thesis
    proof −
      assume card V≥2
      hence ∃ v∈V. degree v G = 2
        using exist-degree-two[OF friend-assm] ‹finite V› by auto
      thus ?thesis
        using degree-two-windmill[OF friend-assm] ‹card V≥2› ‹finite V› by auto
    qed
  ultimately show ?thesis by force
qed

end
```

References

[1] J. Q. Longyear and T. Parsons. The friendship theorem. *Indagationes Mathematicae (Proceedings)*, 75(3):257 – 262, 1972.

[2] F. Martin. The Seven Bridges of Königsberg. http://www.ugr.es/~fmartin/gi/bridges.pdf.

[3] G. B. Mertzios and W. Unger. The friendship problem on graphs. 2008.

[4] B. Nordhoff and P. Lammich. Dijkstra's shortest path algorithm. *Archive of Formal Proofs*, June 2012. http://isa-afp.org/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.