

The Floyd-Warshall Algorithm for Shortest Paths

Simon Wimmer and Peter Lammich

March 14, 2025

Abstract

The Floyd-Warshall algorithm [Flo62, Roy59, War62] is a classic dynamic programming algorithm to compute the length of all shortest paths between any two vertices in a graph (i.e. to solve the all-pairs shortest path problem, or *APSP* for short). Given a representation of the graph as a matrix of weights M , it computes another matrix M' which represents a graph with the same path lengths and contains the length of the shortest path between any two vertices i and j . This is only possible if the graph does not contain any negative cycles. However, in this case the Floyd-Warshall algorithm will detect the situation by calculating a negative diagonal entry. This entry includes a formalization of the algorithm and of these key properties. The algorithm is refined to an efficient imperative version using the Imperative Refinement Framework.

Contents

1	Floyd-Warshall Algorithm for the All-Pairs Shortest Paths Problem	2
1.1	Introduction	2
1.2	Preliminaries	3
1.3	Definition of the Algorithm	11
1.4	Result Under The Absence of Negative Cycles	21
1.5	Definition of Shortest Paths	28
1.6	Intermezzo: Equivalent Characterizations of Cycle-Freeness	34
1.7	Result Under the Presence of Negative Cycles	42
1.8	More on Canonical Matrices	48
1.9	Additional Theorems	48
1.10	Refinement to Efficient Imperative Code	54

```
theory Floyd-Warshall
  imports Main
begin
```

1 Floyd-Warshall Algorithm for the All-Pairs Shortest Paths Problem

1.1 Introduction

The Floyd-Warshall algorithm [Flo62, Roy59, War62] is a classic dynamic programming algorithm to compute the length of all shortest paths between any two vertices in a graph (i.e. to solve the all-pairs shortest path problem, or *APSP* for short). Given a representation of the graph as a matrix of weights M , it computes another matrix M' which represents a graph with the same path lengths and contains the length of the shortest path between any two vertices i and j . This is only possible if the graph does not contain any negative cycles (then the length of the shortest path is $-\infty$). However, in this case the Floyd-Warshall algorithm will detect the situation by calculating a negative diagonal entry corresponding to the negative cycle. In the following, we present a formalization of the algorithm and of the aforementioned key properties.

Abstractly, the algorithm corresponds to the following imperative pseudo-code:

```
for k = 1 .. n do
  for i = 1 .. n do
    for j = 1 .. n do
      m[i, j] := min(m[i, j], m[i, k] + m[k, j])
```

However, we will carry out the whole formalization on a recursive version of the algorithm, and refine it to an efficient imperative version corresponding to the above pseudo-code in the end. The main observation underlying the algorithm is that the shortest path from i to j which only uses intermediate vertices from the set $\{0..k+1\}$, is: either the shortest path from i to j using intermediate vertices from the set $\{0..k\}$; or a combination of the shortest path from i to k and the shortest path from k to j , each of them only using intermediate vertices from $\{0..k\}$. Our presentation we be slightly more general than the typical textbook version, in that we will factor out the inner two loops as a separate algorithm and show that it has similar properties as the full algorithm for a single intermediate vertex k .

1.2 Preliminaries

1.2.1 Cycles in Lists

abbreviation $\text{cnt } x \text{ } xs \equiv \text{length } (\text{filter } (\lambda y. x = y) \text{ } xs)$

fun $\text{remove-cycles} :: 'a \text{ list} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$

where

$\text{remove-cycles } [] \text{ } - \text{acc} = \text{rev } \text{acc} \mid$

$\text{remove-cycles } (x\#xs) \text{ } y \text{ } \text{acc} =$

$(\text{if } x = y \text{ then } \text{remove-cycles } xs \text{ } y \text{ } [x] \text{ else } \text{remove-cycles } xs \text{ } y \text{ } (x\#\text{acc}))$

lemma $\text{cnt-rev}: \text{cnt } x \text{ } (\text{rev } xs) = \text{cnt } x \text{ } xs$ **by** $(\text{metis } \text{length-rev } \text{rev-filter})$

value $as @ [x] @ bs @ [x] @ cs @ [x] @ ds$

lemma $\text{remove-cycles-removes}: \text{cnt } x \text{ } (\text{remove-cycles } xs \text{ } x \text{ } ys) \leq \max 1 \text{ } (\text{cnt } x \text{ } ys)$

proof $(\text{induction } xs \text{ arbitrary: } ys)$

case Nil **thus** $?case$

by $(\text{simp}, \text{cases } x \in \text{set } ys, (\text{auto } \text{simp}: \text{cnt-rev}[\text{of } x \text{ } ys]))$

next

case $(Cons \text{ } y \text{ } xs)$

thus $?case$

proof $(\text{cases } x = y)$

case $True$

thus $?thesis$ **using** $Cons[\text{of } [y]] \text{ } True$ **by** $auto$

next

case $False$

thus $?thesis$ **using** $Cons[\text{of } y \text{ } \# \text{ } ys]$ **by** $auto$

qed

qed

lemma $\text{remove-cycles-id}: x \notin \text{set } xs \implies \text{remove-cycles } xs \text{ } x \text{ } ys = \text{rev } ys @ xs$

by $(\text{induction } xs \text{ arbitrary: } ys) \text{ } auto$

lemma $\text{remove-cycles-cnt-id}:$

$x \neq y \implies \text{cnt } y \text{ } (\text{remove-cycles } xs \text{ } x \text{ } ys) \leq \text{cnt } y \text{ } ys + \text{cnt } y \text{ } xs$

proof $(\text{induction } xs \text{ arbitrary: } ys \text{ } x)$

case Nil **thus** $?case$ **by** $(\text{simp } \text{add}: \text{cnt-rev})$

next

case $(Cons \text{ } z \text{ } xs)$

thus $?case$

```

proof (cases x = z)
  case True thus ?thesis using Cons.IH[of z [z]] Cons.prem by auto
next
  case False
  thus ?thesis using Cons.IH[of x z # ys] Cons.prem False by auto
qed
qed

```

lemma *remove-cycles-ends-cycle*: $\text{remove-cycles } xs \ x \ ys \neq \text{rev } ys \ @ \ xs \implies x \in \text{set } xs$
using *remove-cycles-id* **by** *fastforce*

lemma *remove-cycles-begins-with*: $x \in \text{set } xs \implies \exists \ zs. \text{remove-cycles } xs \ x \ ys = x \ # \ zs \wedge x \notin \text{set } zs$

```

proof (induction xs arbitrary: ys)
  case Nil thus ?case by auto
next
  case (Cons y xs)
  thus ?case
  proof (cases x = y)
    case True thus ?thesis
    proof (cases x ∈ set xs, goal-cases)
      case 1 with Cons show ?case by auto
    next
      case 2 with remove-cycles-id[of x xs [y]] show ?case by auto
    qed
  next
  case False
  with Cons show ?thesis by auto
  qed
qed

```

lemma *remove-cycles-self*:

$x \in \text{set } xs \implies \text{remove-cycles } (\text{remove-cycles } xs \ x \ ys) \ x \ zs = \text{remove-cycles } xs \ x \ ys$

proof –

```

  assume x: x ∈ set xs
  then obtain ws where ws:  $\text{remove-cycles } xs \ x \ ys = x \ # \ ws \wedge x \notin \text{set } ws$ 
  using remove-cycles-begins-with[OF x, of ys] by blast
  from remove-cycles-id[OF this(2)] have  $\text{remove-cycles } ws \ x \ [x] = x \ # \ ws$ 
by auto
  with ws(1) show  $\text{remove-cycles } (\text{remove-cycles } xs \ x \ ys) \ x \ zs = \text{remove-cycles } xs \ x \ ys$ 
by simp
qed

```

lemma *remove-cycles-one*: $\text{remove-cycles } (as @ x \# xs) x ys = \text{remove-cycles } (x \# xs) x ys$
by (*induction as arbitrary: ys*) *auto*

lemma *remove-cycles-cycles*:

$\exists xxs \ as. \ as @ \text{concat } (\text{map } (\lambda xs. x \# xs) \ xxs) @ \text{remove-cycles } xs \ x \ ys$
 $= xs \wedge x \notin \text{set } as$

if $x \in \text{set } xs$

using *that proof* (*induction xs arbitrary: ys*)

case Nil thus *?case by auto*

next

case (*Cons y xs*)

thus *?case*

proof (*cases x = y*)

case True thus *?thesis*

proof (*cases x ∈ set xs, goal-cases*)

case 1

then obtain *as xxs where as @ concat (map (λxs. y#xs) xxs) @ remove-cycles xs y [y] = xs*

using *Cons.IH[of [y]] by auto*

hence $[] @ \text{concat } (\text{map } (\lambda xs. x \# xs) \ (as \# xxs)) @ \text{remove-cycles } (y \# xs)$
 $x \ ys = y \ \# \ xs$

by (*simp add: ⟨x = y⟩*)

thus *?thesis by fastforce*

next

case 2

hence $\text{remove-cycles } (y \ \# \ xs) \ x \ ys = y \ \# \ xs$ **using** *remove-cycles-id[of x xs [y]] by auto*

hence $[] @ \text{concat } (\text{map } (\lambda xs. x \# xs) \ []) @ \text{remove-cycles } (y \# xs) \ x \ ys$
 $= y \ \# \ xs$ **by auto**

thus *?thesis by fastforce*

qed

next

case False

then obtain *as xxs where as:*

$as @ \text{concat } (\text{map } (\lambda xs. x \# xs) \ xxs) @ \text{remove-cycles } xs \ x \ (y \# ys) =$
 $xs \ x \notin \text{set } as$

using *Cons.IH[of y # ys] Cons.prem by auto*

hence $(y \ \# \ as) @ \text{concat } (\text{map } (\lambda xs. x \# xs) \ xxs) @ \text{remove-cycles } (y \# xs) \ x \ ys = y \ \# \ xs$

using $\langle x \neq y \rangle$ **by auto**

thus *?thesis using as(2) ⟨x ≠ y⟩ by fastforce*

qed

qed

fun *start-remove* :: 'a list \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a list

where

start-remove [] - acc = rev acc |

start-remove (x#xs) y acc =

(if x = y then rev acc @ remove-cycles xs y [y] else *start-remove* xs y (x # acc))

lemma *start-remove-decomp*:

$x \in \text{set } xs \implies \exists \text{ as bs. } xs = \text{as} @ x \# \text{bs} \wedge \text{start-remove } xs \ x \ y = \text{rev } ys @ \text{as} @ \text{remove-cycles } \text{bs } \ x \ [x]$

proof (*induction xs arbitrary: ys*)

case Nil thus ?case by auto

next

case (*Cons y xs*)

thus ?case

proof (*auto, goal-cases*)

case 1

from 1(1)[*of y # ys*]

obtain as bs where

$xs = \text{as} @ x \# \text{bs}$ *start-remove* xs x (y # ys) = rev (y # ys) @ as @ remove-cycles bs x [x]

by blast

hence $y \# xs = (y \# as) @ x \# bs$

start-remove xs x (y # ys) = rev ys @ (y # as) @ remove-cycles bs x [x] **by simp+**

thus ?case by blast

qed

qed

lemma *start-remove-removes*: $\text{cnt } x \ (\text{start-remove } xs \ x \ ys) \leq \text{Suc} \ (\text{cnt } x \ ys)$

proof (*induction xs arbitrary: ys*)

case Nil thus ?case using cnt-rev[*of x ys*] by auto

next

case (*Cons y xs*)

thus ?case

proof (*cases x = y*)

case True

thus ?thesis using remove-cycles-removes[*of y xs [y]*] cnt-rev[*of y ys*]

by auto

next

case False

thus ?thesis using Cons[*of y # ys*] by auto

qed
qed

lemma *start-remove-id[simp]*: $x \notin \text{set } xs \implies \text{start-remove } xs \ x \ ys = \text{rev } ys$
@ *xs*
by (*induction xs arbitrary: ys*) *auto*

lemma *start-remove-cnt-id*:
 $x \neq y \implies \text{cnt } y \ (\text{start-remove } xs \ x \ ys) \leq \text{cnt } y \ ys + \text{cnt } y \ xs$
proof (*induction xs arbitrary: ys*)
 case Nil thus ?case by (*simp add: cnt-rev*)
next
 case (Cons z xs)
 thus ?case
 proof (*cases x = z, goal-cases*)
 case 1 thus ?case using *remove-cycles-cnt-id[of x y xs [x]] by* (*simp add: cnt-rev*)
 next
 case 2 from this(1)[of (z # ys)] this(2,3) show ?case by *auto*
 qed
qed

fun *remove-all-cycles* :: 'a list \Rightarrow 'a list \Rightarrow 'a list
where
 remove-all-cycles [] xs = xs |
 remove-all-cycles (x # xs) ys = remove-all-cycles xs (start-remove ys x [])

lemma *cnt-remove-all-mono*: $\text{cnt } y \ (\text{remove-all-cycles } xs \ ys) \leq \max 1 \ (\text{cnt } y \ ys)$
proof (*induction xs arbitrary: ys*)
 case Nil thus ?case by *auto*
next
 case (Cons x xs)
 thus ?case
 proof (*cases x = y*)
 case True thus ?thesis using *start-remove-removes[of y ys []] Cons[of start-remove ys y []]*
 by *auto*
 next
 case False
 hence $\text{cnt } y \ (\text{start-remove } ys \ x \ []) \leq \text{cnt } y \ ys$
 using *start-remove-cnt-id[of x y ys []] by* *auto*
 thus ?thesis using *Cons[of start-remove ys x []] by* *auto*
 qed

qed

lemma *cnt-remove-all-cycles*: $x \in \text{set } xs \implies \text{cnt } x (\text{remove-all-cycles } xs \ ys) \leq 1$

proof (*induction xs arbitrary: ys*)

case *Nil* **thus** ?*case* **by** *auto*

next

case (*Cons y xs*)

thus ?*case*

using *start-remove-removes[of x ys []] cnt-remove-all-mono[of y xs start-remove ys y []]*

by *auto*

qed

lemma *cnt-mono*:

$\text{cnt } a (b \# xs) \leq \text{cnt } a (b \# c \# xs)$

by (*induction xs*) *auto*

lemma *cnt-distinct-intro*: $\forall x \in \text{set } xs. \text{cnt } x \ xs \leq 1 \implies \text{distinct } xs$

proof (*induction xs*)

case *Nil* **thus** ?*case* **by** *auto*

next

case (*Cons x xs*)

from *this(2)* **have** $\forall x \in \text{set } xs. \text{cnt } x \ xs \leq 1$

by (*metis filter.simps(2) impossible-Cons linorder-class.linear list.set-intros(2) preorder-class.order-trans*)

with *Cons.IH* **have** *distinct xs* **by** *auto*

moreover **have** $x \notin \text{set } xs$ **using** *Cons.prem*

proof (*induction xs*)

case *Nil* **then show** ?*case* **by** *auto*

next

case (*Cons a xs*)

from *this(2)* **have** $\forall xa \in \text{set } (x \# xs). \text{cnt } xa (x \# a \# xs) \leq 1$

by *auto*

then have *: $\forall xa \in \text{set } (x \# xs). \text{cnt } xa (x \# xs) \leq 1$

proof (*safe, goal-cases*)

case (1 b)

then have $\text{cnt } b (x \# a \# xs) \leq 1$ **by** *auto*

with *cnt-mono[of b x xs a]* **show** ?*case* **by** *fastforce*

qed

with *Cons(1)* **have** $x \notin \text{set } xs$ **by** *auto*

moreover **have** $x \neq a$

by (*metis (full-types) Cons.prem One-nat-def * empty-iff filter.simps(2)*)

impossible-Cons

le-0-eq le-Suc-eq length-0-conv list.set(1) list.set-intros(1))

ultimately show *?case by auto*

qed

ultimately show *?case by auto*

qed

lemma *remove-cycles-subs:*

set (remove-cycles xs x ys) \subseteq set xs \cup set ys

by *(induction xs arbitrary: ys; auto; fastforce)*

lemma *start-remove-subs:*

set (start-remove xs x ys) \subseteq set xs \cup set ys

using *remove-cycles-subs by (induction xs arbitrary: ys; auto; fastforce)*

lemma *remove-all-cycles-subs:*

set (remove-all-cycles xs ys) \subseteq set ys

using *start-remove-subs by (induction xs arbitrary: ys, auto) (fastforce+)*

lemma *remove-all-cycles-distinct: set ys \subseteq set xs \implies distinct (remove-all-cycles xs ys)*

proof *–*

assume *set ys \subseteq set xs*

hence $\forall x \in \text{set } ys. \text{cnt } x \text{ (remove-all-cycles xs ys)} \leq 1$ **using** *cnt-remove-all-cycles by fastforce*

hence $\forall x \in \text{set (remove-all-cycles xs ys)}. \text{cnt } x \text{ (remove-all-cycles xs ys)} \leq 1$

using *remove-all-cycles-subs by fastforce*

thus *distinct (remove-all-cycles xs ys) using cnt-distinct-intro by auto*

qed

lemma *distinct-remove-cycles-inv: distinct (xs @ ys) \implies distinct (remove-cycles xs x ys)*

proof *(induction xs arbitrary: ys)*

case *Nil thus ?case by auto*

next

case *(Cons y xs)*

thus *?case by auto*

qed

definition

remove-all x xs = (if x \in set xs then tl (remove-cycles xs x []) else xs)

definition

$remove\text{-}all\text{-}rev\ x\ xs = (if\ x \in set\ xs\ then\ rev\ (tl\ (remove\text{-}cycles\ (rev\ xs)\ x\ []))\ else\ xs)$

lemma *remove-all-distinct:*

$distinct\ xs \implies distinct\ (x\ \#\ remove\text{-}all\ x\ xs)$

proof (cases $x \in set\ xs$, goal-cases)

case 1

from *remove-cycles-begins-with*[OF 1(2), of []] **obtain** zs

where *remove-cycles* $xs\ x\ [] = x\ \#\ zs\ x \notin set\ zs$ **by** *auto*

thus *?thesis* **using** 1(1) *distinct-remove-cycles-inv*[of $xs\ []\ x$] **by** (*simp* *add: remove-all-def*)

next

case 2 **thus** *?thesis* **by** (*simp* *add: remove-all-def*)

qed

lemma *remove-all-removes:*

$x \notin set\ (remove\text{-}all\ x\ xs)$

by (*metis* *list.sel(3)* *remove-all-def* *remove-cycles-begins-with*)

lemma *remove-all-subsets:*

$set\ (remove\text{-}all\ x\ xs) \subseteq set\ xs$

using *remove-cycles-subsets* *remove-all-def*

by (*metis* (*no-types*, *lifting*) *append-Nil2* *list.sel(2)* *list.set-sel(2)* *set-append* *subsetCE* *subsetI*)

lemma *remove-all-rev-distinct:* $distinct\ xs \implies distinct\ (x\ \#\ remove\text{-}all\text{-}rev\ x\ xs)$

proof (cases $x \in set\ xs$, goal-cases)

case 1

then **have** $x \in set\ (rev\ xs)$ **by** *auto*

from *remove-cycles-begins-with*[OF *this*, of []] **obtain** zs

where *remove-cycles* $(rev\ xs)\ x\ [] = x\ \#\ zs\ x \notin set\ zs$ **by** *auto*

thus *?thesis* **using** 1(1) *distinct-remove-cycles-inv*[of $rev\ xs\ []\ x$]

by (*simp* *add: remove-all-rev-def*)

next

case 2 **thus** *?thesis* **by** (*simp* *add: remove-all-rev-def*)

qed

lemma *remove-all-rev-removes:* $x \notin set\ (remove\text{-}all\text{-}rev\ x\ xs)$

by (*metis* *remove-all-def* *remove-all-removes* *remove-all-rev-def* *set-rev*)

lemma *remove-all-rev-subsets:* $set\ (remove\text{-}all\text{-}rev\ x\ xs) \subseteq set\ xs$

by (*metis* *remove-all-def* *remove-all-subsets* *set-rev* *remove-all-rev-def*)

abbreviation *rem-cycles* $i\ j\ xs \equiv \text{remove-all } i\ (\text{remove-all-rev } j\ (\text{remove-all-cycles } xs\ xs))$

lemma *rem-cycles-distinct'*: $i \neq j \implies \text{distinct } (i \# j \# \text{rem-cycles } i\ j\ xs)$

proof –

assume $i \neq j$

have *distinct* (*remove-all-cycles* $xs\ xs$) **by** (*simp add: remove-all-cycles-distinct*)

from *remove-all-rev-distinct*[*OF this*] **have**

distinct (*remove-all-rev* $j\ (\text{remove-all-cycles } xs\ xs)$)

by *simp*

from *remove-all-distinct*[*OF this*] **have** *distinct* ($i \# \text{rem-cycles } i\ j\ xs$)

by *simp*

moreover have

$j \notin \text{set } (\text{rem-cycles } i\ j\ xs)$

using *remove-all-subs remove-all-rev-removes remove-all-removes* **by** *fast-force*

ultimately show *?thesis* **by** (*simp add: <i ≠ j>*)

qed

lemma *rem-cycles-removes-last*: $j \notin \text{set } (\text{rem-cycles } i\ j\ xs)$

by (*meson remove-all-rev-removes remove-all-subs rev-subsetD*)

lemma *rem-cycles-distinct*: *distinct* (*rem-cycles* $i\ j\ xs$)

by (*meson distinct.simps(2) order-refl remove-all-cycles-distinct remove-all-distinct remove-all-rev-distinct*)

lemma *rem-cycles-subs*: $\text{set } (\text{rem-cycles } i\ j\ xs) \subseteq \text{set } xs$

by (*meson order-trans remove-all-cycles-subs remove-all-subs remove-all-rev-subs*)

1.3 Definition of the Algorithm

1.3.1 Definitions

In our formalization of the Floyd-Warshall algorithm, edge weights are from a linearly ordered abelian monoid.

class *linordered-ab-monoid-add* = *linorder* + *ordered-comm-monoid-add*
begin

subclass *linordered-ab-semigroup-add* ..

end

subclass (**in** *linordered-ab-group-add*) *linordered-ab-monoid-add* ..

context *linordered-ab-monoid-add*
begin

type-synonym $'c \text{ mat} = \text{nat} \Rightarrow \text{nat} \Rightarrow 'c$

definition $\text{upd} :: 'c \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'c \Rightarrow 'c \text{ mat}$
where

$$\text{upd } m \ x \ y \ v = m \ (x := (m \ x)) \ (y := v)$$

definition $\text{fw-upd} :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$ **where**
 $\text{fw-upd } m \ k \ i \ j \equiv \text{upd } m \ i \ j \ (\text{min } (m \ i \ j) \ (m \ i \ k + m \ k \ j))$

Recursive version of the two inner loops.

fun $\text{fwi} :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$ **where**
 $\text{fwi } m \ n \ k \ 0 \quad 0 = \text{fw-upd } m \ k \ 0 \ 0 \ |$
 $\text{fwi } m \ n \ k \ (\text{Suc } i) \ 0 = \text{fw-upd } (\text{fwi } m \ n \ k \ i \ n) \ k \ (\text{Suc } i) \ 0 \ |$
 $\text{fwi } m \ n \ k \ i \quad (\text{Suc } j) = \text{fw-upd } (\text{fwi } m \ n \ k \ i \ j) \ k \ i \ (\text{Suc } j)$

Recursive version of the full algorithm.

fun $\text{fw} :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$ **where**
 $\text{fw } m \ n \ 0 = \text{fwi } m \ n \ 0 \ n \ n \ |$
 $\text{fw } m \ n \ (\text{Suc } k) = \text{fwi } (\text{fw } m \ n \ k) \ n \ (\text{Suc } k) \ n \ n$

1.3.2 Elementary Properties

lemma *fw-upd-mono*:

$$\text{fw-upd } m \ k \ i \ j \ i' \ j' \leq m \ i' \ j'$$

by (*cases* $i = i'$, *cases* $j = j'$) (*auto simp*: *fw-upd-def upd-def*)

lemma *fw-upd-out-of-bounds1*:

assumes $i' > i$

shows $(\text{fw-upd } M \ k \ i \ j) \ i' \ j' = M \ i' \ j'$

using *assms unfolding fw-upd-def upd-def* **by** (*auto split*: *split-min*)

lemma *fw-upd-out-of-bounds2*:

assumes $j' > j$

shows $(\text{fw-upd } M \ k \ i \ j) \ i' \ j' = M \ i' \ j'$

using *assms unfolding fw-upd-def upd-def* **by** (*auto split*: *split-min*)

lemma *fwi-out-of-bounds1*:

assumes $i' > n \ i \leq n$

shows $(\text{fwi } M \ n \ k \ i \ j) \ i' \ j' = M \ i' \ j'$

using *assms*

```

apply (induction - (i, j) arbitrary: i j rule: wf-induct[of less-than <*>lex*>
less-than])
  apply (auto; fail)
  subgoal for i j
    by (cases i; cases j; auto simp add: fw-upd-out-of-bounds1)
  done

```

```

lemma fw-out-of-bounds1:
  assumes  $i' > n$ 
  shows  $(fw\ M\ n\ k)\ i'\ j' = M\ i'\ j'$ 
  using assms by (induction k; simp add: fwi-out-of-bounds1)

```

```

lemma fwi-out-of-bounds2:
  assumes  $j' > n\ j \leq n$ 
  shows  $(fwi\ M\ n\ k\ i\ j)\ i'\ j' = M\ i'\ j'$ 
using assms
apply (induction - (i, j) arbitrary: i j rule: wf-induct[of less-than <*>lex*>
less-than])
  apply (auto; fail)
  subgoal for i j
    by (cases i; cases j; auto simp add: fw-upd-out-of-bounds2)
  done

```

```

lemma fw-out-of-bounds2:
  assumes  $j' > n$ 
  shows  $(fw\ M\ n\ k)\ i'\ j' = M\ i'\ j'$ 
  using assms by (induction k; simp add: fwi-out-of-bounds2)

```

```

lemma fwi-invariant-aux-1:
   $j'' \leq j \implies fwi\ m\ n\ k\ i\ j\ i'\ j' \leq fwi\ m\ n\ k\ i\ j''\ i'\ j'$ 
proof (induction j)
  case 0 thus ?case by simp
next
  case (Suc j) thus ?case
  proof (cases  $j'' = Suc\ j$ )
    case True thus ?thesis by simp
  next
    case False
    have fw-upd  $(fwi\ m\ n\ k\ i\ j)\ k\ i\ (Suc\ j)\ i'\ j' \leq fwi\ m\ n\ k\ i\ j\ i'\ j'$ 
      by (simp add: fw-upd-mono)
    thus ?thesis using Suc False by simp
  qed
qed

```

lemma *fwi-invariant*:
 $j \leq n \implies i'' \leq i \implies j'' \leq j$
 $\implies \text{fwi } m \ n \ k \ i \ j \ i' \ j' \leq \text{fwi } m \ n \ k \ i'' \ j'' \ i' \ j'$
proof (*induction i*)
 case 0 **thus** ?*case* **using** *fwi-invariant-aux-1* **by** *auto*
next
 case (*Suc i*) **thus** ?*case*
 proof (*cases i'' = Suc i*)
 case *True* **thus** ?*thesis* **using** *Suc fwi-invariant-aux-1* **by** *simp*
 next
 case *False*
 have $\text{fwi } m \ n \ k \ (\text{Suc } i) \ j \ i' \ j' \leq \text{fwi } m \ n \ k \ (\text{Suc } i) \ 0 \ i' \ j'$
 by (*rule fwi-invariant-aux-1[of 0]; simp*)
 also have $\dots \leq \text{fwi } m \ n \ k \ i \ n \ i' \ j'$ **by** (*simp add: fw-upd-mono*)
 also have $\dots \leq \text{fwi } m \ n \ k \ i \ j \ i' \ j'$ **using** *fwi-invariant-aux-1 False Suc*
by *simp*
 also have $\dots \leq \text{fwi } m \ n \ k \ i'' \ j'' \ i' \ j'$ **using** *Suc False* **by** *simp*
 finally show ?*thesis* **by** *simp*
 qed
qed

lemma *single-row-inv*:
 $j' < j \implies \text{fwi } m \ n \ k \ i' \ j \ i' \ j' = \text{fwi } m \ n \ k \ i' \ j' \ i' \ j'$
proof (*induction j*)
 case 0 **thus** ?*case* **by** *simp*
next
 case (*Suc j*) **thus** ?*case* **by** (*cases j' = j*) (*simp add: fw-upd-def upd-def*)+
qed

lemma *single-iteration-inv'*:
 $i' < i \implies j' \leq n \implies \text{fwi } m \ n \ k \ i \ j \ i' \ j' = \text{fwi } m \ n \ k \ i' \ j' \ i' \ j'$
proof (*induction i arbitrary: j*)
 case 0 **thus** ?*case* **by** *simp*
next
 case (*Suc i*) **thus** ?*case*
 proof (*induction j*)
 case 0 **thus** ?*case*
 proof (*cases i = i', goal-cases*)
 case 2 **thus** ?*case* **by** (*simp add: fw-upd-def upd-def*)
 next
 case 1 **thus** ?*case* **using** *single-row-inv'[of j' n]*
 by (*cases j' = n*) (*fastforce simp add: fw-upd-def upd-def*)+
 qed
 next

```

    case (Suc j) thus ?case by (simp add: fw-upd-def upd-def)
  qed
qed

lemma single-iteration-inv:
   $i' \leq i \implies j' \leq j \implies j \leq n \implies \text{fwi } m \ n \ k \ i \ j \ i' \ j' = \text{fwi } m \ n \ k \ i' \ j' \ i' \ j'$ 
proof (induction i arbitrary: j)
  case 0 thus ?case
  proof (induction j)
    case 0 thus ?case by simp
  next
    case (Suc j) thus ?case using 0 by (cases j' = Suc j) (simp add:
fw-upd-def upd-def)+
  qed
next
  case (Suc i) thus ?case
  proof (induction j)
    case 0 thus ?case by (cases i' = Suc i) (simp add: fw-upd-def upd-def)+
  next
    case (Suc j) thus ?case
    proof (cases i' = Suc i, goal-cases)
      case 1 thus ?case
      proof (cases j' = Suc j, goal-cases)
        case 1 thus ?case by simp
      next
        case 2 thus ?case by (simp add: fw-upd-def upd-def)
      qed
    next
      case 2 thus ?case
      proof (cases j' = Suc j, goal-cases)
        case 1 thus ?case by - (rule single-iteration-inv'; simp)
      next
        case 2 thus ?case by (simp add: fw-upd-def upd-def)
      qed
    qed
  qed
qed
qed

```

```

lemma fwi-innermost-id:
   $i' < i \implies \text{fwi } m \ n \ k \ i' \ j' \ i \ j = m \ i \ j$ 
proof (induction i' arbitrary: j')
  case 0 thus ?case
  proof (induction j')
    case 0 thus ?case by (simp add: fw-upd-def upd-def)
  
```

```

next
  case (Suc j') thus ?case by (auto simp: fw-upd-def upd-def)
qed
next
  case (Suc i') thus ?case
  proof (induction j')
    case 0 thus ?case by (auto simp add: fw-upd-def upd-def)
  next
    case (Suc j') thus ?case by (auto simp add: fw-upd-def upd-def)
  qed
qed

```

lemma *fwi-middle-id*:

$$j' < j \implies i' \leq i \implies \text{fwi } m \ n \ k \ i' \ j' \ i \ j = m \ i \ j$$

proof (*induction* i' *arbitrary*: j')

case 0 **thus** ?case

proof (*induction* j')

case 0 **thus** ?case **by** (*simp* *add*: *fw-upd-def upd-def*)

next

case (Suc j') **thus** ?case **by** (*auto* *simp*: *fw-upd-def upd-def*)

qed

next

case (Suc i') **thus** ?case

proof (*induction* j')

case 0 **thus** ?case **using** *fwi-innermost-id* **by** (*auto* *simp* *add*: *fw-upd-def upd-def*)

next

case (Suc j') **thus** ?case **by** (*auto* *simp* *add*: *fw-upd-def upd-def*)

qed

qed

lemma *fwi-outermost-mono*:

$$i \leq n \implies j \leq n \implies \text{fwi } m \ n \ k \ i \ j \ i \ j \leq m \ i \ j$$

proof (*cases* j)

case 0

assume $i \leq n$

thus ?thesis

proof (*cases* i)

case 0 **thus** ?thesis **using** $\langle j = 0 \rangle$ **by** (*simp* *add*: *fw-upd-def upd-def*)

next

case (Suc i')

hence $\text{fwi } m \ n \ k \ i' \ n \ (\text{Suc } i') \ 0 = m \ (\text{Suc } i') \ 0$ **using** *fwi-innermost-id*

$\langle i \leq n \rangle$ **by** *simp*

thus ?thesis **using** $\langle j = 0 \rangle$ *Suc* **by** (*simp* *add*: *fw-upd-def upd-def*)

qed
next
 case (*Suc j'*)
 assume $i \leq n \ j \leq n$
 hence $fwi \ m \ n \ k \ i \ j' \ i \ (Suc \ j') = m \ i \ (Suc \ j')$
 using *fwi-middle-id Suc* by *simp*
 thus *?thesis* using *Suc* by (*simp add: fw-upd-def upd-def*)
qed

lemma *fwi-mono*:

$fwi \ m \ n \ k \ i' \ j' \ i \ j \leq m \ i \ j$ **if** $i \leq n \ j \leq n$

proof (*cases i' < i*)

case *True*

then have $fwi \ m \ n \ k \ i' \ j' \ i \ j = m \ i \ j$

by (*simp add: fwi-innermost-id*)

then show *?thesis* by *simp*

next

case *False*

show *?thesis*

proof (*cases i' > i*)

case *True*

then have $fwi \ m \ n \ k \ i' \ j' \ i \ j = fwi \ m \ n \ k \ i \ j \ i \ j$

by (*simp add: single-iteration-inv' that(2)*)

with *fwi-outermost-mono[OF that]* **show** *?thesis* by *simp*

next

case *False*

with $\langle \neg \ i' < i \rangle$ **have** [*simp*]: $i' = i$ by *simp*

show *?thesis*

proof (*cases j' < j*)

case *True*

then have $fwi \ m \ n \ k \ i' \ j' \ i \ j = m \ i \ j$

by (*simp add: fwi-middle-id*)

then show *?thesis* by *simp*

next

case *False*

then have $fwi \ m \ n \ k \ i' \ j' \ i \ j = fwi \ m \ n \ k \ i \ j \ i \ j$

by (*cases j' = j; simp add: single-row-inv*)

with *fwi-outermost-mono[OF that]* **show** *?thesis* by *simp*

qed

qed

qed

lemma *Suc-innermost-mono*:

$i \leq n \implies j \leq n \implies fw \ m \ n \ (Suc \ k) \ i \ j \leq fw \ m \ n \ k \ i \ j$

by (simp add: fwi-mono)

lemma *fw-mono*:

$i \leq n \implies j \leq n \implies fw\ m\ n\ k\ i\ j \leq m\ i\ j$

proof (induction k)

case 0 thus ?case using fwi-mono by simp

next

case (Suc k) thus ?case using Suc-innermost-mono[OF Suc.premis, of m k] by simp

qed

Justifies the use of destructive updates in the case that there is no negative cycle for k .

lemma *fwi-step*:

$m\ k\ k \geq 0 \implies i \leq n \implies j \leq n \implies k \leq n \implies fwi\ m\ n\ k\ i\ j\ i\ j = \min(m\ i\ j)\ (m\ i\ k + m\ k\ j)$

proof (induction - (i, j) arbitrary: i j rule: wf-induct[of less-than <lex> less-than],

(auto; fail), goal-cases)

case (1 i' j')

note *assms* = 1(2-)

note *IH* = 1(1)

note [simp] = fwi-innermost-id fwi-middle-id

note *simps* = add-increasing add-increasing2 ord.min-def fw-upd-def upd-def

show ?case

proof (cases i')

case [simp]: 0 thus ?thesis

proof (cases j')

case 0 thus ?thesis by (simp add: fw-upd-def upd-def)

next

case (Suc j)

hence $fwi\ m\ n\ k\ 0\ j\ 0\ (Suc\ j) = m\ 0\ (Suc\ j)$ by simp

moreover have $fwi\ m\ n\ k\ 0\ j\ k\ (Suc\ j) = m\ k\ (Suc\ j)$ by simp

moreover have $fwi\ m\ n\ k\ 0\ j\ 0\ k = m\ 0\ k$

proof (cases j < k)

case True

then show ?thesis by simp

next

case False

then show ?thesis

apply (subst single-iteration-inv; simp)

subgoal

using *assms* Suc by auto

using *assms* by (cases k; simp add: *simps*)

```

    qed
    ultimately show ?thesis using Suc assms by (simp add: fw-upd-def
upd-def)
    qed
  next
  case [simp]: (Suc i)
  show ?thesis
  proof (cases j')
    case 0
    have fwi m n k i n (Suc i) 0 = m (Suc i) 0 by simp
    moreover have fwi m n k i n (Suc i) k = m (Suc i) k by simp
    moreover have fwi m n k i n k 0 = m k 0
    proof (cases i < k)
      case True
      then show ?thesis by simp
    next
      case False
      then show ?thesis
        apply (subst single-iteration-inv; simp)
        using 0 <m k k ≥ -> by (cases k; simp add:_simps)
    qed
    ultimately show ?thesis using 0 by (simp add: fw-upd-def upd-def)
  next
  case Suc-j: (Suc j)
  from <j' ≤ n> <j' = -> have [simp]: j ≤ n Suc j ≤ n by simp+
  have diag: fwi m n k k k k k = m k k if k ≤ i
  proof -
    from that IH assms have fwi m n k k k k k = min (m k k) (m k k
+ m k k) by auto
    with <m k k ≥ 0> <k ≤ n> show ?thesis by (simp add:_simps)
  qed
  have **: fwi m n k i n k k = m k k
  proof (cases i < k)
    case True
    then show ?thesis by simp
  next
    case False
    then show ?thesis
      by (subst single-iteration-inv; simp add: diag <k ≤ n>)
  qed
  have diag2: fwi m n k k j k k = m k k if k ≤ i
  proof (cases j < k)
    case True
    then show ?thesis by simp

```

```

next
  case False
  with  $\langle k \leq i \rangle$  show ?thesis
    by (subst single-iteration-inv; simp add: diag)
qed
have ***: fwi m n k (Suc i) j k (Suc j) = m k (Suc j)
proof (cases Suc i  $\leq$  k)
  case True
  then show ?thesis by simp
next
  case False
  then have fwi m n k k j k (Suc j) = m k (Suc j)
    by simp
  with False  $\langle m k k \geq 0 \rangle$  show ?thesis
    by (subst single-iteration-inv'; simp add:_simps diag2)
qed
have fwi m n k (Suc i) j (Suc i) k = m (Suc i) k
proof (cases j < k)
  case True thus ?thesis by simp
next
  case False
  then show ?thesis
    apply (subst single-iteration-inv; simp)
    apply (cases k)
    subgoal premises prems
    proof -
      have fwi m n 0 i n 0 0  $\geq$  0
        using ** assms(1) prems(2) by force
      moreover have fwi m n 0 i n (Suc i) 0 = m (Suc i) 0
        by simp
      ultimately show ?thesis
        using prems by (simp add:_simps)
    qed
    subgoal premises prems for k'
    proof -
      have fwi m n (Suc k') (Suc i) k' (Suc k') (Suc k')  $\geq$  0
        by (metis ** assms(1,4) fwi-innermost-id fwi-middle-id le-SucE
lessI
          linorder-class.not-le-imp-less prems(2) preorder-class.order-refl
          single-iteration-inv single-iteration-inv'
          )
      with prems show ?thesis
        by (simp add:_simps)
    qed

```

```

    done
  qed
  moreover have fwi m n k (Suc i) j (Suc i) (Suc j) = m (Suc i) (Suc
j) by simp
    ultimately show ?thesis using <j' = -> by (simp add:_simps ***)
  qed
  qed
  qed

```

1.4 Result Under The Absence of Negative Cycles

If the given input graph does not contain any negative cycles, the Floyd-Warshall algorithm computes the **unique** shortest paths matrix corresponding to the graph. It contains the shortest path between any two nodes $i, j \leq n$.

1.4.1 Length of Paths

```

fun len :: 'a mat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat list  $\Rightarrow$  'a where
  len m u v [] = m u v |
  len m u v (w#ws) = m u w + len m w v ws

```

```

lemma len-decomp: xs = ys @ y # zs  $\implies$  len m x z xs = len m x y ys +
len m y z zs
by (induction ys arbitrary: x xs) (simp add: add.assoc)+

```

```

lemma len-comp: len m a c (xs @ b # ys) = len m a b xs + len m b c ys
by (induction xs arbitrary: a) (auto simp: add.assoc)

```

1.4.2 Canonicity

The unique shortest path matrices are in a so-called *canonical form*. We will say that a matrix m is in canonical form for a set of indices I if the following holds:

```

definition canonical-subs :: nat  $\Rightarrow$  nat set  $\Rightarrow$  'a mat  $\Rightarrow$  bool where
  canonical-subs n I m = ( $\forall$  i j k. i  $\leq$  n  $\wedge$  k  $\leq$  n  $\wedge$  j  $\in$  I  $\longrightarrow$  m i k  $\leq$  m i
j + m j k)

```

Similarly we express that m does not contain a negative cycle which only uses intermediate vertices from the set I as follows:

```

abbreviation cyc-free-subs :: nat  $\Rightarrow$  nat set  $\Rightarrow$  'a mat  $\Rightarrow$  bool where
  cyc-free-subs n I m  $\equiv$   $\forall$  i xs. i  $\leq$  n  $\wedge$  set xs  $\subseteq$  I  $\longrightarrow$  len m i i xs  $\geq$  0

```

To prove the main result under *the absence of negative cycles*, we will proceed as follows:

- we show that an invocation of $fwi\ m\ n\ k\ n\ n$ extends canonicity to index k ,
- we show that an invocation of $fw\ m\ n\ n$ computes a matrix in canonical form,
- and finally we show that canonical forms specify the lengths of *shortest paths*, provided that there are no negative cycles.

Canonical forms specify lower bounds for the length of any path.

lemma *canonical-subs-len*:

$M\ i\ j \leq len\ M\ i\ j\ xs$ **if** *canonical-subs* $n\ I\ M\ i \leq n\ j \leq n$ *set* $xs \subseteq I\ I \subseteq \{0..n\}$

using *that*

proof (*induction* xs *arbitrary*: i)

case *Nil* **thus** *?case* **by** *auto*

next

case (*Cons* $x\ xs$)

then have $M\ x\ j \leq len\ M\ x\ j\ xs$ **by** *auto*

from *Cons.prem*s \langle *canonical-subs* $n\ I\ M\rangle$ **have** $M\ i\ j \leq M\ i\ x + M\ x\ j$

unfolding *canonical-subs-def* **by** *auto*

also with *Cons* **have** $\dots \leq M\ i\ x + len\ M\ x\ j\ xs$ **by** (*auto simp add: add-mono*)

finally show *?case* **by** *simp*

qed

This lemma justifies the use of destructive updates under the absence of negative cycles.

lemma *fwi-step'*:

$fwi\ m\ n\ k\ i'\ j'\ i\ j = min\ (m\ i\ j)\ (m\ i\ k + m\ k\ j)$ **if**

$m\ k\ k \geq 0\ i' \leq n\ j' \leq n\ k \leq n\ i \leq i'\ j \leq j'$

using *that* **by** (*subst single-iteration-inv; auto simp: fwi-step*)

An invocation of *fwi* extends canonical forms.

lemma *fwi-canonical-extend*:

canonical-subs $n\ (I \cup \{k\})\ (fwi\ m\ n\ k\ n\ n)$ **if**

canonical-subs $n\ I\ m\ I \subseteq \{0..n\}\ 0 \leq m\ k\ k\ k \leq n$

using *that*

unfolding *canonical-subs-def*

apply *safe*

subgoal for $i\ j\ k'$

apply (*subst fwi-step', (auto; fail)+*)

unfolding *min-def*

proof (*clarsimp, safe, goal-cases*)

```

    case 1
    then show ?case by force
next
case prems: 2
from prems have  $m i k \leq m i j + m j k$ 
  by auto
with prems(10) show ?case
  by (auto simp: add.assoc[symmetric] add-mono intro: order.trans)
next
case prems: 3
from prems have  $m i k \leq m i j + m j k$ 
  by auto
with prems(10) show ?case
  by (auto simp: add.assoc[symmetric] add-mono intro: order.trans)
next
case prems: 4
from prems have  $m k k' \leq m k j + m j k'$ 
  by auto
with prems(10) show ?case
  by (auto simp: add-mono add.assoc intro: order.trans)
next
case prems: 5
from prems have  $m k k' \leq m k j + m j k'$ 
  by auto
with prems(10) show ?case
  by (auto simp: add-mono add.assoc intro: order.trans)
next
case prems: 6
from prems have  $0 \leq m k j + m j k$ 
  by (auto intro: order.trans)
with prems(10) show ?case
  apply -
  apply (rule order.trans, assumption)
  apply (simp add: add.assoc[symmetric])
  by (rule add-mono, auto simp: add-increasing2 add.assoc intro: or-
der.trans)
next
case prems: 7
from prems have  $0 \leq m k j + m j k$ 
  by (auto intro: order.trans)
with prems(10) show ?case
  by (simp add: add.assoc[symmetric])
  (rule add-mono, auto simp: add-increasing2 add.assoc intro: or-
der.trans)

```

```

qed
subgoal for  $i\ j\ k'$ 
  apply (subst fwi-step', (auto; fail)+)+
  unfolding min-def by (auto intro: add-increasing add-increasing2)
done

```

An invocation of *fwi* will not produce a negative diagonal entry if there is no negative cycle.

```

lemma fwi-cyc-free-diag:
  fwi m n k n n i i ≥ 0 if
  cyc-free-subst n I m 0 ≤ m k k k ≤ n k ∈ I i ≤ n
  using that
  apply (subst fwi-step', (auto; fail)+)+
  unfolding min-def
  proof (clarsimp; safe, goal-cases)
    case 1
    have set [] ⊆ I
      by simp
    with 1(1)  $\langle i \leq n \rangle$  show ?case
      by fastforce
    next
    case 2
    then have set [k] ⊆ I
      by simp
    with 2(1)  $\langle i \leq n \rangle$  show ?case by fastforce
qed

```

```

lemma cyc-free-subst-diag:
  m i i ≥ 0 if cyc-free-subst n I m i ≤ n
proof –
  have set [] ⊆ I by auto
  with that show ?thesis by fastforce
qed

```

```

lemma fwi-cyc-free-subst':
  cyc-free-subst n (I ∪ {k}) (fwi m n k n n) if
  cyc-free-subst n I m canonical-subst n I m I ⊆ {0..n} k ≤ n
   $\forall i \leq n. \text{fwi } m\ n\ k\ n\ n\ i\ i \geq 0$ 
proof (safe, goal-cases)
  case prems: (1 i xs)
  from that(1)  $\langle k \leq n \rangle$  have  $0 \leq m\ k\ k$  by (rule cyc-free-subst-diag)
  from that  $\langle 0 \leq m\ k\ k \rangle$  have  $*$ : canonical-subst n (I ∪ {k}) (fwi m n k n n)
  by – (rule fwi-canonical-extend; auto)

```


from *prems* that **have** $0 \leq fwi\ m\ n\ k\ n\ n\ i\ i$ **by** *blast*
also from * *prems* that **have** $fwi\ m\ n\ k\ n\ n\ i\ i \leq len\ (fwi\ m\ n\ k\ n\ n)\ i\ i$
xs
by (*auto intro: canonical-subs-len*)
finally show ?*case* .
qed

lemma *fwi-cyc-free-subs*:
cyc-free-subs $n\ (I \cup \{k\})\ (fwi\ m\ n\ k\ n\ n)$ **if**
cyc-free-subs $n\ (I \cup \{k\})\ m\ canonical-subs\ n\ I\ m\ I \subseteq \{0..n\}\ k \leq n$
proof (*safe, goal-cases*)
case *prems*: ($1\ i\ xs$)
from *that*(1) $\langle k \leq n \rangle$ **have** $0 \leq m\ k\ k$ **by** (*rule cyc-free-subs-diag*)
from *that* $\langle 0 \leq m\ k\ k \rangle$ **have** *: *canonical-subs* $n\ (I \cup \{k\})\ (fwi\ m\ n\ k\ n\ n)$
by – (*rule fwi-canonical-extend; auto*)
from *prems* that $\langle 0 \leq m\ k\ k \rangle$ **have** $0 \leq fwi\ m\ n\ k\ n\ n\ i\ i$ **by** (*auto intro!:*
fwi-cyc-free-diag)
also from * *prems* that **have** $fwi\ m\ n\ k\ n\ n\ i\ i \leq len\ (fwi\ m\ n\ k\ n\ n)\ i\ i$
xs
by (*auto intro: canonical-subs-len*)
finally show ?*case* .
qed

lemma *canonical-subs-empty* [*simp*]:
canonical-subs $n\ \{\}\ m$
unfolding *canonical-subs-def* **by** *simp*

lemma *fwi-neg-diag-neg-cycle*:
 $\exists i \leq n. \exists xs. set\ xs \subseteq \{0..k\} \wedge len\ m\ i\ i\ xs < 0$ **if** $fwi\ m\ n\ k\ n\ n\ i\ i < 0$
 $i \leq n\ k \leq n$
proof (*cases* $m\ k\ k \geq 0$)
case *True*
from *fwi-step*[*of* m , *OF* *True*] *that* **have** $min\ (m\ i\ i)\ (m\ i\ k + m\ k\ i) < 0$
by *auto*
then show ?*thesis*
unfolding *min-def*
proof (*clarsimp split: if-split-asm, goal-cases*)
case 1
then have $len\ m\ i\ i\ [] < 0$ *set* $[] \subseteq \{\}$ **by** *auto*
with $\langle i \leq n \rangle$ **show** ?*case* **by** *fastforce*
next
case 2

```

    then have len m i i [k] < 0 set [k] ⊆ {0..k} by auto
    with ⟨i ≤ n⟩ show ?case by fastforce
qed
next
case False
with ⟨k ≤ n⟩ have len m k k [] < 0 set [] ⊆ {} by auto
with ⟨k ≤ n⟩ show ?thesis by fastforce
qed

```

fwi preserves the length of paths.

lemma *fwi-len*:

```

  ∃ ys. set ys ⊆ set xs ∪ {k} ∧ len (fwi m n k n n) i j xs = len m i j ys
  if i ≤ n j ≤ n k ≤ n m k k ≥ 0 set xs ⊆ {0..n}
  using that

```

proof (*induction xs arbitrary: i*)

```

case Nil
then show ?case
  apply (simp add: fwi-step')
  unfolding min-def
  apply (clarsimp; safe)
  apply (rule exI[where x = []]; simp)
  by (rule exI[where x = [k]]; simp)
next
case (Cons x xs)
then obtain ys where set ys ⊆ set xs ∪ {k} len (fwi m n k n n) x j xs
= len m x j ys
  by force
with Cons.prem1 show ?case
  apply (simp add: fwi-step')
  unfolding min-def
  apply (clarsimp; safe)
  apply (rule exI[where x = x # ys]; auto; fail)
  by (rule exI[where x = k # x # ys]; auto simp: add.assoc)
qed

```

lemma *fwi-neg-cycle-neg-cycle*:

```

  ∃ i ≤ n. ∃ ys. set ys ⊆ set xs ∪ {k} ∧ len m i i ys < 0 if
  len (fwi m n k n n) i i xs < 0 i ≤ n k ≤ n set xs ⊆ {0..n}

```

proof (*cases m k k ≥ 0*)

```

case True
  from fwi-len[OF that(2,2,3), of m, OF True that(4)] that(1,2) show
  ?thesis
  by safe (rule exI conjI | simp)+
next

```

case *False*
then have $\text{len } m \text{ } k \text{ } k \text{ } [] < 0 \text{ set } [] \subseteq \text{set } xs \cup \{k\}$
by *auto*
with $\langle k \leq n \rangle$ **show** *?thesis by (intro exI conjI)*
qed

If the Floyd-Warshall algorithm produces a negative diagonal entry, then there is a negative cycle.

lemma *fw-neg-diag-neg-cycle:*

$\exists i \leq n. \exists ys. \text{set } ys \subseteq \text{set } xs \cup \{0..k\} \wedge \text{len } m \text{ } i \text{ } i \text{ } ys < 0$ **if**
 $\text{len } (fw \text{ } m \text{ } n \text{ } k) \text{ } i \text{ } i \text{ } xs < 0 \text{ } i \leq n \text{ } k \leq n \text{ set } xs \subseteq \{0..n\}$
using *that*
proof (*induction k arbitrary: i xs*)
case *0*
then show *?case by simp (drule fwi-neg-cycle-neg-cycle; auto)*
next
case (*Suc k*)
from *fwi-neg-cycle-neg-cycle[OF Suc.prem(1)[simplified]] Suc.prem*
obtain *i' ys where*
 $i' \leq n \text{ set } ys \subseteq \text{set } xs \cup \{Suc \text{ } k\} \text{ len } (fw \text{ } m \text{ } n \text{ } k) \text{ } i' \text{ } i' \text{ } ys < 0$
by *auto*
with *Suc.prem* **obtain** *i'' zs where*
 $i'' \leq n \text{ set } zs \subseteq \text{set } ys \cup \{0..k\} \text{ len } m \text{ } i'' \text{ } i'' \text{ } zs < 0$
by *atomize-elim (auto intro!: Suc.IH)*
with $\langle \text{set } ys \subseteq \rightarrow \rangle$ **have** $\text{set } zs \subseteq \text{set } xs \cup \{0..Suc \text{ } k\} \wedge \text{len } m \text{ } i'' \text{ } i'' \text{ } zs < 0$
by *force*
with $\langle i'' \leq n \rangle$ **show** *?case by blast*
qed

Main theorem under the absence of negative cycles.

theorem *fw-correct:*

$\text{canonical-subs } n \text{ } \{0..k\} (fw \text{ } m \text{ } n \text{ } k) \wedge \text{cyc-free-subs } n \text{ } \{0..k\} (fw \text{ } m \text{ } n \text{ } k)$
if $\text{cyc-free-subs } n \text{ } \{0..k\} \text{ } m \text{ } k \leq n$
using *that*
proof (*induction k*)
case *0*
then show *?case*
using *fwi-cyc-free-subs[of n {} 0 m] fwi-canonical-extend[of n {}]*
by (*auto simp: cyc-free-subs-diag*)
next
case (*Suc k*)
then have *IH:*
 $\text{canonical-subs } n \text{ } \{0..k\} (fw \text{ } m \text{ } n \text{ } k) \wedge \text{cyc-free-subs } n \text{ } \{0..k\} (fw \text{ } m \text{ } n \text{ } k)$

```

  by fastforce
  have *: {0..Suc k} = {0..k} ∪ {Suc k} by auto
  then have **: canonical-subs n {0..Suc k} (fw m n (Suc k))
    apply simp
    apply (rule fwi-canonical-extend[of n {0..k} - Suc k, simplified])
  subgoal
    using IH ..
  subgoal
    using IH Suc.prem by (auto intro: cyc-free-subs-diag[of n {0..k} fw
m n k])
    by (rule Suc)
  show ?case
  proof (cases ∃ i ≤ n. fw m n (Suc k) i i < 0)
  case True
    then obtain i where i ≤ n len (fw m n (Suc k)) i i [] < 0
      by auto
    from fw-neg-diag-neg-cycle[OF this(2,1) ‹Suc k ≤ n› Suc.prem show
?thesis by fastforce
  next
  case False
    have cyc-free-subs n {0..Suc k} (fw m n (Suc k))
      apply (simp add: *)
      apply (rule fwi-cyc-free-subs'[of n {0..k}, simplified])
      using Suc IH False by force+
    with ** show ?thesis by blast
  qed
qed

```

```

lemmas fw-canonical-subs = fw-correct[THEN conjunct1]
lemmas fw-cyc-free-subs = fw-correct[THEN conjunct2]
lemmas cyc-free-diag = cyc-free-subs-diag

```

1.5 Definition of Shortest Paths

We define the notion of the length of the shortest *simple* path between two vertices, using only intermediate vertices from the set $\{0..k\}$.

definition $D :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ where}$

$D \ m \ i \ j \ k \equiv \text{Min} \ \{ \text{len } m \ i \ j \ xs \mid xs. \text{set } xs \subseteq \{0..k\} \wedge i \notin \text{set } xs \wedge j \notin \text{set } xs \wedge \text{distinct } xs \}$

lemma $\text{distinct-length-le:finite } s \Longrightarrow \text{set } xs \subseteq s \Longrightarrow \text{distinct } xs \Longrightarrow \text{length } xs \leq \text{card } s$

by (metis card-mono distinct-card)

lemma *finite-distinct*: $finite\ s \implies finite\ \{xs.\ set\ xs \subseteq s \wedge distinct\ xs\}$
proof –
 assume *finite s*
 hence $\{xs.\ set\ xs \subseteq s \wedge distinct\ xs\} \subseteq \{xs.\ set\ xs \subseteq s \wedge length\ xs \leq card\ s\}$
 using *distinct-length-le* **by** *auto*
 moreover have $finite\ \{xs.\ set\ xs \subseteq s \wedge length\ xs \leq card\ s\}$
 using *finite-lists-length-le*[*OF* $\langle finite\ s \rangle$] **by** *auto*
 ultimately show *?thesis* **by** (*rule finite-subset*)
qed

lemma *D-base-finite*:
 $finite\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ xs\}$
using *finite-distinct finite-image-set* **by** *blast*

lemma *D-base-finite'*:
 $finite\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ (i\ \# \ j\ \# \ xs)\}$
proof –
 have $\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ (i\ \# \ j\ \# \ xs)\}$
 $\subseteq \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ xs\}$ **by** *auto*
 with *D-base-finite*[*of m i j k*] **show** *?thesis* **by** (*rule rev-finite-subset*)
qed

lemma *D-base-finite''*:
 $finite\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
using *D-base-finite*[*of m i j k*] **by** – (*rule finite-subset, auto*)

definition *cycle-free* :: $'a\ mat \Rightarrow nat \Rightarrow bool$ **where**
 $cycle-free\ m\ n \equiv \forall\ i\ xs.\ i \leq n \wedge set\ xs \subseteq \{0..n\} \longrightarrow$
 $(\forall\ j.\ j \leq n \longrightarrow len\ m\ i\ j\ (rem-cycles\ i\ j\ xs) \leq len\ m\ i\ j\ xs) \wedge len\ m\ i\ i\ xs \geq 0$

lemma *D-eqI*:
 fixes $m\ n\ i\ j\ k$
 defines $A \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\}\}$
 defines $A-distinct \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
 assumes $cycle-free\ m\ n\ i \leq n\ j \leq n\ k \leq n (\bigwedge y.\ y \in A-distinct \implies x \leq y) x \in A$
 shows $D\ m\ i\ j\ k = x$ **using** *assms*
proof –
 let $?S = \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge$

distinct xs
show *?thesis unfolding D-def*
proof (*rule Min-eqI*)
have $?S \subseteq \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ xs\}$ **by** *auto*
thus *finite* $\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
using *D-base-finite[of m i j k]* **by** (*rule finite-subset*)
next
fix *y* **assume** $y \in ?S$
hence $y \in A\text{-distinct}$ **using** *assms(2,7)* **by** *fastforce*
thus $x \leq y$ **using** *assms* **by** *meson*
next
from *assms* **obtain** *xs* **where** $xs: x = len\ m\ i\ j\ xs\ set\ xs \subseteq \{0..k\}$ **by**
auto
let $?ys = rem\text{-cycles}\ i\ j\ xs$
let $?y = len\ m\ i\ j\ ?ys$
from *assms(3-6)* *xs* **have** $*: ?y \leq x$ **by** (*fastforce simp add: cycle-free-def*)
have *distinct*: $i \notin set\ ?ys\ j \notin set\ ?ys\ distinct\ ?ys$
using *rem-cycles-distinct remove-all-removes rem-cycles-removes-last* **by**
fast+
with *xs(2)* **have** $?y \in A\text{-distinct}$ **unfolding** *A-distinct-def* **using**
rem-cycles-subs **by** *fastforce*
hence $x \leq ?y$ **using** *assms* **by** *meson*
moreover **have** $?y \leq x$ **using** *assms(3-6)* *xs* **by** (*fastforce simp add:*
cycle-free-def)
ultimately **have** $x = ?y$ **by** *simp*
thus $x \in ?S$ **using** *distinct xs(2) rem-cycles-subs[of i j xs]* **by** *fastforce*
qed
qed

lemma *D-base-not-empty*:

$\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\} \neq \{\}$

proof –

have $len\ m\ i\ j\ [] \in \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$

by *fastforce*

thus *?thesis* **by** *auto*

qed

lemma *Min-elem-dest*: $finite\ A \implies A \neq \{\} \implies x = Min\ A \implies x \in A$ **by**
simp

lemma *D-dest*: $x = D\ m\ i\ j\ k \implies$

$x \in \{\text{len } m \ i \ j \ xs \mid xs. \text{ set } xs \subseteq \{0..Suc \ k\} \wedge i \notin \text{ set } xs \wedge j \notin \text{ set } xs \wedge \text{ distinct } xs\}$
using *Min-elem-dest*[*OF D-base-finite'' D-base-not-empty*] **by** (*fastforce simp add: D-def*)

lemma *D-dest'*: $x = D \ m \ i \ j \ k \implies x \in \{\text{len } m \ i \ j \ xs \mid xs. \text{ set } xs \subseteq \{0..Suc \ k\}\}$
using *Min-elem-dest*[*OF D-base-finite'' D-base-not-empty*] **by** (*fastforce simp add: D-def*)

lemma *D-dest''*: $x = D \ m \ i \ j \ k \implies x \in \{\text{len } m \ i \ j \ xs \mid xs. \text{ set } xs \subseteq \{0..k\}\}$
using *Min-elem-dest*[*OF D-base-finite'' D-base-not-empty*] **by** (*fastforce simp add: D-def*)

lemma *cycle-free-loop-dest*: $i \leq n \implies \text{ set } xs \subseteq \{0..n\} \implies \text{ cycle-free } m \ n \implies \text{ len } m \ i \ i \ xs \geq 0$
unfolding *cycle-free-def* **by** *auto*

lemma *cycle-free-dest*:
 $\text{ cycle-free } m \ n \implies i \leq n \implies j \leq n \implies \text{ set } xs \subseteq \{0..n\} \implies \text{ len } m \ i \ j \ (\text{rem-cycles } i \ j \ xs) \leq \text{ len } m \ i \ j \ xs$
by (*auto simp add: cycle-free-def*)

definition *cycle-free-up-to* :: 'a mat \Rightarrow nat \Rightarrow nat \Rightarrow bool **where**
 $\text{ cycle-free-up-to } m \ k \ n \equiv \forall \ i \ xs. i \leq n \wedge \text{ set } xs \subseteq \{0..k\} \longrightarrow (\forall \ j. j \leq n \longrightarrow \text{ len } m \ i \ j \ (\text{rem-cycles } i \ j \ xs) \leq \text{ len } m \ i \ j \ xs) \wedge \text{ len } m \ i \ i \ xs \geq 0$

lemma *cycle-free-up-to-loop-dest*:
 $i \leq n \implies \text{ set } xs \subseteq \{0..k\} \implies \text{ cycle-free-up-to } m \ k \ n \implies \text{ len } m \ i \ i \ xs \geq 0$
unfolding *cycle-free-up-to-def* **by** *auto*

lemma *cycle-free-up-to-diag*:
assumes *cycle-free-up-to* $m \ k \ n \ i \leq n$
shows $m \ i \ i \geq 0$
using *cycle-free-up-to-loop-dest*[*OF assms(2) - assms(1)*], *of []* **by** *auto*

lemma *D-eqI2*:
fixes $m \ n \ i \ j \ k$
defines $A \equiv \{\text{len } m \ i \ j \ xs \mid xs. \text{ set } xs \subseteq \{0..k\}\}$
defines $A\text{-distinct} \equiv \{\text{len } m \ i \ j \ xs \mid xs. \text{ set } xs \subseteq \{0..k\} \wedge i \notin \text{ set } xs \wedge j \notin \text{ set } xs \wedge \text{ distinct } xs\}$
assumes *cycle-free-up-to* $m \ k \ n \ i \leq n \ j \leq n \ k \leq n$
 $(\bigwedge y. y \in A\text{-distinct} \implies x \leq y) \ x \in A$

```

shows  $D\ m\ i\ j\ k = x$  using assms
proof –
  show ?thesis
  proof (simp add: D-def A-distinct-def[symmetric], rule Min-eqI)
    show finite A-distinct using D-base-finite''[of m i j k] unfolding
A-distinct-def by auto
  next
    fix y assume  $y \in A\text{-distinct}$ 
    thus  $x \leq y$  using assms by meson
  next
    from assms obtain xs where  $xs: x = \text{len } m\ i\ j\ xs$   $\text{set } xs \subseteq \{0..k\}$  by
auto
    let  $?ys = \text{rem-cycles } i\ j\ xs$ 
    let  $?y = \text{len } m\ i\ j\ ?ys$ 
    from assms(3-6) xs have  $*: ?y \leq x$  by (fastforce simp add: cycle-free-up-to-def)
    have distinct: i ∉ set ?ys j ∉ set ?ys distinct ?ys
    using rem-cycles-distinct remove-all-removes rem-cycles-removes-last by
fast+
    with  $xs(2)$  have  $?y \in A\text{-distinct}$  unfolding A-distinct-def using
rem-cycles-subs by fastforce
    hence  $x \leq ?y$  using assms by meson
    moreover have  $?y \leq x$  using assms(3-6) xs by (fastforce simp add:
cycle-free-up-to-def)
    ultimately have  $x = ?y$  by simp
    then show  $x \in A\text{-distinct}$  using distinct xs(2) rem-cycles-subs[of i j xs]
    unfolding A-distinct-def by fastforce
  qed
qed

```

1.5.1 Connecting the Algorithm to the Notion of Shortest Paths

Under the absence of negative cycles, the Floyd-Warshall algorithm correctly computes the length of the shortest path between any pair of vertices i, j .

lemma *canonical-D*:

assumes

cycle-free-up-to m k n canonical-subs n {0..k} m i ≤ n j ≤ n k ≤ n

shows $D\ m\ i\ j\ k = m\ i\ j$

using *assms*

apply –

apply (*rule D-eqI2*)

apply (*assumption | simp; fail*)+

subgoal

by (*auto intro: canonical-subs-len*)

apply *clarsimp*
by (*rule exI*[**where** $x = []$]) *auto*

theorem *fw-subst-len*:

(*fw m n k*) $i j \leq \text{len } m \ i \ j \ xs$ **if**
cyc-free-subst n $\{0..k\}$ $m \ k \leq n \ i \leq n \ j \leq n$ *set xs* $\subseteq I \ I \subseteq \{0..k\}$

proof –

from *fw-correct*[*OF that*(1,2)] **have** *canonical-subst n* $\{0..k\}$ (*fw m n k*)

..

from *canonical-subst-len*[*OF this, of i j xs*] **that** **have** *fw m n k* $i j \leq \text{len}$
(*fw m n k*) $i j \ xs$

by *auto*

also from *that*(2–) **have** $\dots \leq \text{len } m \ i \ j \ xs$

proof (*induction xs arbitrary: i*)

case *Nil*

then show *?case by* (*auto intro: fw-mono*)

next

case (*Cons x xs*)

then have $\text{len } (fw \ m \ n \ k) \ x \ j \ xs \leq \text{len } m \ x \ j \ xs$

by *auto*

moreover from *Cons.prem*s **have** *fw m n k i x* $\leq m \ i \ x$ **by** – (*rule fw-mono; auto*)

ultimately show *?case by* (*auto simp: add-mono*)

qed

finally show *?thesis by auto*

qed

This shows that the value calculated by *fw* for a pair i, j always corresponds to the length of an actual path between i and j .

lemma *fw-len'*:

$\exists \ xs. \ \text{set } xs \subseteq \{k\} \wedge \text{fw } m \ n \ k \ i' \ j' \ i \ j = \text{len } m \ i \ j \ xs$ **if**

$m \ k \geq 0 \ i' \leq n \ j' \leq n \ k \leq n \ i \leq i' \ j \leq j'$

using *that* **apply** (*subst fw-step'*; *auto*)

unfolding *min-def*

apply (*clarsimp; safe*)

apply (*rule exI*[**where** $x = []$]; *auto; fail*)

by (*rule exI*[**where** $x = [k]$]; *auto; fail*)

The same result for *fw*.

lemma *fw-len*:

$\exists \ xs. \ \text{set } xs \subseteq \{0..k\} \wedge \text{fw } m \ n \ k \ i \ j = \text{len } m \ i \ j \ xs$ **if**

cyc-free-subst n $\{0..k\}$ $m \ i \leq n \ j \leq n \ k \leq n$

using *that*

proof (*induction k arbitrary: i j*)
case 0
from *cyc-free-sub-diag*[*OF this(1)*] **have** $m\ 0\ 0 \geq 0$ **by** *blast*
with 0 **show** ?*case* **by** (*auto intro: fwi-len'*)
next
case (*Suc k*)
have *IH*: $\exists xs. \text{set } xs \subseteq \{0..k\} \wedge \text{fw } m\ n\ k\ i\ j = \text{len } m\ i\ j\ xs$ **if** $i \leq n\ j$
 $\leq n$ **for** $i\ j$
apply (*rule Suc.IH*)
using *Suc.prem*s that **by** *force+*
from *fw-cyc-free-sub*s[*OF Suc.prem*s(1,4)] **have** *cyc-free-sub*s $n\ \{0..Suc\ k\}$ (*fw m n (Suc k)*) .
then have $0 \leq \text{fw } m\ n\ k\ (Suc\ k)\ (Suc\ k)$ **using** *IH Suc.prem*s(1, 4) **by** *fastforce*
with *Suc.prem*s *fwi-len'*[*of fw m n k Suc k n n i j*] **obtain** *xs* **where**
 $\text{set } xs \subseteq \{Suc\ k\}$ *fwi* (*fw m n k*) $n\ (Suc\ k)\ n\ n\ i\ j = \text{len } (fw\ m\ n\ k)\ i\ j\ xs$
by *auto*
moreover from *Suc.prem*s(2-) *this(1)* **have**
 $\exists ys. \text{set } ys \subseteq \{0..Suc\ k\} \wedge \text{len } (fw\ m\ n\ k)\ i\ j\ xs = \text{len } m\ i\ j\ ys$
proof (*induction xs arbitrary: i*)
case *Nil*
then show ?*case* **by** (*force dest: IH*)
next
case (*Cons x xs*)
then obtain *ys* **where** *ys*:
 $\text{set } ys \subseteq \{0..Suc\ k\}$ $\text{len } (fw\ m\ n\ k)\ x\ j\ xs = \text{len } m\ x\ j\ ys$
by *force*
moreover from *IH*[*of i x*] *Cons.prem*s **obtain** *zs* **where**
 $\text{set } zs \subseteq \{0..k\}$ $\text{fw } m\ n\ k\ i\ x = \text{len } m\ i\ x\ zs$
by *auto*
ultimately have
 $\text{set } (zs\ @\ x\ \#\ ys) \subseteq \{0..Suc\ k\}$ $\text{len } (fw\ m\ n\ k)\ i\ j\ (x\ \#\ xs) = \text{len } m\ i\ j\ (zs\ @\ x\ \#\ ys)$
using $\langle Suc\ k \leq n \rangle$ $\langle \text{set } (x\ \#\ xs) \subseteq \cdot \rangle$ **by** (*auto simp: len-comp*)
then show ?*case* **by** (*intro exI conjI*)
qed
ultimately show ?*case* **by** *auto*
qed

1.6 Intermezzo: Equivalent Characterizations of Cycle-Freeness

1.6.1 Shortening Negative Cycles

lemma *remove-cycles-neg-cycles-aux*:

```

fixes  $i$   $xs$   $ys$ 
defines  $xs' \equiv i \# ys$ 
assumes  $i \notin \text{set } ys$ 
assumes  $i \in \text{set } xs$ 
assumes  $xs = as @ \text{concat} (\text{map} ((\#) i) xss) @ xs'$ 
assumes  $\text{len } m \ i \ j \ ys > \text{len } m \ i \ j \ xs$ 
shows  $\exists ys. \text{set } ys \subseteq \text{set } xs \wedge \text{len } m \ i \ i \ ys < 0$  using assms
proof (induction xss arbitrary: xs as)
  case Nil
    with Nil show ?case
    proof (cases len m i i as  $\geq 0$ , goal-cases)
      case 1
        from this(4,6) len-decomp[of xs as i ys m i j]
        have  $\text{len } m \ i \ j \ xs = \text{len } m \ i \ i \ as + \text{len } m \ i \ j \ ys$  by simp
        with 1(11)
        have  $\text{len } m \ i \ j \ ys \leq \text{len } m \ i \ j \ xs$  using add-mono by fastforce
        thus ?thesis using Nil(5) by auto
      next
        case 2 thus ?case by auto
      qed
    next
      case (Cons zs xss)
        let  $?xs = zs @ \text{concat} (\text{map} ((\#) i) xss) @ xs'$ 
        from Cons show ?case
        proof (cases len m i i as  $\geq 0$ , goal-cases)
          case 1
            from this(5,7) len-decomp add-mono
            have  $\text{len } m \ i \ j \ ?xs \leq \text{len } m \ i \ j \ xs$  by fastforce
            hence  $4:\text{len } m \ i \ j \ ?xs < \text{len } m \ i \ j \ ys$  using 1(6) by simp
            have  $2:i \in \text{set } ?xs$  using Cons(2) by auto
            have  $\text{set } ?xs \subseteq \text{set } xs$  using Cons(5) by auto
            moreover from Cons(1)[OF 1(2,3) 2 - 4] have  $\exists ys. \text{set } ys \subseteq \text{set } ?xs$ 
             $\wedge \text{len } m \ i \ i \ ys < 0$  by auto
            ultimately show ?case by blast
          next
            case 2
              from this(5,7) show ?case by auto
            qed
          qed
        qed
      qed

```

```

lemma add-lt-neutral:  $a + b < b \implies a < 0$ 
proof (rule ccontr)
  assume  $a + b < b \neg a < 0$ 
  hence  $a \geq 0$  by auto

```

from *add-mono*[*OF this, of b b*] $\langle a + b < b \rangle$ **show** *False* **by** *auto*
qed

lemma *remove-cycles-neg-cycles-aux'*:

fixes $j\ xs\ ys$

assumes $j \notin \text{set } ys$

assumes $j \in \text{set } xs$

assumes $xs = ys @ j \# \text{concat } (\text{map } (\lambda xs. xs @ [j])\ xss) @ as$

assumes $\text{len } m\ i\ j\ ys > \text{len } m\ i\ j\ xs$

shows $\exists\ ys. \text{set } ys \subseteq \text{set } xs \wedge \text{len } m\ j\ j\ ys < 0$ **using** *assms*

proof (*induction xss arbitrary: xs as*)

case *Nil*

show *?case*

proof (*cases len m j j as ≥ 0*)

case *True*

from *Nil(3)* *len-decomp*[*of xs ys j as m i j*]

have $\text{len } m\ i\ j\ xs = \text{len } m\ i\ j\ ys + \text{len } m\ j\ j\ as$ **by** *simp*

with *True*

have $\text{len } m\ i\ j\ ys \leq \text{len } m\ i\ j\ xs$ **using** *add-mono* **by** *fastforce*

with *Nil* **show** *?thesis* **by** *auto*

next

case *False* **with** *Nil* **show** *?thesis* **by** *auto*

qed

next

case (*Cons zs xss*)

let $?xs = ys @ j \# \text{concat } (\text{map } (\lambda xs. xs @ [j])\ xss) @ as$

let $?t = \text{concat } (\text{map } (\lambda xs. xs @ [j])\ xss) @ as$

show *?case*

proof (*cases len m i j ?xs \leq len m i j xs*)

case *True*

hence $4:\text{len } m\ i\ j\ ?xs < \text{len } m\ i\ j\ ys$ **using** *Cons(5)* **by** *simp*

have $2:j \in \text{set } ?xs$ **using** *Cons(2)* **by** *auto*

have $\text{set } ?xs \subseteq \text{set } xs$ **using** *Cons(4)* **by** *auto*

moreover from *Cons(1)*[*OF Cons(2) 2 - 4*] **have** $\exists\ ys. \text{set } ys \subseteq \text{set } ?xs$

$\wedge \text{len } m\ j\ j\ ys < 0$ **by** *blast*

ultimately show *?thesis* **by** *blast*

next

case *False*

hence $\text{len } m\ i\ j\ xs < \text{len } m\ i\ j\ ?xs$ **by** *auto*

from *this* *len-decomp* *Cons(4)* *add-mono*

have $\text{len } m\ j\ j\ (\text{concat } (\text{map } (\lambda xs. xs @ [j])\ (zs \# xss)) @ as) < \text{len } m$

$j\ j\ ?t$

using *False local.leI* **by** *fastforce*

hence $\text{len } m\ j\ j\ (zs @ j \# ?t) < \text{len } m\ j\ j\ ?t$ **by** *simp*

with *len-decomp*[*of zs @ j # ?t zs j ?t m j j*]
have *len m j j zs + len m j j ?t < len m j j ?t* **by** *auto*
hence *len m j j zs < 0* **using** *add-lt-neutral* **by** *auto*
thus *?thesis* **using** *Cons.prem3* **by** *auto*
qed
qed

lemma *add-le-impl*: $a + b < a + c \implies b < c$
proof (*rule ccontr*)
assume $a + b < a + c \neg b < c$
hence $b \geq c$ **by** *auto*
from *add-mono*[*OF - this, of a a*] $\langle a + b < a + c \rangle$ **show** *False* **by** *auto*
qed

lemma *start-remove-neg-cycles*:

$len\ m\ i\ j\ (start\ remove\ xs\ k\ []) > len\ m\ i\ j\ xs \implies \exists\ ys.\ set\ ys \subseteq set\ xs \wedge len\ m\ k\ k\ ys < 0$

proof–

let $?xs = start\ remove\ xs\ k\ []$
assume *len-lt*: $len\ m\ i\ j\ ?xs > len\ m\ i\ j\ xs$
hence $k \in set\ xs$ **using** *start-remove-id* **by** *fastforce*
from *start-remove-decomp*[*OF this, of []*] **obtain** *as bs* **where** *as-bs*:
 $xs = as\ @\ k\ \#\ bs\ ?xs = as\ @\ remove\ cycles\ bs\ k\ [k]$
by *fastforce*
let $?xs' = remove\ cycles\ bs\ k\ [k]$
have $k \in set\ bs$ **using** *as-bs len-lt remove-cycles-id* **by** *fastforce*
then obtain *ys* **where** $?xs = as\ @\ k\ \#\ ys\ ?xs' = k\ \#\ ys\ k \notin set\ ys$
using *as-bs(2) remove-cycles-begins-with*[*OF <k ∈ set bs>*] **by** *auto*
have *len-lt'*: $len\ m\ k\ j\ bs < len\ m\ k\ j\ ys$
using *len-decomp*[*OF as-bs(1), of m i j*] *len-decomp*[*OF ys(1), of m i j*]
len-lt add-le-impl **by** *metis*
from *remove-cycles-cycles*[*OF <k ∈ set bs>*] **obtain** *xss as'*
where $as' @ concat\ (map\ ((\#)\ k)\ xss) @ ?xs' = bs$ **by** *fastforce*
hence $as' @ concat\ (map\ ((\#)\ k)\ xss) @ k\ \#\ ys = bs$ **using** *ys(2)* **by**
simp
from *remove-cycles-neg-cycles-aux*[*OF <k ∉ set ys> <k ∈ set bs> this[symmetric]* *len-lt'*]
show *?thesis* **using** *as-bs(1)* **by** *auto*
qed

lemma *remove-all-cycles-neg-cycles*:

$len\ m\ i\ j\ (remove\ all\ cycles\ ys\ xs) > len\ m\ i\ j\ xs$
 $\implies \exists\ ys\ k.\ set\ ys \subseteq set\ xs \wedge k \in set\ xs \wedge len\ m\ k\ k\ ys < 0$

proof (*induction ys arbitrary: xs*)

```

  case Nil thus ?case by auto
next
case (Cons y ys)
let ?xs = start-remove xs y []
show ?case
proof (cases len m i j xs < len m i j ?xs)
  case True
  with start-remove-id have y ∈ set xs by fastforce
  with start-remove-neg-cycles[OF True] show ?thesis by blast
next
  case False
  with Cons(2) have len m i j ?xs < len m i j (remove-all-cycles (y #
ys) xs) by auto
  hence len m i j ?xs < len m i j (remove-all-cycles ys ?xs) by auto
  from Cons(1)[OF this] show ?thesis using start-remove-subst[of xs y []]
by auto
qed
qed

```

lemma *concat-map-cons-rev*:

```

  rev (concat (map ((#) j) xss)) = concat (map (λ xs. xs @ [j]) (rev (map
rev xss)))
by (induction xss) auto

```

lemma *negative-cycle-dest*: $\text{len } m \ i \ j \ (\text{rem-cycles } i \ j \ xs) > \text{len } m \ i \ j \ xs$

$\implies \exists \ i' \ ys. \ \text{len } m \ i' \ i' \ ys < 0 \wedge \text{set } ys \subseteq \text{set } xs \wedge \ i' \in \text{set } (i \ # \ j \ # \ xs)$

proof –

```

let ?xsij = rem-cycles i j xs
let ?xsj = remove-all-rev j (remove-all-cycles xs xs)
let ?xs = remove-all-cycles xs xs
assume len-lt: len m i j ?xsij > len m i j xs
show ?thesis
proof (cases len m i j ?xsij ≤ len m i j ?xsj)
  case True
  hence len-lt: len m i j ?xsj > len m i j xs using len-lt by simp
  show ?thesis
  proof (cases len m i j ?xsj ≤ len m i j ?xs)
    case True
    hence len m i j ?xs > len m i j xs using len-lt by simp
    with remove-all-cycles-neg-cycles[OF this] show ?thesis by auto
  next
    case False
    then have len-lt': len m i j ?xsj > len m i j ?xs by simp

```

```

show ?thesis
proof (cases  $j \in \text{set } ?xs$ )
  case False
    thus ?thesis using len- $lt'$  by (simp add: remove-all-rev-def)
  next
    case True
      from remove-all-rev-removes[of  $j$ ] have  $1: j \notin \text{set } ?xsj$  by simp
      from True have  $j \in \text{set } (\text{rev } ?xs)$  by auto
      from remove-cycles-cycles[OF this] obtain  $xss$  as where  $as$ :
         $as @ \text{concat } (\text{map } ((\#) j) xss) @ \text{remove-cycles } (\text{rev } ?xs) j [] = \text{rev } ?xs$ 
         $j \notin \text{set } as$ 
      by blast
      from True have  $?xsj = \text{rev } (\text{tl } (\text{remove-cycles } (\text{rev } ?xs) j []))$  by
        (simp add: remove-all-rev-def)
      with remove-cycles-begins-with[OF  $\langle j \in \text{set } (\text{rev } ?xs) \rangle$ , of []]
      have  $\text{remove-cycles } (\text{rev } ?xs) j [] = j \# \text{rev } ?xsj$   $j \notin \text{set } ?xsj$ 
      by auto
      with  $as(1)$  have  $xss: as @ \text{concat } (\text{map } ((\#) j) xss) @ j \# \text{rev } ?xsj$ 
         $= \text{rev } ?xs$  by simp
      hence  $\text{rev } (as @ \text{concat } (\text{map } ((\#) j) xss) @ j \# \text{rev } ?xsj) = ?xs$ 
by simp
      hence  $?xsj @ j \# \text{rev } (\text{concat } (\text{map } ((\#) j) xss)) @ \text{rev } as = ?xs$ 
by simp
      hence  $?xsj @ j \# \text{concat } (\text{map } (\lambda xs. xs @ [j]) (\text{rev } (\text{map } \text{rev } xss)))$ 
         $@ \text{rev } as = ?xs$ 
      by (simp add: concat-map-cons-rev)
      from remove-cycles-neg-cycles-aux'[OF 1 True this[symmetric]
        len- $lt'$ ]
      show ?thesis using remove-all-cycles-subs by fastforce
    qed
  qed
next
  case False
    hence len- $lt'$ :  $\text{len } m i j ?xsij > \text{len } m i j ?xsj$  by simp
    show ?thesis
    proof (cases  $i \in \text{set } ?xsj$ )
      case False
        thus ?thesis using len- $lt'$  by (simp add: remove-all-def)
      next
        case True
          from remove-all-removes[of  $i$ ] have  $1: i \notin \text{set } ?xsij$  by (simp add:
            remove-all-def)
          from remove-cycles-cycles[OF True] obtain  $xss$  as where  $as$ :
             $as @ \text{concat } (\text{map } ((\#) i) xss) @ \text{remove-cycles } ?xsj i [] = ?xsj$ 
             $i \notin \text{set } as$ 

```

as **by** *blast*
from *True* **have** $?xsij = tl (remove-cycles ?xsj i [])$ **by** (*simp add: remove-all-def*)
with *remove-cycles-begins-with*[*OF True, of []*]
have $remove-cycles ?xsj i [] = i \# ?xsij \ i \notin set ?xsij$
by *auto*
with *as(1)* **have** $xss: as @ concat (map ((\#) i) xss) @ i \# ?xsij = ?xsj$ **by** *simp*
from *remove-cycles-neg-cycles-aux*[*OF 1 True this[symmetric] len-lt'*]
show *?thesis* **using** *remove-all-rev-subst remove-all-cycles-subst* **by** *fastforce*
qed
qed
qed

1.6.2 Cycle-Freeness

lemma *cycle-free-alt-def*:

$cycle-free\ M\ n \longleftrightarrow cycle-free-up-to\ M\ n\ n$
unfolding *cycle-free-def cycle-free-up-to-def ..*

lemma *negative-cycle-dest-diag*:

$\neg cycle-free-up-to\ m\ k\ n \implies k \leq n \implies \exists\ i\ xs. i \leq n \wedge set\ xs \subseteq \{0..k\}$
 $\wedge len\ m\ i\ i\ xs < 0$

proof (*auto simp: cycle-free-up-to-def, goal-cases*)

case (*1 i xs j*)

from *this(5)* **have** $len\ m\ i\ j\ xs < len\ m\ i\ j (rem-cycles\ i\ j\ xs)$ **by** *auto*

from *negative-cycle-dest*[*OF this*] **obtain** $i'\ ys$

where $*: len\ m\ i'\ i'\ ys < 0\ set\ ys \subseteq set\ xs\ i' \in set\ (i \# j \# xs)$ **by** *auto*

from *this(2,3) 1(1-4)* **have** $set\ ys \subseteq \{0..k\}\ i' \leq n$ **by** *auto*

with $*$ **show** *?case* **by** *auto*

next

case *2* **then show** *?case* **by** *fastforce*

qed

lemma *negative-cycle-dest-diag'*:

$\neg cycle-free\ m\ n \implies \exists\ i\ xs. i \leq n \wedge set\ xs \subseteq \{0..n\} \wedge len\ m\ i\ i\ xs < 0$

by (*rule negative-cycle-dest-diag*) (*auto simp: cycle-free-alt-def*)

abbreviation *cyc-free* :: $'a\ mat \Rightarrow nat \Rightarrow bool$ **where**

$cyc-free\ m\ n \equiv \forall\ i\ xs. i \leq n \wedge set\ xs \subseteq \{0..n\} \longrightarrow len\ m\ i\ i\ xs \geq 0$

lemma *cycle-free-diag-intro*:

$cyc-free\ m\ n \implies cycle-free\ m\ n$

using *negative-cycle-dest-diag'* **by** *force*

lemma *cycle-free-diag-equiv:*

cyc-free m n \longleftrightarrow *cycle-free m n* **using** *negative-cycle-dest-diag'*
by (*force simp: cycle-free-def*)

lemma *cycle-free-diag-dest:*

cycle-free m n \implies *cyc-free m n*
using *cycle-free-diag-equiv* **by** *blast*

lemma *cycle-free-up-to-diag-equiv:*

cycle-free-up-to m k n \longleftrightarrow *cyc-free-sub n {0..k} m* **if** $k \leq n$
using *negative-cycle-dest-diag[of m k n]* **that** **by** (*force simp: cycle-free-up-to-def*)

theorem *fw-shortest-path-up-to:*

$D m i j k = fw m n k i j$ **if** *cyc-free-sub n {0..k} m* $i \leq n$ $j \leq n$ $k \leq n$

proof –

from *that(1,4)* **have** *cycle-free: cycle-free-up-to m k n* **by** (*subst cycle-free-up-to-diag-equiv*)

from *that* **have** *canonical-sub n {0..k} (fw m n k) cyc-free-sub n {0..k}*
(*fw m n k*)

by (*auto dest: fw-correct*)

show *?thesis*

proof (*rule D-eqI2[where n = n], safe, goal-cases*)

case (*5 y xs*)

with *that(1)* **that** **show** *?case* **by** (*auto intro: fw-sub-len*)

next

case *6*

from *fw-len[OF that(1) that(2–)]* **show** *?case* **by** *blast*

qed (*rule that cycle-free*)+

qed

We do not need to prove this because the definitions match.

lemma

cyc-free m n \longleftrightarrow *cyc-free-sub n {0..n} m ..*

lemma *cycle-free-cycle-free-up-to:*

cycle-free m n \implies $k \leq n \implies$ *cycle-free-up-to m k n*

unfolding *cycle-free-def cycle-free-up-to-def* **by** *force*

lemma *cycle-free-diag:*

cycle-free m n \implies $i \leq n \implies$ $0 \leq m i i$

using *cycle-free-up-to-diag[OF cycle-free-cycle-free-up-to]* **by** *blast*

corollary *fw-shortest-path:*

$cyc\text{-free } m \ n \implies i \leq n \implies j \leq n \implies k \leq n \implies D \ m \ i \ j \ k = fw \ m \ n \ k \ i$
 j

by (*rule fw-shortest-path-up-to; force*)

corollary *fw-shortest:*

assumes $cyc\text{-free } m \ n \ i \leq n \ j \leq n \ k \leq n$

shows $fw \ m \ n \ n \ i \ j \leq fw \ m \ n \ n \ i \ k + fw \ m \ n \ n \ k \ j$

using $fw\text{-canonical-subs}[OF \ assms(1)] \ assms(2-)$ **unfolding** *canonical-subs-def*
by *auto*

1.7 Result Under the Presence of Negative Cycles

Under the presence of negative cycles, the Floyd-Warshall algorithm will detect the situation by computing a negative diagonal entry.

lemma *not-cylce-free-dest:* $\neg \text{cycle-free } m \ n \implies \exists k \leq n. \neg \text{cycle-free-up-to}$
 $m \ k \ n$

by (*auto simp add: cycle-free-def cycle-free-up-to-def*)

lemma *D-not-diag-le:*

$(x :: 'a) \in \{len \ m \ i \ j \ xs \mid xs. \text{set } xs \subseteq \{0..k\} \wedge i \notin \text{set } xs \wedge j \notin \text{set } xs \wedge$
 $\text{distinct } xs\}$

$\implies D \ m \ i \ j \ k \leq x$ **using** $Min\text{-le}[OF \ D\text{-base-finite}']$ **by** (*auto simp add:*
 $D\text{-def}$)

lemma *D-not-diag-le':* $\text{set } xs \subseteq \{0..k\} \implies i \notin \text{set } xs \implies j \notin \text{set } xs \implies$
 $\text{distinct } xs$

$\implies D \ m \ i \ j \ k \leq len \ m \ i \ j \ xs$ **using** $Min\text{-le}[OF \ D\text{-base-finite}']$ **by** (*fastforce*
 $\text{simp add: } D\text{-def}$)

lemma *nat-upto-subs-top-removal':*

$S \subseteq \{0..Suc \ n\} \implies Suc \ n \notin S \implies S \subseteq \{0..n\}$

apply (*induction n*)

apply *safe*

apply (*rename-tac x*)

apply (*case-tac x = Suc 0; fastforce*)

apply (*rename-tac n x*)

apply (*case-tac x = Suc (Suc n); fastforce*)

done

lemma *nat-upto-subs-top-removal:*

$S \subseteq \{0..n::nat\} \implies n \notin S \implies S \subseteq \{0..n - 1\}$

using *nat-upto-subs-top-removal'* **by** (*cases n; simp*)

Monotonicity with respect to k .

lemma *fw-invariant*:

$k' \leq k \implies i \leq n \implies j \leq n \implies k \leq n \implies fw\ m\ n\ k\ i\ j \leq fw\ m\ n\ k'\ i\ j$

proof (*induction k*)

case 0

then show *?case* **by** (*auto intro: fwi-invariant*)

next

case (*Suc k*)

show *?case*

proof (*cases k' = Suc k*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

with *Suc* **have** $fw\ m\ n\ k\ i\ j \leq fw\ m\ n\ k'\ i\ j$

by *auto*

moreover from $\langle i \leq n \rangle \langle j \leq n \rangle$ **have** $fw\ m\ n\ (Suc\ k)\ i\ j \leq fw\ m\ n\ k\ i$

j

by (*auto intro: fwi-mono*)

ultimately show *?thesis* **by** *auto*

qed

qed

lemma *negative-len-shortest*:

$length\ xs = n \implies len\ m\ i\ i\ xs < 0$

$\implies \exists j\ ys. distinct\ (j\ \# \ ys) \wedge len\ m\ j\ j\ ys < 0 \wedge j \in set\ (i\ \# \ xs) \wedge set\ ys \subseteq set\ xs$

proof (*induction n arbitrary: xs i rule: less-induct*)

case (*less n*)

show *?case*

proof (*cases xs*)

case *Nil*

thus *?thesis* **using** *less.prem*s **by** *auto*

next

case (*Cons y ys*)

then have $length\ xs \geq 1$ **by** *auto*

show *?thesis*

proof (*cases i \in set xs*)

assume $i: i \in set\ xs$

then obtain $as\ bs$ **where** $xs = as\ @\ i\ \# \ bs$ **by** (*meson split-list*)

show *?thesis*

proof (*cases len m i i as < 0*)

case *True*

```

    from xs less.premis have length as < n by auto
    from less.IH[OF this - True] xs show ?thesis by auto
next
  case False
  from len-decomp[OF xs] have len m i i xs = len m i i as + len m i
i bs by auto
  with False less.premis have *: len m i i bs < 0
  by (metis add-lt-neutral local.dual-order.strict-trans local.neqE)
  from xs less.premis have length bs < n by auto
  from less.IH[OF this - *] xs show ?thesis by auto
qed
next
  assume i: i ∉ set xs
  show ?thesis
  proof (cases distinct xs)
    case True
    with i less.premis show ?thesis by auto
  next
    case False
    from not-distinct-decomp[OF this] obtain a as bs cs where xs:
      xs = as @ a # bs @ a # cs
    by auto
    show ?thesis
    proof (cases len m a a bs < 0)
      case True
      from xs less.premis have length bs < n by auto
      from less.IH[OF this - True] xs show ?thesis by auto
    next
      case False
      from len-decomp[OF xs, of m i i] len-decomp[of bs @ a # cs bs a
cs m a i]
      have *: len m i i xs = len m i a as + (len m a a bs + len m a i cs)
    by auto
    from False have len m a a bs ≥ 0 by auto
    with add-mono have len m a a bs + len m a i cs ≥ len m a i cs
  by fastforce
  with * have len m i i xs ≥ len m i a as + len m a i cs by (simp
add: add-mono)
  with less.premis(2) have len m i a as + len m a i cs < 0 by auto
  with len-comp have len m i i (as @ a # cs) < 0 by auto
  from less.IH[OF - - this, of length (as @ a # cs)] xs less.premis
  show ?thesis by auto
qed
qed

```

qed
 qed
 qed

lemma *fw-upd-leI*:

fw-upd m' k i j i j ≤ fw-upd m k i j i j **if**
m' i k ≤ m i k m' k j ≤ m k j m' i j ≤ m i j
using that **unfolding** *fw-upd-def upd-def min-def* **using** *add-mono* **by**
fastforce

lemma *fwi-fw-upd-mono*:

fwi m n k i j i j ≤ fw-upd m k i j i j **if** $k ≤ n$ $i ≤ n$ $j ≤ n$
using that **by** (*cases i; cases j*) (*auto intro: fw-upd-leI fwi-mono*)

The Floyd-Warshall algorithm will always detect negative cycles. The argument goes as follows: In case there is a negative cycle, then we know that there is some smallest k for which there is a negative cycle containing only intermediate vertices from the set $\{0..k\}$. We will show that then *fwi m n k* computes a negative entry on the diagonal, and thus, by monotonicity, *fw m n n* will compute a negative entry on the diagonal.

theorem *FW-neg-cycle-detect*:

\neg *cyc-free m n* $\implies \exists i \leq n. fw\ m\ n\ n\ i\ i < 0$

proof –

assume *A*: \neg *cyc-free m n*
let $?K = \{k. k \leq n \wedge \neg\ cyc\text{-free}\text{-subs}\ n\ \{0..k\}\ m\}$
define k **where** $k = Min\ ?K$
have *not-empty-K*: $?K \neq \{\}$ **using** *A* **by** *auto*
have *finite ?K* **by** *auto*
with *not-empty-K* **have** $*$:
 $\forall k' < k. cyc\text{-free}\text{-subs}\ n\ \{0..k'\}\ m$
by (*auto simp add: k-def not-le*)
 (*meson less-imp-le-nat local.leI order-less-irrefl preorder-class.order-trans*)
from *linorder-class.Min-in[OF <finite ?K> <?K ≠ {}>]* **have**
 $\neg\ cyc\text{-free}\text{-subs}\ n\ \{0..k\}\ m\ k \leq n$
unfolding *k-def* **by** *auto*
then **have** $\exists\ xs\ j. j \leq n \wedge len\ m\ j\ j\ xs < 0 \wedge set\ xs \subseteq \{0..k\}$
by *force*
then **obtain** *a-as* **where** $a\text{-as}: a \leq n \wedge len\ m\ a\ a\ as < 0 \wedge set\ as \subseteq \{0..k\}$ **by** *auto*
with *negative-len-shortest[of as length as m a]* **obtain** $j\ xs$ **where** $j\text{-xs}$:
 $distinct\ (j\ \#\ xs) \wedge len\ m\ j\ j\ xs < 0 \wedge j \in set\ (a\ \#\ as) \wedge set\ xs \subseteq set\ as$
by *auto*
with $a\text{-as}\ \langle k \leq n \rangle$ **have** *cyc*: $j \leq n\ set\ xs \subseteq \{0..k\}\ len\ m\ j\ j\ xs < 0$
 $distinct\ (j\ \#\ xs)$

```

by auto
{ assume  $k > 0$ 
  then have  $k - 1 < k$  by simp
  with * have **:cyc-free-sub $s$   $n \{0..k - 1\} m$  by blast
  have  $k \in \text{set } xs$ 
  proof (rule ccontr, goal-cases)
    case 1
      with  $\langle \text{set } xs \subseteq \{0..k\} \rangle$  nat-upto-sub $s$ -top-removal have  $\text{set } xs \subseteq \{0..k-1\}$  by auto
      with  $\langle \text{cyc-free-sub $s$  } n \{0..k - 1\} m \rangle \langle j \leq n \rangle$  have  $0 \leq \text{len } m \ j \ j \ xs$ 
by blast
    with cyc(3) show ?case by simp
  qed
  with cyc(4) have  $j \neq k$  by auto
  from  $\langle k \in \text{set } xs \rangle$  obtain  $ys \ zs$  where  $xs = ys @ k \# zs$  by (meson split-list)
  with  $\langle \text{distinct } (j \# xs) \rangle$ 
  have  $xs: xs = ys @ k \# zs$  distinct  $ys$  distinct  $zs$   $k \notin \text{set } ys$   $k \notin \text{set } zs$ 
     $j \notin \text{set } ys$   $j \notin \text{set } zs$  by auto
  from  $xs(1,4)$   $\langle \text{set } xs \subseteq \{0..k\} \rangle$  nat-upto-sub $s$ -top-removal have  $ys: \text{set } ys \subseteq \{0..k-1\}$  by auto
  from  $xs(1,5)$   $\langle \text{set } xs \subseteq \{0..k\} \rangle$  nat-upto-sub $s$ -top-removal have  $zs: \text{set } zs \subseteq \{0..k-1\}$  by auto
  have  $D \ m \ j \ k \ (k - 1) = \text{fw } m \ n \ (k - 1) \ j \ k$ 
    using  $\langle k \leq n \rangle \langle j \leq n \rangle$  fw-shortest-path-up-to[OF **] by auto
  moreover have  $D \ m \ k \ j \ (k - 1) = \text{fw } m \ n \ (k - 1) \ k \ j$ 
    using  $\langle k \leq n \rangle \langle j \leq n \rangle$  fw-shortest-path-up-to[OF **] by auto
  ultimately have  $\text{fw } m \ n \ (k - 1) \ j \ k + \text{fw } m \ n \ (k - 1) \ k \ j \leq \text{len } m \ j$ 
 $k \ ys + \text{len } m \ k \ j \ zs$ 
    using  $D$ -not-diag-le'[OF  $zs(1)$   $xs(5,7,3)$ , of  $m$ ]  $D$ -not-diag-le'[OF  $ys(1)$ 
 $xs(6,4,2)$ , of  $m$ ]
    by (auto simp: add-mono)
  then have  $\text{neg}: \text{fw } m \ n \ (k - 1) \ j \ k + \text{fw } m \ n \ (k - 1) \ k \ j < 0$ 
    using  $xs(1)$   $\langle \text{len } m \ j \ j \ xs < 0 \rangle$  len-comp by auto
  have  $\text{fw } m \ n \ k \ j \ j \leq \text{fw } m \ n \ (k - 1) \ j \ k + \text{fw } m \ n \ (k - 1) \ k \ j$ 
  proof -
    from  $\langle k > 0 \rangle$  have *:  $\text{fw } m \ n \ k = \text{fwi } (\text{fw } m \ n \ (k - 1)) \ n \ k \ n \ n$ 
      by (cases  $k$ ) auto
    from fw-cyc-free-sub $s$ [OF **, THEN cyc-free-sub $s$ -diag]  $\langle k \leq n \rangle$  have
       $\text{fw } m \ n \ (k - 1) \ k \ k \geq 0$ 
      by auto
    from fwi-step'[of  $\text{fw } m \ n \ (k - 1)$ , OF this]  $\langle k \leq n \rangle \langle j \leq n \rangle$  show
?thesis
      by (auto intro: min.cobounded2 simp: *)

```

```

qed
with neg have fw m n k j j < 0 by auto
moreover from fw-invariant  $\langle j \leq n \rangle \langle k \leq n \rangle$  have fw m n n j j ≤ fw
m n k j j
  by blast
ultimately have ?thesis using  $\langle j \leq n \rangle$  by auto
}
moreover
{ assume k = 0
with cyc(2,4) have xs = [] ∨ xs = [0]
  apply safe
  apply (case-tac xs)
  apply fastforce
  apply (rename-tac ys)
  apply (case-tac ys)
  apply auto
done
then have ?thesis
proof
  assume xs = []
  with cyc have m j j < 0 by auto
  with fw-mono[of j n j m n]  $\langle j \leq n \rangle$  have fw m n n j j < 0 by auto
  with  $\langle j \leq n \rangle$  show ?thesis by auto
next
  assume xs: xs = [0]
  with cyc have m j 0 + m 0 j < 0 by auto
  moreover from  $\langle j \leq n \rangle$  have fw m n 0 j j ≤ fw-upd m 0 j j j j
    by (auto intro: order.trans[OF fwi-invariant fwi-fw-upd-mono])
  ultimately have fw m n 0 j j < 0
    unfolding fw-upd-def upd-def by auto
    then have fw m n 0 j j < 0 by (metis cyc(1) less-or-eq-imp-le
single-iteration-inv)
    with  $\langle j \leq n \rangle$  have fw m n n j j < 0 using fw-invariant[of 0 n j n j
m] by auto
    with  $\langle j \leq n \rangle$  show ?thesis by blast
  qed
}
ultimately show ?thesis by auto
qed
end

```

1.8 More on Canonical Matrices

abbreviation

$\text{canonical } M\ n \equiv \forall\ i\ j\ k. i \leq n \wedge j \leq n \wedge k \leq n \longrightarrow M\ i\ k \leq M\ i\ j + M\ j\ k$

lemma *canonical-alt-def*:

$\text{canonical } M\ n \longleftrightarrow \text{canonical-subs } n\ \{0..n\}\ M$

unfolding *canonical-subs-def* **by** *auto*

lemma *fw-canonical*:

$\text{canonical } (\text{fw } m\ n\ n)\ n$ **if** *cyc-free* $m\ n$

using *fw-canonical-subs*[*OF* $\langle \text{cyc-free } m\ n \rangle$] **unfolding** *canonical-alt-def* **by** *auto*

lemma *canonical-len*:

$\text{canonical } M\ n \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow \text{set } xs \subseteq \{0..n\} \Longrightarrow M\ i\ j \leq \text{len } M\ i\ j\ xs$

proof (*induction* xs *arbitrary*: i)

case *Nil* **thus** *?case* **by** *auto*

next

case (*Cons* $x\ xs$)

then have $M\ x\ j \leq \text{len } M\ x\ j\ xs$ **by** *auto*

from *Cons.prem*s $\langle \text{canonical } M\ n \rangle$ **have** $M\ i\ j \leq M\ i\ x + M\ x\ j$ **by** *simp*

also with *Cons* **have** $\dots \leq M\ i\ x + \text{len } M\ x\ j\ xs$ **by** (*simp* *add*: *add-mono*)

finally show *?case* **by** *simp*

qed

1.9 Additional Theorems

lemma *D-cycle-free-len-dest*:

cycle-free $m\ n$

$\Longrightarrow \forall\ i \leq n. \forall\ j \leq n. D\ m\ i\ j\ n = m'\ i\ j \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow \text{set } xs \subseteq \{0..n\}$

$\Longrightarrow \exists\ ys. \text{set } ys \subseteq \{0..n\} \wedge \text{len } m'\ i\ j\ xs = \text{len } m\ i\ j\ ys$

proof (*induction* xs *arbitrary*: i)

case *Nil*

with *Nil* **have** $m'\ i\ j = D\ m\ i\ j\ n$ **by** *simp*

from *D-dest'*[*OF* *this*]

obtain ys **where** $\text{set } ys \subseteq \{0..n\}$ $\text{len } m'\ i\ j\ [] = \text{len } m\ i\ j\ ys$

by *auto*

then show *?case* **by** *auto*

next

case (*Cons* $y\ ys$)

from *Cons.IH*[*OF Cons.prem*s(1,2) - $\langle j \leq n \rangle$, *of y*] *Cons.prem*s(5)
obtain *zs* **where** $zs:set\ zs \subseteq \{0..n\}$ $len\ m'\ y\ j\ ys = len\ m\ y\ j\ zs$ **by** *auto*
with *Cons* **have** $m'\ i\ y = D\ m\ i\ y\ n$ **by** *simp*
from *D-dest'*[*OF this*] **obtain** *ws* **where** $ws:set\ ws \subseteq \{0..n\}$ $m'\ i\ y =$
 $len\ m\ i\ y\ ws$ **by** *auto*
with *len-comp*[*of m i j ws y zs*] *zs* *Cons.prem*s(5)
have $len\ m'\ i\ j\ (y\ \#\ ys) = len\ m\ i\ j\ (ws\ @\ y\ \#\ zs)$ $set\ (ws\ @\ y\ \#\ zs) \subseteq$
 $\{0..n\}$ **by** *auto*
then *show* *?case* **by** *blast*
qed

lemma *D-cyc-free-preservation*:

$cyc-free\ m\ n \implies \forall\ i \leq n. \forall\ j \leq n. D\ m\ i\ j\ n = m'\ i\ j \implies cyc-free\ m'\ n$
proof (*auto*, *goal-cases*)
case (*1 i xs*)
from *D-cycle-free-len-dest*[*OF - 1(2,3,3,4)*] *1(1)* *cycle-free-diag-equiv*
obtain *ys* **where** $set\ ys \subseteq \{0..n\} \wedge len\ m'\ i\ i\ xs = len\ m\ i\ i\ ys$ **by** *fast*
with *1(1,3)* **show** *?case* **by** *auto*
qed

abbreviation $FW\ m\ n \equiv fw\ m\ n\ n$

lemma *FW-out-of-bounds1*:

assumes $i > n$
shows $(FW\ M\ n)\ i\ j = M\ i\ j$
using *assms* **by** (*rule fw-out-of-bounds1*)

lemma *FW-out-of-bounds2*:

assumes $j > n$
shows $(FW\ M\ n)\ i\ j = M\ i\ j$
using *assms* **by** (*rule fw-out-of-bounds2*)

lemma *FW-cyc-free-preservation*:

$cyc-free\ m\ n \implies cyc-free\ (FW\ m\ n)\ n$
apply (*rule D-cyc-free-preservation*)
apply *assumption*
apply *safe*
apply (*rule fw-shortest-path*)
using *cycle-free-diag-equiv* **by** *auto*

lemma *cyc-free-diag-dest'*:

$cyc-free\ m\ n \implies i \leq n \implies m\ i\ i \geq 0$
by (*rule cyc-free-subst-diag*)

lemma *FW-diag-neutral-preservation:*

$\forall i \leq n. M i i = 0 \implies \text{cyc-free } M n \implies \forall i \leq n. (FW M n) i i = 0$

proof (*auto, goal-cases*)

case (*1 i*)

from *this(3)* **have** $(FW M n) i i \leq M i i$ **by** (*auto intro: fw-mono*)

with *1* **have** $(FW M n) i i \leq 0$ **by** *auto*

with *cyc-free-diag-dest'[OF FW-cyc-free-preservation[OF 1(2)] <i ≤ n>]*

show $FW M n i i = 0$

by *auto*

qed

lemma *FW-fixed-preservation:*

fixes $M :: ('a::\text{linordered-ab-monoid-add}) \text{ mat}$

assumes $A: i \leq n \ M \ 0 \ i + M \ i \ 0 = 0$ *canonical* $(FW M n) \ n$ *cyc-free*
 $(FW M n) \ n$

shows $FW M n \ 0 \ i + FW M n \ i \ 0 = 0$ **using** *assms*

proof –

let $?M' = FW M n$

assume $A: i \leq n \ M \ 0 \ i + M \ i \ 0 = 0$ *canonical* $?M' \ n$ *cyc-free* $?M' \ n$

from $\langle i \leq n \rangle$ **have** $?M' \ 0 \ i + ?M' \ i \ 0 \leq M \ 0 \ i + M \ i \ 0$ **by** (*auto intro: fw-mono add-mono*)

with $A(2)$ **have** $?M' \ 0 \ i + ?M' \ i \ 0 \leq 0$ **by** *auto*

moreover from $\langle \text{canonical } ?M' \ n \rangle \langle i \leq n \rangle$

have $?M' \ 0 \ 0 \leq ?M' \ 0 \ i + ?M' \ i \ 0$ **by** *auto*

moreover from *cyc-free-diag-dest'[OF <cyc-free ?M' n>]* **have** $0 \leq ?M' \ 0 \ 0$ **by** *simp*

ultimately show $?M' \ 0 \ i + ?M' \ i \ 0 = 0$ **by** (*auto simp: add-mono*)

qed

lemma *diag-cyc-free-neutral:*

$\text{cyc-free } M n \implies \forall k \leq n. M \ k \ k \leq 0 \implies \forall i \leq n. M \ i \ i = 0$

proof (*clarify, goal-cases*)

case (*1 i*)

note $A = \text{this}$

then have $i \leq n \wedge \text{set } [] \subseteq \{0..n\}$ **by** *auto*

with $A(1)$ **have** $0 \leq M \ i \ i$ **by** *fastforce*

with $A(2) \langle i \leq n \rangle$ **show** $M \ i \ i = 0$ **by** *auto*

qed

lemma *fw-upd-canonical-subs-id:*

$\text{canonical-subs } n \ \{k\} \ M \implies i \leq n \implies j \leq n \implies \text{fw-upd } M \ k \ i \ j = M$

proof (*auto simp: fw-upd-def upd-def less-eq[symmetric] min.coboundedI2, goal-cases*)

case *1*

then have $M i j \leq M i k + M k j$ **unfolding** *canonical-subs-def* **by** *auto*
then have $\min (M i j) (M i k + M k j) = M i j$ **by** (*simp split: split-min*)
thus *?case* **by force**
qed

lemma *fw-upd-canonical-id*:

canonical $M n \implies i \leq n \implies j \leq n \implies k \leq n \implies fw\text{-}upd\ M\ k\ i\ j = M$
using *fw-upd-canonical-subs-id*[*of n k M i j*] **unfolding** *canonical-subs-def*
by *auto*

lemma *fwi-canonical-id*:

fwi $M\ n\ k\ i\ j = M$ **if** *canonical-subs n {k}* $M\ i \leq n\ j \leq n\ k \leq n$
using *that*

proof (*induction i arbitrary: j*)

case *0*

then show *?case* **by** (*induction j*) (*auto intro: fw-upd-canonical-subs-id*)

next

case *Suc*

then show *?case* **by** (*induction j*) (*auto intro: fw-upd-canonical-subs-id*)

qed

lemma *fw-canonical-id*:

fw $M\ n\ k = M$ **if** *canonical-subs n {0..k}* $M\ k \leq n$
using *that* **by** (*induction k*) (*auto simp: canonical-subs-def fwi-canonical-id*)

lemmas *FW-canonical-id = fw-canonical-id*[*OF - order.refl, unfolded canonical-alt-def*[*symmetric*]]

definition $FWI\ M\ n\ k \equiv fwi\ M\ n\ k\ n\ n$

The characteristic property of *fwi*.

theorem *fwi-characteristic*:

canonical-subs n (I \cup {k::nat}) ($FWI\ M\ n\ k$) \vee ($\exists\ i \leq n. FWI\ M\ n\ k\ i\ i < 0$) **if**

canonical-subs n I M I \subseteq $\{0..n\}$ $k \leq n$

proof (*cases 0 \leq M k k*)

case *True*

from *fwi-canonical-extend*[*OF that(1,2) this <k \leq n>*] **show** *?thesis* **unfolding** *FWI-def ..*

next

case *False*

with $\langle k \leq n \rangle$ *fwi-mono*[*OF <k \leq n> <k \leq n>, of M k n n*] **show** *?thesis*
unfolding *FWI-def* **by** *fastforce*

qed

end

theory *Recursion-Combinators*
imports *Refine-Imperative-HOL.IICF*
begin

context
begin

private definition *for-comb* **where**
 $for-comb\ f\ a0\ n = nfoldli\ [0..<n + 1]\ (\lambda\ x.\ True)\ (\lambda\ k\ a.\ (f\ a\ k))\ a0$

fun *for-rec* :: ('a \Rightarrow nat \Rightarrow 'a nres) \Rightarrow 'a \Rightarrow nat \Rightarrow 'a nres **where**
 $for-rec\ f\ a\ 0 = f\ a\ 0\ |$
 $for-rec\ f\ a\ (Suc\ n) = for-rec\ f\ a\ n \gg (\lambda\ x.\ f\ x\ (Suc\ n))$

private lemma *for-comb-for-rec*: $for-comb\ f\ a\ n = for-rec\ f\ a\ n$
unfolding *for-comb-def*

proof (*induction f a n rule: for-rec.induct*)
case 1 then show ?case **by** (*auto simp: pw-eq-iff refine-pw-simps*)
next
case IH: (2 a n)
then show ?case **by** (*fastforce simp: nfoldli-append pw-eq-iff refine-pw-simps*)
qed

private definition *for-rec2'* **where**
 $for-rec2'\ f\ a\ n\ i\ j =$
 $(if\ i = 0\ then\ RETURN\ a\ else\ for-rec\ (\lambda\ a.\ i.\ for-rec\ (\lambda\ a.\ f\ a\ i)\ a\ n)\ a$
 $(i - 1))$
 $\gg (\lambda\ a.\ for-rec\ (\lambda\ a.\ f\ a\ i)\ a\ j)$

fun *for-rec2* :: ('a \Rightarrow nat \Rightarrow nat \Rightarrow 'a nres) \Rightarrow 'a \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow
'a nres **where**
 $for-rec2\ f\ a\ n\ 0\ 0 = f\ a\ 0\ 0\ |$
 $for-rec2\ f\ a\ n\ (Suc\ i)\ 0 = for-rec2\ f\ a\ n\ i\ n \gg (\lambda\ a.\ f\ a\ (Suc\ i)\ 0)\ |$
 $for-rec2\ f\ a\ n\ i\ (Suc\ j) = for-rec2\ f\ a\ n\ i\ j \gg (\lambda\ a.\ f\ a\ i\ (Suc\ j))$

private lemma *for-rec2-for-rec2'*:
 $for-rec2\ f\ a\ n\ i\ j = for-rec2'\ f\ a\ n\ i\ j$
unfolding *for-rec2'-def*
apply (*induction f a n i j rule: for-rec2.induct*)
apply *simp-all*
subgoal for $f\ a\ n\ i$

apply (*cases i*)
by *auto*
done

fun *for-rec3* :: ('a ⇒ nat ⇒ nat ⇒ nat ⇒ 'a nres) ⇒ 'a ⇒ nat ⇒ nat ⇒ nat ⇒ nat ⇒ 'a nres

where

for-rec3 f m n 0 0 0 = f m 0 0 0 |
for-rec3 f m n (Suc k) 0 0 = for-rec3 f m n k n n ≫ (λ a. f a (Suc k) 0 0) |
for-rec3 f m n k (Suc i) 0 = for-rec3 f m n k i n ≫ (λ a. f a k (Suc i) 0) |
for-rec3 f m n k i (Suc j) = for-rec3 f m n k i j ≫ (λ a. f a k i (Suc j))

private definition *for-rec3'* **where**

for-rec3' f a n k i j =
(if k = 0 then RETURN a else for-rec (λ a k. for-rec2' (λ a. f a k) a n n n) a (k - 1))
≫ (λ a. for-rec2' (λ a. f a k) a n i j)

private lemma *for-rec3-for-rec3'*:

for-rec3 f a n k i j = for-rec3' f a n k i j

unfolding *for-rec3'-def*

apply (*induction f a n k i j rule: for-rec3.induct*)

apply (*simp-all add: for-rec2-for-rec2'[symmetric]*)

subgoal for *f a n k*

apply (*cases k*)

by *auto*

done

private lemma *for-rec2'-for-rec*:

for-rec2' f a n n n =

for-rec (λ a i. for-rec (λ a. f a i) a n) a n

unfolding *for-rec2'-def* **by** (*cases n*) *auto*

private lemma *for-rec3'-for-rec*:

for-rec3' f a n n n n =

for-rec (λ a k. for-rec (λ a i. for-rec (λ a. f a k i) a n) a n) a n

unfolding *for-rec3'-def for-rec2'-for-rec* **by** (*cases n*) *auto*

theorem *for-rec-eq*:

for-rec f a n = nfoldli [0..<n + 1] (λ x. True) (λ k a. f a k) a

using *for-comb-for-rec[unfolded for-comb-def, symmetric]* .

```

theorem for-rec2-eq:
  for-rec2 f a n n n =
    nfoldli [0..<n + 1] ( $\lambda x. \text{True}$ )
      ( $\lambda i. \text{nfoldli}$  [0..<n + 1] ( $\lambda x. \text{True}$ ) ( $\lambda j a. f a i j$ )) a
using
  [for-rec2'-for-rec
    unfolded for-rec2-for-rec2'[symmetric], unfolded for-comb-for-rec[symmetric]
for-comb-def
  ] .

theorem for-rec3-eq:
  for-rec3 f a n n n n =
    nfoldli [0..<n + 1] ( $\lambda x. \text{True}$ )
      ( $\lambda k. \text{nfoldli}$  [0..<n + 1] ( $\lambda x. \text{True}$ )
        ( $\lambda i. \text{nfoldli}$  [0..<n + 1] ( $\lambda x. \text{True}$ ) ( $\lambda j a. f a k i j$ )))
      a
using
  [for-rec3'-for-rec
    unfolded for-rec3-for-rec3'[symmetric], unfolded for-comb-for-rec[symmetric]
for-comb-def
  ] .

end

lemmas [intf-of-assn] = intf-of-assnI[where R= is-mtx n and 'a'= 'b i-mtx
for n]

declare param-upt[sepref-import-param]

end
theory FW-Code
  imports
    Recursion-Combinators
    Floyd-Warshall
begin

```

1.10 Refinement to Efficient Imperative Code

We will now refine the recursive version of the Floyd-Warshall algorithm to an efficient imperative version. To this end, we use the Sepref framework, yielding an implementation in Imperative HOL.

definition $fw\text{-upd}' :: ('a::\text{linordered-ab-monoid-add})\ \text{mtx} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a\ \text{mtx}\ \text{nres}$ **where**
 $fw\text{-upd}'\ m\ k\ i\ j =$
 $RETURN\ (\$
 $\quad op\text{-mtx-set}\ m\ (i, j)\ (min\ (op\text{-mtx-get}\ m\ (i, j))\ (op\text{-mtx-get}\ m\ (i, k) +$
 $op\text{-mtx-get}\ m\ (k, j)))$
 $\quad)$

lemma $fw\text{-upd}'\text{-alt-def}$:

$fw\text{-upd}'\ m\ k\ i\ j =$
 $RETURN\ (\$
 $\quad let$
 $\quad\quad e = op\text{-mtx-get}\ m\ (i, k) + op\text{-mtx-get}\ m\ (k, j)$
 $\quad\quad in\ if\ e < op\text{-mtx-get}\ m\ (i, j)\ then\ op\text{-mtx-set}\ m\ (i, j)\ e\ else\ m$
 $\quad)$

unfolding $fw\text{-upd}'\text{-def}\ min\text{-def}\ Let\text{-def}$ **by** $auto$

definition $fwi' :: ('a::\text{linordered-ab-monoid-add})\ \text{mtx} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a\ \text{mtx}\ \text{nres}$

where

$fwi'\ m\ n\ k\ i\ j = RECT\ (\lambda\ fw\ (m, k, i, j).$
 $\quad case\ (i, j)\ of$
 $\quad\quad (0, 0) \Rightarrow fw\text{-upd}'\ m\ k\ 0\ 0\ |$
 $\quad\quad (Suc\ i, 0) \Rightarrow do\ \{m' \leftarrow fw\ (m, k, i, n); fw\text{-upd}'\ m'\ k\ (Suc\ i)\ 0\}\ |$
 $\quad\quad (i, Suc\ j) \Rightarrow do\ \{m' \leftarrow fw\ (m, k, i, j); fw\text{-upd}'\ m'\ k\ i\ (Suc\ j)\}$
 $\quad)\ (m, k, i, j)$

lemma $fwi'\text{-simps}$:

$fwi'\ m\ n\ k\ 0\ 0 = fw\text{-upd}'\ m\ k\ 0\ 0$
 $fwi'\ m\ n\ k\ (Suc\ i)\ 0 = do\ \{m' \leftarrow fwi'\ m\ n\ k\ i\ n; fw\text{-upd}'\ m'\ k\ (Suc\ i)\ 0\}$
 $fwi'\ m\ n\ k\ i\ (Suc\ j) = do\ \{m' \leftarrow fwi'\ m\ n\ k\ i\ j; fw\text{-upd}'\ m'\ k\ i\ (Suc\ j)\}$

unfolding $fwi'\text{-def}$ **by** $(subst\ RECT\text{-unfold}, (refine\text{-mono};\ fail), (auto\ split:\ nat.\text{split};\ fail))+$

lemma

$fwi'\ m\ n\ k\ i\ j \leq SPEC\ (\lambda\ r.\ r = uncurry\ (fwi\ (curry\ m)\ n\ k\ i\ j))$

by $(induction\ curry\ m\ n\ k\ i\ j\ arbitrary;\ m\ rule:\ fwi.\text{induct})$

$(fastforce\ simp\ add:\ fw\text{-upd}'\text{-def}\ fw\text{-upd}\text{-def}\ upd\text{-def}\ fwi'\text{-simps}\ pw\text{-le}\text{-iff}\ refine\text{-pw}\text{-simps})+$

lemma $fw\text{-upd}'\text{-spec}$:

$fw\text{-upd}'\ M\ k\ i\ j \leq SPEC\ (\lambda\ M'. M' = uncurry\ (fw\text{-upd}\ (curry\ M)\ k\ i\ j))$

by (*auto simp: fw-upd'-def fw-upd-def upd-def pw-le-iff refine-pw-simps*)

lemma *for-rec2-fwi*:

for-rec2 ($\lambda M. fw\text{-upd}' M k$) $M n i j \leq SPEC (\lambda M'. M' = uncurry (fwi (curry M) n k i j))$

using *fw-upd'-spec*

by (*induction* $\lambda M. fw\text{-upd}' (M :: (nat \times nat \Rightarrow 'a)) k M n i j$ *rule: for-rec2.induct*)

(*fastforce simp: pw-le-iff refine-pw-simps*) $+$

definition *fw'* :: (*'a::linordered-ab-monoid-add*) *mtx* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *'a* *mtx nres* **where**

*fw' m n k = nfoldli [0..*k* + 1] ($\lambda -. True$) ($\lambda k M. for\text{-rec2} (\lambda M. fw\text{-upd}' M k) M n n n$) *m**

lemma *fw'-spec*:

fw' m n k $\leq SPEC (\lambda M'. M' = uncurry (fw (curry m) n k))$

unfolding *fw'-def*

apply (*induction k*)

using *for-rec2-fwi* **by** (*fastforce simp add: pw-le-iff refine-pw-simps curry-def*) $+$

context

fixes *n* :: *nat*

fixes *dummy* :: *'a::{linordered-ab-monoid-add,zero,heap}*

begin

lemma [*sepref-import-param*]: $((+), (+)::'a \Rightarrow -) \in Id \rightarrow Id \rightarrow Id$ **by** *simp*

lemma [*sepref-import-param*]: $(min, min)::'a \Rightarrow -) \in Id \rightarrow Id \rightarrow Id$ **by** *simp*

abbreviation *node-assn* \equiv *nat-assn*

abbreviation *mtx-assn* \equiv *asmtx-assn (Suc n) id-assn::('a mtx \Rightarrow -)*

sepref-definition *fw-upd-impl1* **is**

uncurry2 (*uncurry fw-upd'*) ::

$[\lambda (((-,k),i),j). k \leq n \wedge i \leq n \wedge j \leq n]_a$ *mtx-assn^d* $*$ _{*a*} *node-assn^k* $*$ _{*a*} *node-assn^k* $*$ _{*a*} *node-assn^k*

\rightarrow *mtx-assn*

unfolding *fw-upd'-def* **by** *sepref*

sepref-definition *fw-upd-impl* **is**

uncurry2 (*uncurry fw-upd'*) ::

$[\lambda (((-,k),i),j). k \leq n \wedge i \leq n \wedge j \leq n]_a$ *mtx-assn^d* $*$ _{*a*} *node-assn^k* $*$ _{*a*} *node-assn^k* $*$ _{*a*} *node-assn^k*

\rightarrow *mtx-assn*

unfolding *fw-upd'-alt-def* by *sepref*

sepref-register *fw-upd'* :: 'a i-mtx \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a i-mtx nres

definition

fwi-impl' (*M* :: 'a i-mtx) *k* = *for-rec2* (λ *M*. *fw-upd'* *M* *k*) *M* *n* *n* *n*

definition

fw-impl' (*M* :: 'a i-mtx) = *fw'* *M* *n* *n*

context

notes [*id-rules*] = *itypeI*[of *n* *TYPE* (*nat*)]

and [*sepref-import-param*] = *IdI*[of *n*]

begin

sepref-definition *fw-impl* is

fw-impl' :: *mtx-assn*^{*d*} \rightarrow_a *mtx-assn*

unfolding *fw-impl'-def*[*abs-def*] *fw'-def* *for-rec2-eq*

supply [*sepref-fr-rules*] = *fw-upd-impl.refine*

by *sepref*

sepref-definition *fw-impl1* is

fw-impl' :: *mtx-assn*^{*d*} \rightarrow_a *mtx-assn*

unfolding *fw-impl'-def*[*abs-def*] *fw'-def* *for-rec2-eq*

supply [*sepref-fr-rules*] = *fw-upd-impl1.refine*

by *sepref*

sepref-definition *fwi-impl* is

uncurry *fwi-impl'* :: [λ (*-*,*k*). *k* \leq *n*]_{*a*} *mtx-assn*^{*d*} *_{*a*} *node-assn*^{*k*} \rightarrow *mtx-assn*

unfolding *fwi-impl'-def*[*abs-def*] *for-rec2-eq*

supply [*sepref-fr-rules*] = *fw-upd-impl.refine*

by *sepref*

sepref-definition *fwi-impl1* is

uncurry *fwi-impl'* :: [λ (*-*,*k*). *k* \leq *n*]_{*a*} *mtx-assn*^{*d*} *_{*a*} *node-assn*^{*k*} \rightarrow *mtx-assn*

supply [*sepref-fr-rules*] = *fw-upd-impl1.refine*

unfolding *fwi-impl'-def*[*abs-def*] *for-rec2-eq* **by** *sepref*

end

end

export-code *fw-impl* in *SML-imp*

A compact specification for the characteristic property of the Floyd-Warshall algorithm.

definition *fw-spec* where

fw-spec n $M \equiv SPEC (\lambda M'.$
 if $(\exists i \leq n. M' i i < 0)$
 then $\neg \text{cyc-free } M n$
 else $\forall i \leq n. \forall j \leq n. M' i j = D M i j n \wedge \text{cyc-free } M n)$

lemma *D-diag-nonnegI*:

assumes *cycle-free* $M n i \leq n$
shows $D M i i n \geq 0$
using *assms* *D-dest'*[*OF refl, of M i i n*] **unfolding** *cycle-free-def* **by** *auto*

lemma *fw-fw-spec*:

RETURN (*FW* $M n$) \leq *fw-spec* n M
unfolding *fw-spec-def* *cycle-free-diag-equiv*
proof (*simp, safe, goal-cases*)
 case *prems*: (1 i)
 with *fw-shortest-path*[*unfolded cycle-free-diag-equiv, OF prems(3)*] *D-diag-nonnegI*
show *?case*
 by *fastforce*
next
 case 2 **then show** *?case* **using** *FW-neg-cycle-detect*[*unfolded cycle-free-diag-equiv*]
 by (*force intro: fw-shortest-path*[*symmetric, unfolded cycle-free-diag-equiv*])
next
 case 3 **then show** *?case* **using** *FW-neg-cycle-detect*[*unfolded cycle-free-diag-equiv*]
by *blast*
qed

definition

mat-curry-rel = $\{(Mu, Mc). \text{curry } Mu = Mc\}$

definition

mtx-curry-assn $n = \text{hr-comp } (\text{mtx-assn } n) (\text{br curry } (\lambda -. \text{True}))$

declare *mtx-curry-assn-def*[*symmetric, fcomp-norm-unfold*]

lemma *fw-impl'-correct*:

$(\text{fw-impl}', \text{fw-spec}) \in \text{Id} \rightarrow \text{br curry } (\lambda -. \text{True}) \rightarrow \langle \text{br curry } (\lambda -. \text{True}) \rangle$
nres-rel

unfolding *fw-impl'-def*[*abs-def*] **using** *fw'-spec* *fw-fw-spec*

by (fastforce simp: in-br-conv pw-le-iff refine-pw-simps intro!: nres-rel)

1.10.1 Main Result

This is one way to state that the *fw-impl* fulfills the specification *fw-spec*.

theorem *fw-impl-correct*:

(*fw-impl* *n*, *fw-spec* *n*) ∈ (*mtx-curry-assn* *n*)^d →_a *mtx-curry-assn* *n*
using *fw-impl.refine*[*FCOMP* *fw-impl'-correct*[*THEN* *fun-relD*, *OF* *IdI*]] .

An alternative version: a Hoare triple for total correctness.

corollary

<*mtx-curry-assn* *n* *M* *Mi*> *fw-impl* *n* *Mi* <λ *Mi'*. ∃_A *M'*. *mtx-curry-assn*
n *M'* *Mi'* * ↑

(if (∃ *i* ≤ *n*. *M'* *i* *i* < 0)

then ¬ *cyc-free* *M* *n*

else ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M'* *i* *j* = *D* *M* *i* *j* *n* ∧ *cyc-free* *M* *n*)>_t

unfolding *cycle-free-diag-equiv*

by (*rule* *cons-rule*[*OF* - - *fw-impl-correct*[*THEN* *hfrefD*, *THEN* *hn-refineD*]])

(*sep-auto* *simp*: *fw-spec-def*[*unfolded* *cycle-free-diag-equiv*])+

1.10.2 Alternative versions for Uncurried Matrices.

definition *FWI'* = *uncurry* o o o *FWI* o *curry*

lemma *fwi-impl'-refine-FWI'*:

(*fwi-impl'* *n*, *RETURN* o o *PR-CONST* (λ *M*. *FWI'* *M* *n*)) ∈ *Id* → *Id* →
<*Id*> *nres-rel*

unfolding *fwi-impl'-def*[*abs-def*] *FWI-def*[*abs-def*] *FWI'-def* **using** *for-rec2-fwi*

by (*force* *simp*: *pw-le-iff* *pw-nres-rel-iff* *refine-pw-simps*)

lemmas *fwi-impl-refine-FWI'* = *fwi-impl.refine*[*FCOMP* *fwi-impl'-refine-FWI'*]

definition *FW'* = *uncurry* o o *FW* o *curry*

definition *FW''* *n* *M* = *FW'* *M* *n*

lemma *fw-impl'-refine-FW''*:

(*fw-impl'* *n*, *RETURN* o *PR-CONST* (*FW''* *n*)) ∈ *Id* → <*Id*> *nres-rel*

unfolding *fw-impl'-def*[*abs-def*] *FW''-def*[*abs-def*] *FW'-def* **using** *fw'-spec*

by (*force* *simp*: *pw-le-iff* *pw-nres-rel-iff* *refine-pw-simps*)

lemmas *fw-impl-refine-FW''* = *fw-impl.refine*[*FCOMP* *fw-impl'-refine-FW''*]

lemmas *fw-impl1-refine-FW''* = *fw-impl1.refine*[*FCOMP* *fw-impl'-refine-FW''*]

end

References

- [Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.
- [Roy59] Bernard Roy. Transitivité et connexité. In *Extrait des comptes rendus des séances de l'Académie des Sciences*, pages 216–218. Gauthier-Villars, July 1959. <http://gallica.bnf.fr/ark:/12148/bpt6k3201c/f222.image.langFR>.
- [War62] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962.