The Floyd-Warshall Algorithm for Shortest Paths

Simon Wimmer and Peter Lammich

September 13, 2023

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:

\[
\begin{align*}
\text{for } k &= 1 \ldots n \text{ do} \\
\text{for } i &= 1 \ldots n \text{ do} \\
\text{for } j &= 1 \ldots n \text{ do} \\
&\quad m[i, j] := \min(m[i, j], m[i, k] + m[k, j])
\end{align*}
\]

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 $\{0 \ldots k+1\}$, is: either the shortest path from $i$ to $j$ using intermediate vertices from the set $\{0 \ldots k\}$; 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 $\{0 \ldots k\}$. Our presentation will be slightly more general than the typical textbook version, in that we will factor our 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 \( \text{cnt} \; x \; \text{xs} \equiv \text{length} \; (\lambda \; y. \; x = y) \; \text{xs} \)

fun remove-cycles :: 'a list ⇒ 'a ⇒ 'a list ⇒ 'a list

where
\[
\begin{align*}
\text{remove-cycles} \; [] \; - \; \text{acc} & = \; \text{rev} \; \text{acc} \\
\text{remove-cycles} \; (x\#\text{xs}) \; y \; \text{acc} & = \\
& \quad \text{(if} \; x = y \; \text{then remove-cycles} \; \text{xs} \; y \; [x] \; \text{else remove-cycles} \; \text{xs} \; y \; (x\#\text{acc}))
\end{align*}
\]

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

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

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

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

\begin{align*}
\text{case} \; \text{Nil} \; \text{thus} \; ?\text{case} \\
\text{by} \; (\text{simp}, \; \text{cases} \; x \in \text{set} \; \text{ys}, \; (\text{auto simp: cnt-rev[of} \; x \; \text{ys]}))
\end{align*}

next

\begin{align*}
\text{case} \; (\text{Cons} \; y \; \text{xs}) \\
\text{thus} \; ?\text{case} \\
\text{proof} \; (\text{cases} \; x = y) \\
\text{case} \; \text{True} \\
\quad \text{thus} \; ?\text{thesis using} \; \text{Cons[of} \; [y]] \; \text{True by} \; \text{auto}
\end{align*}

next

\begin{align*}
\text{case} \; \text{False} \\
\quad \text{thus} \; ?\text{thesis using} \; \text{Cons[of} \; y \# \; \text{ys]} \; \text{by} \; \text{auto}
\end{align*}

qed

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

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

lemma remove-cycles-cnt-id:
\( x \neq y \implies \text{cnt} \; y \; (\text{remove-cycles} \; \text{xs} \; x \; \text{ys}) \leq \text{cnt} \; y \; \text{ys} + \text{cnt} \; y \; \text{xs} \)

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

\begin{align*}
\text{case} \; \text{Nil} \; \text{thus} \; ?\text{case} \; (\text{simp add: cnt-rev}) \\
\text{next} \\
\text{case} \; (\text{Cons} \; z \; \text{xs}) \\
\text{thus} \; ?\text{case}
\end{align*}
proof \( (\text{cases } x = z) \)
\begin{align*}
\text{case } \text{True} & \quad \text{thus } \text{thesis using } \text{Cons.IH[of } z \text{ [z]} \text{] Cons.prems by auto} \\
\text{next} & \\
\text{case } \text{False} & \\
& \text{thus } \text{thesis using } \text{Cons.IH[of } x \text{ z } \# \text{ ys] Cons.prems False by auto} \\
& \quad \text{qed} \\
& \text{qed}
\end{align*}

\begin{quote}
\textbf{lemma} \text{remove-cycles-ends-cycle: } \text{remove-cycles xs x ys } \neq \text{ rev ys } @ \text{ xs } \Longrightarrow x \in \text{ set xs} \\
\text{using } \text{remove-cycles-id by fastforce}
\end{quote}

\begin{quote}
\textbf{lemma} \text{remove-cycles-begins-with: } x \in \text{ set xs } \Longrightarrow \exists \ z s. \text{ remove-cycles xs x ys } \hspace{1em} y s = x \# z s \wedge x \notin \text{ set zs} \\
\text{proof (induction xs arbitrary; ys)} \\
\text{case } \text{Nil thus } \text{case by auto} \\
\text{next} \\
\text{case } \text{(Cons y xs)} \\
& \text{thus } \text{case} \\
\text{proof (cases } x = y) \\
& \text{case } \text{True thus } \text{thesis} \\
& \quad \text{proof (cases } x \in \text{ set xs, goal-cases)} \\
& & \text{case } \text{1 with } \text{Cons show } \text{case by auto} \\
& & \text{next} \\
& & \text{case } \text{2 with } \text{remove-cycles-id[of } x \text{ xs [y]} \text{] show } \text{case by auto} \\
& & \text{qed} \\
& \text{next} \\
& \text{case } \text{False} \\
& \quad \text{with } \text{Cons show } \text{thesis by auto} \\
& \text{qed} \\
& \text{qed}
\end{quote}

\begin{quote}
\textbf{lemma} \text{remove-cycles-self:} \\
\hspace{1em} x \in \text{ set xs } \Longrightarrow \text{ remove-cycles (remove-cycles xs x ys) x zs } = \text{ remove-cycles xs x ys} \\
\text{proof} - \\
\hspace{1em} \text{assume } x : x \in \text{ set xs} \\
\hspace{2em} \text{then obtain } \text{us where } \text{us: remove-cycles xs x ys } = x \# \text{ us x } \notin \text{ set us} \\
\hspace{3em} \text{using } \text{remove-cycles-begins-with[of } \text{ xs, of ys] by blast} \\
\hspace{4em} \text{from } \text{remove-cycles-id[of this(2)] have } \text{remove-cycles us x [x] } = x \# \text{ us by auto} \\
\hspace{5em} \text{with } \text{us(1) show } \text{remove-cycles (remove-cycles xs x ys) x zs } = \text{ remove-cycles xs x ys by simp} \\
& \text{qed}
\end{quote}
lemma remove-cycles-one: remove-cycles (as @ x # xs) x ys = remove-cycles (x#xs) x ys 
by (induction as arbitrary: ys) auto

lemma remove-cycles-cycles:

\exists xxs as. as @ concat (map (λxs. x # xs) xxs) @ remove-cycles xs x ys = xs ∧ x \notin set as

if x ∈ 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. x # xs) xxs) @ remove-cycles xs y [y] = xs
using Cons.IH[of [y]] by auto
hence [] @ concat (map (λxs. x # xs) (as#xxs)) @ remove-cycles (y#xs) x ys = y # xs
by (simp add: \langle x = y \rangle)
thus ?thesis by fastforce
next
case 2
hence remove-cycles (y # xs) x ys = y # xs using remove-cycles-id[of x xs [y]] by auto
hence [] @ concat (map (λxs. x # xs) []) @ 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 @ concat (map (λxs. x # xs) xxs) @ remove-cycles xs x (y#ys) = xs x \notin set as
 using Cons.IH[of y # ys] Cons.prems by auto
hence (y # as) @ concat (map (λxs. x # xs) xxs) @ remove-cycles (y#xs) x ys = y # xs
using \langle x \neq y \rangle by auto
thus ?thesis using as(2) \langle x \neq y \rangle by fastforce
qed
fun start-remove :: 'a list ⇒ 'a ⇒ 'a list ⇒ '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 ∈ set xs ⇒ ∃ as bs. xs = as @ x # bs ∧ start-remove xs x ys = rev ys @ as @ remove-cycles 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 = as @ x # 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

lemma start-remove-removes: cnt x (start-remove xs x ys) ≤ Suc (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
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 \(?\) by (simp add: cnt-rev)
next
case (Cons \( z \) \( xs \))
  thus \(?\) by (cases \( x = z \), goal-cases)
  proof
    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

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.prems
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.prems 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 \textit{?case by auto}
qedeultimately show \textit{?case by auto}
qede
tlemma remove-cycles-subso: s
et (remove-cycles xs x ys) \subseteq set xs \cup set ys
by (induction xs arbitrary: ys; auto; fastforce)
lemma start-remove-subs:
et (start-remove xs x ys) \subseteq set xs \cup set ys
using remove-cycles-subso by (induction xs arbitrary: ys; auto; fastforce)
lemma remove-all-cycles-subs:
et (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 set ys. cnt x (remove-all-cycles xs ys) \leq 1 using cnt-remove-all-cycles
by fastforce
hence \forall x \in set (remove-all-cycles xs ys). cnt x (remove-all-cycles xs ys)
\leq 1
using remove-all-cycles-subso by fastforce
thus distinct (remove-all-cycles xs ys) using cnt-distinct-intro by auto
qede
lemma distinct-remove-cycles-inv: distinct (xs @ ys) \implies distinct (remove-cycles xs x ys)
proof (induction xs arbitrary: ys)
case Nil thus \textit{?case by auto}
next
case (Cons y xs)
thus \textit{?case by auto}
qede
definition
remove-all x xs = (if x \in set xs then tl (remove-cycles xs x [[]]) else xs)
definition
remove-all-rev \ x \ xs = (if \ x \ \in\ \set \ \xs \ then \ \rev \ (\tl \ (\remove-cycles \ (\rev \ \xs) \ x \ []) \ else \ \xs)

\textbf{lemma} remove-all-distinct:
\textbf{distinct} \ \xs \ \Rightarrow \ \textbf{distinct} \ (x \ \# \ \textbf{remove-all} \ x \ \xs)
\textbf{proof} \ (\textbf{cases} \ x \ \in\ \set \ \xs, \ \textbf{goal-cases})
  \textbf{case} 1
  \textbf{from} \ \textbf{remove-cycles-begins-with}[\textbf{OF} \ 1(2), \ \textbf{of} \ [] \ \textbf{obtain} \ zs
  \textbf{where} \ \textbf{remove-cycles} \ \xs \ x \ [] = x \ \# \ zs \ x \ \notin\ \set \ zs \ \textbf{by} \ \textbf{auto}
  \textbf{thus} \ ?\textbf{thesis using} \ 1(1) \ \textbf{distinct-remove-cycles-inv}[\textbf{of} \ \xs \ [] \ x] \ \textbf{by} \ (\textbf{simp add: remove-all-def})
\textbf{next}
  \textbf{case} 2 \ \textbf{thus} \ ?\textbf{thesis} \ \textbf{by} \ (\textbf{simp add: remove-all-def})
\textbf{qed}

\textbf{lemma} remove-all-removes:
\ x \ \notin\ \set \ (\textbf{remove-all} \ x \ \xs)
\textbf{by} \ (\textbf{metis} \ \textbf{list.sel}(3) \ \textbf{remove-all-def} \ \textbf{remove-cycles-begins-with})

\textbf{lemma} remove-all-subs:
\ \set \ (\textbf{remove-all} \ x \ \xs) \ \subseteq\ \set \ \xs
\textbf{using} \ \textbf{remove-cycles-subs} \ \textbf{remove-all-def}
\textbf{by} \ (\textbf{metis} \ \textbf{no-types}, \ \textbf{lifting}) \ \textbf{append-Nil2} \ \textbf{list.sel}(2) \ \textbf{list.set-sel}(2) \ \textbf{set-append} \ \textbf{subsetCE} \ \textbf{subsetI})

\textbf{lemma} remove-all-rev-distinct: \ \textbf{distinct} \ \xs \ \Rightarrow \ \textbf{distinct} \ (x \ \# \ \textbf{remove-all-rev} \ x \ \xs)
\textbf{proof} \ (\textbf{cases} \ x \ \in\ \set \ \xs, \ \textbf{goal-cases})
  \textbf{case} 1
  \textbf{then} \ \textbf{have} \ x \ \in\ \set \ (\rev \ \xs) \ \textbf{by} \ \textbf{auto}
  \textbf{from} \ \textbf{remove-cycles-begins-with}[\textbf{OF} \ \textbf{this}, \ \textbf{of} \ [] \ \textbf{obtain} \ zs
  \textbf{where} \ \textbf{remove-cycles} \ (\rev \ \xs) \ x \ [] = x \ \# \ zs \ x \ \notin\ \set \ zs \ \textbf{by} \ \textbf{auto}
  \textbf{thus} \ ?\textbf{thesis using} \ 1(1) \ \textbf{distinct-remove-cycles-inv}[\textbf{of} \ \rev \ \xs \ [] \ x]
    \ \textbf{by} \ (\textbf{simp add: remove-all-rev-def})
\textbf{next}
  \textbf{case} 2 \ \textbf{thus} \ ?\textbf{thesis} \ \textbf{by} \ (\textbf{simp add: remove-all-rev-def})
\textbf{qed}

\textbf{lemma} remove-all-rev-removes: \ x \ \notin\ \set \ (\textbf{remove-all-rev} \ x \ \xs)
\textbf{by} \ (\textbf{metis} \ \textbf{remove-all-def} \ \textbf{remove-all-removes} \ \textbf{remove-all-rev-def} \ \textbf{set-rev})

\textbf{lemma} remove-all-rev-subs: \ \set \ (\textbf{remove-all-rev} \ x \ \xs) \ \subseteq\ \set \ \xs
\textbf{by} \ (\textbf{metis} \ \textbf{remove-all-def} \ \textbf{remove-all-subs} \ \textbf{set-rev} \ \textbf{remove-all-rev-def})
abbreviation rem-cycles i j xs ≡ remove-all i (remove-all-rev j (remove-all-cycles xs xs))

lemma rem-cycles-distinct*: i ≠ j ϑ distinct (i # j # rem-cycles i j xs)
proof –
  assume i ≠ 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 (remove-all-cycles xs xs))
  by simp
  from remove-all-distinct[OF this] have distinct (i # rem-cycles i j xs)
  by simp
  moreover have
    j ∉ set (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 ∉ set (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: set (rem-cycles i j xs) ⊆ set xs
by (meson order-trans remove-all-cycles-subs remove-all-subs remove-all-rev-subset)

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 mat = nat ⇒ nat ⇒ 'c

definition upd :: 'c mat ⇒ nat ⇒ nat ⇒ 'c ⇒ 'c mat
  where
  upd m x y v = m (x := (m x) (y := v))

definition fw-upd :: 'a mat ⇒ nat ⇒ nat ⇒ nat ⇒ 'a mat
  where
  fw-upd m i j v ≡ upd m i j (min (m i j) (m i k + m k j))

Recursive version of the two inner loops.

fun fwi :: 'a mat ⇒ nat ⇒ nat ⇒ nat ⇒ 'a mat
  where
  fwi m n k 0 0 = fw-upd m k 0 0 | fwi m n k (Suc i) 0 = fw-upd (fwi m n k i n) k (Suc i) 0 | fwi m n k i (Suc j) = fw-upd (fwi m n k i j) k i (Suc j)

Recursive version of the full algorithm.

fun fw :: 'a mat ⇒ nat ⇒ nat ⇒ 'a mat
  where
  fw m n 0 = fwi m n 0 n n | fw m n (Suc k) = fwi (fw m n k) n (Suc k) n n

1.3.2 Elementary Properties

lemma fw-upd-mono:
  fw-upd m k i j i' j' ≤ 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 (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 (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 ≤ 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-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 ≤ 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” ≤ j ⇒ fwi m n k i j i’ j’ ≤ 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’ ≤ fwi m n k i j i’ j’
    by (simp add: fw-upd-mono)
    thus ?thesis using Suc False by simp
  qed
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 \[ \text{fwi} m n k (\text{Suc} i) j i' j' \leq \text{fwi} m n k (\text{Suc} i) 0 i' j' \]
    \[ \text{by (rule fwi-invariant-aux-1[of 0]; simp)} \]
    also have \[ \ldots \leq \text{fwi} m n k i n i' j' \text{ by (simp add: fw-upd-mono)} \]
    also have \[ \ldots \leq \text{fwi} m n k i j i' j' \text{ using fwi-invariant-aux-1 False Suc} \]
    by simp
    also have \[ \ldots \leq \text{fwi} m n k i'' j'' i' j' \text{ using fwi-invariant-aux-1 False Suc} \]
    finally show \[ ?thesis \text{ by simp} \]
  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]
      \[ \text{by (cases j' = n)} \text{ (fastforce simp add: fw-upd-def upd-def)} \]
  qed
next
case \((Suc\ j)\) thus ?case by (simp add: fw-upd-def upd-def)
qed

lemma single-iteration-inv:
i' \leq i \implies j' \leq j \implies j \leq n \implies fwi\ m\ n\ k\ i\ j\ i'\ j' = 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

lemma fwi-innermost-id:
i' < i \implies 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: fw-upd-def upd-def)
next
case (Suc \ j') thus \ ?case by (auto simp: fw-upd-def upd-def)  
qed
qed

lemma \ _fwi-middle-id:
\ j' < \ j \ \Longrightarrow\ \ i' \leq \ i \ \Longrightarrow\ \ _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: fw-upd-def upd-def)  
qed
qed

lemma \ _fwi-outermost-mono:
\ i \leq \ n \ \Longrightarrow\ \ j \leq \ n \ \Longrightarrow\ \ _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 \ j = 0 \ by (simp add: fw-upd-def upd-def)
next
case (Suc \ i')
  hence \ _fwi\ m\ n\ k\ i'\ n\ (Suc \ i')\ 0 = m\ (Suc \ i')\ 0 \ using \ _fwi-innermost-id\ \ i \leq \ n \ by \ simp
  thus \ ?thesis using \ j = 0 \ Suc \ by (simp add: fw-upd-def upd-def)
qed

next
case (Suc j')
  assume i ≤ n j ≤ 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 ≤ m i j if i ≤ n j ≤ 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 ¬ i' < i 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

lemma Suc-innermost-mono:
i ≤ n ⟹ j ≤ n ⟹ fw m n (Suc k) i j ≤ fw m n k i j

17
by (simp add: fwi-mono)

lemma fw-mono:
  \(i \leq n \Rightarrow j \leq n \Rightarrow fwi m n k i j \leq m i j\)
proof (induction \(k\))
  case 0 thus \(?case using fwi-mono by simp\)
  next
  case (Suc \(k\)) thus \(?case using Suc-innermost-mono[OF Suc.prems, 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:
  \(m k k \geq 0 \Rightarrow i \leq n \Rightarrow j \leq n \Rightarrow k \leq n \Rightarrow fwi m n k i j i j = \min (m i j) (m i k + m k j)\)
proof (induction \(- (i, j)\) arbitrary: i j rule: wf-induct[of less-than <\(\ast\)lex\(\ast\)> less-than],
  \(\text{auto; fail}, \text{goal-cases}\))
  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
proof \(?case\)
  case \(\text{(simp \(i'\))}\)
  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)
ultimately show ?thesis using Suc assms by (simp add: fw-upd-def upd-def)

qed
next
  case False
  with \( k \leq i \) show \( ?thesis \)
    by (subst single-iteration-inv; simp add: diag)
qed

have ***: \( \text{fwi} \ m \ n \ k \ (\text{Suc} \ i) \ j \ k \ (\text{Suc} \ j) = m \ k \ (\text{Suc} \ j) \)
proof (cases Suc \( i \) \( \leq \) \( k \))
  case True
  then show \( ?thesis \) by simp
next
  case False
  then have \( \text{fwi} \ m \ n \ k \ k \ j \ k \ (\text{Suc} \ j) = m \ k \ (\text{Suc} \ j) \)
    by simp
  with False \( \langle m \ k \ k \ \geq \ 0 \rangle \) show \( ?thesis \)
    by (subst single-iteration-inv'; simp add: simps diag2)
qed

have \( \text{fwi} \ m \ n \ k \ (\text{Suc} \ i) \ j \ (\text{Suc} \ i) \ k = m \ (\text{Suc} \ i) \ k \)
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 \( \text{prems} \)
proof –
  have \( \text{fwi} \ m \ n \ 0 \ i \ n \ 0 \ 0 \ \geq \ 0 \)
    using ** \( \text{assms}(1) \) \( \text{prems}(2) \) by force
  moreover have \( \text{fwi} \ m \ n \ 0 \ i \ n \ (\text{Suc} \ i) \ 0 = m \ (\text{Suc} \ i) \ 0 \)
    by simp
  ultimately show \( ?thesis \)
    using \( \text{prems} \) by (simp add: simps)
qed
subgoal premises \( \text{prems \ for \} k' \)
proof –
  have \( \text{fwi} \ m \ n \ (\text{Suc} \ k') \ (\text{Suc} \ i) \ k' \ (\text{Suc} \ k') \ (\text{Suc} \ k') \ \geq \ 0 \)
    by (metis ** \( \text{assms}(1,4) \) \( fwi-\text{innermost-id} \) \( fwi-\text{middle-id} \) \( \text{le-SucE} \) \( \text{lessI} \)
linorder-class.\( \text{not-\text{le-imp-less}} \) \( \text{prems}(2) \) preorder-class.\( \text{order-refl} \)
single-iteration-inv single-iteration-inv')
    with \( \text{prems} \) 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 \(j' = \neg\) by \(\text{(simp add: simps \(*\))}\)

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** \(\text{len} :: \forall a. \text{mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow a\) **where**

\[
\text{len} \ m \ u \ v \ [] = m \ u \ v \\
\text{len} \ m \ u \ v \ (\text{w} \# \text{ws}) = m \ u \ w + \text{len} \ m \ w \ v \ \text{ws}
\]

**lemma** \(\text{len-decomp}: \text{xs} = \text{ys} @ \text{y} \# \text{zs} \Longrightarrow \text{len} \ m \ x \ z \ \text{xs} = \text{len} \ m \ x \ y \ \text{ys} + \text{len} \ m \ y \ z \ \text{zs}\)

by \(\text{(induction ys arbitrary: x xs) (simp add: add.assoc)}\)\(+\)

**lemma** \(\text{len-comp}: \text{len} \ m \ a \ c \ (\text{xs} @ \text{b} \# \text{ys}) = \text{len} \ m \ a \ b \ \text{xs} + \text{len} \ m \ b \ c \ \text{ys}\)

by \(\text{(induction xs arbitrary: a) (auto simp: add.assoc)}\)

1.4.2 Canonicality

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 :: \(\text{nat} \Rightarrow \text{nat set} \Rightarrow a\) **where**

\[
\text{canonical-subs} \ n \ I \ m = \forall i j k. \ i \leq n \land k \leq n \land j \in I \longrightarrow \text{i \ k \leq m \ i \ j + m \ j \ k}
\]

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 :: \(\text{nat} \Rightarrow \text{nat set} \Rightarrow a\) **where**

\[
\text{cyc-free-subs} \ n \ I \ m = \forall i \ \text{xs}. \ i \leq n \land \text{set} \ \text{xs} \subseteq I \longrightarrow \text{len} \ m \ i \ i \ \text{xs} \geq 0
\]

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 canonicality 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-sub-len:
\[
M \ i \ j \leq \text{len} \ M \ i \ j \ \text{xs if} \ \text{canonical-sub} \ n \ I \ i \leq n \ j \leq n \ \text{set} \ xs \subseteq I \ I \subseteq \{0..n\}
\]

**Using** that

**Proof** (induction \( xs \) arbitrary; \( i \))

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

**Next**

**Case** (Cons \( x \) \( xs \))

**Then have** \( M \ x \ j \leq \text{len} \ M \ x \ j \ \text{xs by} \ \text{auto} \)

**From** Cons.prems (canonical-sub \( n \ I \ M \) **have** \( M \ i \ j \leq M \ i \ x + M \ x \ j \)

**Unfolding** canonical-sub-def **by** auto

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

**Finally show** ?case **by** simp

**Qed**

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

**Lemma** \( fwi\text{-step}' \):
\[
\begin{align*}
&fwi \ m \ n \ k \ i' \ j' \ i \ j = \min \ (m \ i \ j) \ (m \ i \ k + m \ k \ j)
\text{if} \\
&m \ k \ k \geq 0 \ i' \leq n \ j' \leq n \ k \leq n \ i \leq i' \ j \leq j'
\end{align*}
\]

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

An invocation of \( fwi \) extends canonical forms.

**Lemma** \( fwi\text{-canonical-extend} \):
\[
\begin{align*}
&\text{canonical-sub} \ n \ (I \ \cup \ \{k\}) \ (fwi \ m \ n \ k \ n \ n) \ \text{if} \\
&\text{canonical-sub} \ n \ I \ m \ I \ \subseteq \{0..n\} \ 0 \leq m \ k \ k \ \leq n
\end{align*}
\]

**Using** that

**Unfolding** canonical-sub-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 ≤ 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 ≤ 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' ≤ 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' ≤ 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 ≤ 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: order.trans)
next
case prems: 7
from prems have 0 ≤ 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: order.trans)
An invocation of \textit{fwi} will not produce a negative diagonal entry if there is no negative cycle.

\textbf{lemma} \textit{fwi-cyc-free-diag}:
$$\text{fwi} \ m \ n \ k \ n \ i \ i \geq 0 \text{ if } $$
\textit{cyc-free-subs} \( n \ I \ m \ 0 \leq m \ k \ k \ k \leq n \ k \in I \ i \leq n \)
\textbf{using} that
\textbf{apply} (\textit{subst} \textit{fwi-step'}, (\textit{auto; fail}+)+)
\textbf{unfolding} \textit{min-def} \textbf{by} (\textit{auto intro: add-increasing add-increasing2})
\textbf{proof} (\textit{clarsimp; safe, goal-cases})
\textbf{case} 1
\quad \textbf{have} set [] \subseteq I \\
\quad \textbf{by simp}
\quad \textbf{with} 1(1) \langle i \leq n \rangle \textbf{show} ?\textit{case}
\quad \textbf{by fastforce}
\textbf{next}
\textbf{case} 2
\quad \textbf{then} \textbf{have} set [k] \subseteq I \\
\quad \textbf{by simp}
\quad \textbf{with} 2(1) \langle i \leq n \rangle \textbf{show} ?\textit{case by fastforce}
\textbf{qed}

\textbf{lemma} \textit{cyc-free-sub-diag}:
$$m \ i \ i \geq 0 \text{ if } \textit{cyc-free-subs} \ n \ I \ m \ i \leq n$$
\textbf{proof} –
\quad \textbf{have} set [] \subseteq I \textbf{by auto}
\quad \textbf{with} that \textbf{show} ?\textit{thesis by fastforce}
\textbf{qed}

\textbf{lemma} \textit{fwi-cyc-free-subs}!
$$\textit{cyc-free-subs} \ n \ (I \cup \{k\}) \ (fwi \ m \ n \ k \ n \ n) \textbf{ if } $$
\textit{cyc-free-subs} \ n \ I \ m \ \textit{canonical-subs} \ n \ I \ m \ I \subseteq \{0..n\} \ k \leq n
\quad \forall \ i \leq n. \ fwi \ m \ n \ k \ n \ i \ i \geq 0$$
\textbf{proof} (\textit{safe, goal-cases})
\textbf{case} \textit{prems; (1 i xs)}
\textbf{from} that(1) \langle k \leq n \rangle \textbf{have} 0 \leq m \ k \ k \textbf{ by} (\textit{rule \textit{cyc-free-subs-diag}})
\textbf{from} that \langle 0 \leq m \ k \ k \rangle \textbf{have} *: \textbf{canonical-subs} \ n \ (I \cup \{k\}) \ (fwi \ m \ n \ k \ n \ n)
\textbf{by} – (\textit{rule \textit{fwi-canonical-extend}; auto)
lemma fwi-cyc-free-diag:
  \exists i \leq n. \exists xs. set xs \subseteq \{0..k\} \land 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 \subseteq \{0..k\} \leq \} by auto
  with \(i \leq n\) show ?case by fastforce
next
  case 2
then have \(\text{len } m \ i \ i \ [k] < 0\) set \(\{k\} \subseteq \{0..k\}\) by auto
with \(i \leq n\) show ?case by fastforce
ded
next
case False
with \(k \leq n\) have \(\text{len } m \ k \ k \ [] < 0\) set \(\[] \subseteq \{\}\) by auto
with \(k \leq n\) show ?thesis by fastforce
ded

\(\text{fwi}\) preserves the length of paths.

lemma fwi-len:
\[\exists \ ys. \set \ ys \subseteq \set \ xs \cup \{k\} \land \text{len } (\text{fwi } m \ n \ k \ n \ n) \ i \ j \ xs = \text{len } m \ i \ j \ ys\]
if \(i \leq n \ j \leq n \ k \leq n \ m \ k \geq 0\) set \(\xs \subseteq \{0..n\}\)
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 \(\set \ ys \subseteq \set \ xs \cup \{k\}\) len \(\text{fwi } m \ n \ k \ n \ n\) \(x \ j \ xs = \text{len } m \ x \ j \ ys\)
by force
with Cons.prems 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])
ded

lemma fwi-neg-cycle-neg-cycle:
\[\exists i \leq n. \exists \ ys. \set \ ys \subseteq \set \ xs \cup \{k\} \land \text{len } m \ i \ i \ ys < 0\) if
\(\text{len } (\text{fwi } m \ n \ k \ n \ n) \ i \ i \ xs < 0\) \(i \leq n \ k \leq n\) set \(\xs \subseteq \{0..n\}\)
proof (cases \(m \ k \ k \geq 0\))
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
If the Floyd-Warshall algorithm produces a negative diagonal entry, then there is a negative cycle.

**Lemma fw-neg-diag-neg-cycle:**

\[ \exists i \leq n. \exists ys. \text{set } ys \subseteq \text{set } xs \cup \{0..k\} \land \text{len } m \ i \ i \ ys < 0 \]

**Proof** (induction \( k \) arbitrary: \( i \) \( xs \))

1. **Case 0**
   - Then show \( ?\) case by simp (drule fwi-neg-cycle-neg-cycle; auto)

2. **Case (Suc \( k \))**
   - From fwi-neg-cycle-neg-cycle[\( OF \ Suc.prems(1)[simplified]\] Suc.prems
   - Obtain \( i' \) \( ys \) where
     \[ i' \leq n \ \text{set } ys \subseteq \text{set } xs \cup \{\text{Suc } k\} \land \text{len } m \ i' \ i' \ ys < 0 \]
     - By auto
     - With Suc.prems obtain \( i'' \) \( zs \) where
       \[ i'' \leq n \ \text{set } zs \subseteq \text{set } ys \cup \{0..k\} \land \text{len } m \ i'' \ i'' \ zs < 0 \]
       - By atomize-elim (auto intro! Suc.IH)
       - With \( \text{set } ys \subseteq \to \) have \( \text{set } zs \subseteq \text{set } xs \cup \{0..\text{Suc } k\} \land \text{len } m \ i'' \ i'' \ zs < 0 \)
         - By force
         - With \( \langle i'' \leq n \rangle \) show \( ?\) case by blast

**Qed**

Main theorem under the absence of negative cycles.

**Theorem fw-correct:**

\[ \text{canonical-subs } n \ \{0..k\} \ (fw \ m \ n \ k) \land \text{cyc-free-subs } n \ \{0..k\} \ (fw \ m \ n \ k) \]

If cyc-free-subs \( n \ \{0..k\} \ m \ k \leq n \)

**Proof** (induction \( k \))

1. **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)

2. **Case (Suc \( k \))**
   - Then have IH:
     \[ \text{canonical-subs } n \ \{0..k\} \ (fw \ m \ n \ k) \land \text{cyc-free-subs } n \ \{0..k\} \ (fw \ m \ n \ k) \]
by fastforce
have *: {0..Suc k} = {0..k} ∪ {Suc k} by auto
then have **: canonical-sub 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.prems by (auto intro: cyc-free-sub-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.prems show ?thesis by fastforce
next
  case False
  have cyc-free-sub n {0..Suc k} (fw m n (Suc k))
    apply (simp add: *)
    apply (rule fwi-cyc-free-sub[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-sub-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 mat ⇒ nat ⇒ nat ⇒ nat ⇒ 'a where
  D m i j k ≡ Min {len m i j xs | xs. set xs ⊆ {0..k} ∧ i ∉ set xs ∧ j ∉ set xs ∧ distinct xs}

lemma distinct-length-le:finite s ⇒ set xs ⊆ s ⇒ distinct xs ⇒ length xs ≤ card s
  by (metis card-mono distinct-card)
lemma finite-distinct: finite \( s \implies \text{finite} \{ \text{set } xs \subseteq s \land \text{distinct } xs \} \)
proof
  assume finite \( s \)
  hence \( \{ \text{set } xs \subseteq s \land \text{distinct } xs \} \subseteq \{ \text{length } xs \leq \text{card } s \} \)
  using distinct-length-le by auto
moreover have finite \( \{ \text{set } xs \subseteq s \land \text{length } xs \leq \text{card } s \} \)
  using finite-lists-length-le \([\text{OF } \langle \text{finite } s \rangle]\) by auto
ultimately show \( \text{thesis} \) by (rule finite-subset)
qed

lemma D-base-finite:
finite \( \{ \text{len } m i j \text{ } xs \mid \text{set } xs \subseteq \{ 0..k \} \land \text{distinct } xs \} \)
using finite-distinct finite-image-set by blast

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

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

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

lemma D-eqI:
fixes m n i j k
defines \( A \equiv \{ \text{len } m i j \text{ } xs \mid \text{set } xs \subseteq \{ 0..k \} \} \)
defines \( A\text{-distinct } \equiv \{ \text{len } m i j \text{ } xs \mid \text{set } xs \subseteq \{ 0..k \} \land i \notin \text{set } xs \land j \notin \text{set } xs \land \text{distinct } xs \} \)
assumes cycle-free m n i \leq n j \leq n k \leq n (\forall y. y \in A\text{-distinct } \rightarrow x \leq y) \rightarrow x \in A
shows \( D \text{ } m i j k = x \) using assms
proof
  let \( \forall S = \{ \text{len } m i j \text{ } xs \mid \text{set } xs \subseteq \{ 0..k \} \land i \notin \text{set } xs \land j \notin \text{set } xs \land \text{distinct } xs \} \)
distinct \{xs\}

show ?thesis unfolding \(D\)-def

proof (rule Min-eqI)
  have \(?S \subseteq \{\text{len } m \ i \ j \ xs \mid \text{set } xs \subseteq \{0..k\} \land \text{distinct } xs\}\) by auto
  thus finite \(\{\text{len } m \ i \ j \ xs \mid \text{set } xs \subseteq \{0..k\} \land i \notin \text{set } xs \land j \notin \text{set } xs \land \text{distinct } xs\}\)
    using \(D\)-base-finite[of \(m\) \(i\) \(j\) \(k\)] by (rule finite-subset)
next
  fix \(y\) assume \(y \in ?S\)
  hence \(y \in A\)-distinct using assms(2,7) by fastforce
  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-def)
    have \(\text{distinct } i \notin \text{set } ?ys \ j \notin \text{set } ?ys \ \text{distinct } ?ys\)
      using \(\text{rem-cycles-distinct remove-all-removes rem-cycles-removes-last}\) by fast+
      with \(xs(2)\) have \(?y \in A\)-distinct unfolding \(A\)-distinct-def using rem-cyclessubs 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-cyclessubs[of \(i \ j \ xs\)] by fastforce
      qed
    qed

lemma \(D\)-base-not-empty:
\(\{\text{len } m \ i \ j \ xs \mid \text{set } xs \subseteq \{0..k\} \land i \notin \text{set } xs \land j \notin \text{set } xs \land \text{distinct } xs\}\) \(\neq \{\}\)

proof
  have \(\text{len } m \ i \ j \ [] \in \{\text{len } m \ i \ j \ xs \mid \text{set } xs \subseteq \{0..k\} \land i \notin \text{set } xs \land j \notin \text{set } xs \land \text{distinct } xs\}\)
    by fastforce
  thus ?thesis by auto
qed

lemma Min-elem-dest: finite \(A\) \(\Rightarrow\) \(A \neq \{\}\) \(\Rightarrow\) \(x = \text{Min } A\) \(\Rightarrow\) \(x \in A\) by simp

lemma \(D\)-dest: \(x = D \ m \ i \ j \ k\) \(\Rightarrow\)

30
\( x \in \{ \text{len } m \ i \ j \ xs \ | \ xs. \ set \ xs \subseteq \{ 0..\text{Suc } k \} \land i \notin \ set \ xs \land j \notin \ set \ xs \land \text{distinct } xs \} \)

**using** Min-elem-dest[OF D-base-finite'' D-base-not-empty] **by** (fastforce simp add: D-def)

**lemma** D-dest': \( x = D \ m \ i \ j \ k \implies x \in \{ \text{len } m \ i \ j \ xs \ | \ xs. \ set \ xs \subseteq \{ 0..\text{Suc } k \} \} \)

**using** Min-elem-dest[OF D-base-finite'' D-base-not-empty] **by** (fastforce simp add: D-def)

**lemma** D-dest'': \( x = D \ m \ i \ j \ k \implies x \in \{ \text{len } m \ i \ j \ xs \ | \ xs. \ set \ xs \subseteq \{ 0..k \} \} \)

**using** Min-elem-dest[OF D-base-finite'' D-base-not-empty] **by** (fastforce simp add: D-def)

**lemma** cycle-free-loop-dest: \( i \leq n \implies \text{set } xs \subseteq \{ 0..n \} \implies \text{cycle-free } m \ n \)

**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
\[
\text{cycle-free-up-to } m \ k \ n \equiv \forall i \ xs. \ i \leq n \land \text{set } xs \subseteq \{ 0..k \} \rightarrow \\
(\forall j. \ j \leq n \rightarrow \text{len } m \ i \ j \ (\text{rem-cycles } i \ j \ xs) \leq \text{len } m \ i \ j \ xs) \land \text{len } m \ i \ i \ xs \geq 0
\]

**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:
\begin{itemize}
  \item **assumes** cycle-free-up-to \( m \ k \ n \ i \leq n \)
  \item **shows** \( m \ i \ i \geq 0 \)
\end{itemize}

**using** cycle-free-up-to-loop-dest[OF assms(2) - assms(1), of []] **by** auto

**lemma** D-eqI2:
\begin{itemize}
  \item **fixes** \( m \ n \ i \ j \ k \)
  \item **defines** \( A \equiv \{ \text{len } m \ i \ j \ xs \ | \ xs. \ set \ xs \subseteq \{ 0..k \} \} \)
  \item **defines** \( A\text{-distinct} \equiv \{ \text{len } m \ i \ j \ xs \ | \ xs. \ set \ xs \subseteq \{ 0..k \} \land i \notin \ set \ xs \land j \notin \ set \ xs \land \text{distinct } xs \} \)
  \item **assumes** cycle-free-up-to \( m \ k \ n \ i \leq n \ j \leq n \ k \leq n \)
  \item \( (\forall y. \ y \in A\text{-distinct} \implies x \leq y) \ x \in A \)
\end{itemize}
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 ∈ A-distinct`
  thus `x ≤ y` using `assms` by meson
next
  from `assms` obtain `xs` where `xs: x = len m i j xs set xs ⊆ {0..k}` by auto
  let `?ys = rem-cycles i j xs`
  let `?y = len m i j ?ys`
  from `assms`(3–6) `xs` have `*: ?y ≤ 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 ∈ A-distinct` unfolding `A-distinct-def` using `rem-cycles-subs` by fastforce
    hence `x ≤ ?y` using `assms` by meson
  moreover have `?y ≤ x` using `assms`(3–6) `xs` by (fastforce simp add: `cycle-free-up-to-def`)
    ultimately have `x = ?y` by simp
  then show `x ∈ A-distinct` using `distinct` `xs(2)` `rem-cycles-subs`[`of i j xs`] unfolding `A-distinct-def` by fastforce
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-subslen:
  (fw m n k) i j ≤ len m i j xs if
cyc-free-subs n \{0..k\} m k ≤ n i ≤ n j ≤ n set xs ⊆ I I ⊆ \{0..k\}
proof –
  from fw-correct[OF that(1,2)] have canonical-subs n \{0..k\} (fw m n k)
  ..
  from canonical-subs-len[OF this, of i j xs] that have fw m n k i j ≤ len
    (fw m n k) i j xs
    by auto
  also from that(2–) have ... ≤ 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 len (fw m n k) x j xs ≤ len m x j xs
    by auto
  moreover from Cons.prems have fw m n k i x ≤ 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 fwi for a pair i, j always corresponds
to the length of an actual path between i and j.

lemma fwi-len':
  \exists xs. set xs ⊆ \{k\} ∧ fwi m n k i' j' i j = len m i j xs if
  m k k ≥ 0 i' ≤ n j' ≤ n k ≤ n i ≤ i' j ≤ j'
  using that apply (subst fwi-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. set xs ⊆ \{0..k\} ∧ fw m n k i j = len m i j xs if
  cyc-free-subs n \{0..k\} m i ≤ n j ≤ n k ≤ n
proof (induction k arbitrary: i j)
  case 0
  from cyc-free-subs-diag[OF this(1)] have m 0 0 ≥ 0 by blast
  with 0 show ?case by (auto intro: fwi-len')
next
  case (Suc k)
  have IH: ∃ xs. set xs ⊆ {0..k} ∧ fw m n k i j = len m i j xs if i ≤ n j
    ≤ n for i j
    apply (rule Suc.IH)
    using Suc.prems that by force+
  from fw-cyc-free-subs[OF Suc.prems(1,4)] have cyc-free-subs n {0..Suc k} (fw m n (Suc k)) .
  then have 0 ≤ fw m n k (Suc k) (Suc k) using IH Suc.prems(1, 4) by fastforce
  with Suc.prems fwi-len'[of fw m n k Suc k n n i j] obtain zs where
    set zs ⊆ {Suc k} fwi (fw m n k) n (Suc k) n n i j = len (fw m n k) i j xs
    by auto
  moreover from Suc.prems(2−) this(1) have
    ∃ ys. set ys ⊆ {0..Suc k} ∧ len (fw m n k) i j xs = 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:
      set ys ⊆ {0..Suc k} len (fw m n k) x j xs = len m x j ys
      by force
    moreover from IH[of i x] Cons.prems obtain zs where
      set zs ⊆ {0..k} fw m n k i x = len m i x zs
      by auto
    ultimately have
      set (zs @ x # ys) ⊆ {0..Suc k} len (fw m n k) i j (x # xs) = len m i j (zs @ x # ys)
      using Suc k ≤ n ∨ set (x # xs) ⊆ 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 $xs' = i \#\,ys$
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 $\exists\,ys.\,\text{set}\,ys \subseteq \text{set}\,xs \land \text{len}\,m\,i\,i\,ys < 0$ using assms
proof (induction $xss\,\text{arbitrary:}\,xs\,as$)
case $\text{Nil}$
with $\text{Nil}$ show $?\text{case}$
proof (cases $\text{len}\,m\,i\,i\,as \geq 0$, goal-cases)
case 1
from this(4,6) $\text{len-decomp}[of\,xs\,as\,i\,ys\,m\,j]$ 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 $\text{add-mono}$ by fastforce
thus $?\text{thesis\,using}\,\text{Nil}(5)$ by auto
next
case 2 thus $?\text{case\,by\,auto}$
qed
next
case ($\text{Cons}\,zs\,xss$)
let $?xs = zs \@\,\text{concat}\,(\text{map}\,(\#\,i)\,xss) \@\,xs'$
from $\text{Cons}$ show $?\text{case}$
proof (cases $\text{len}\,m\,i\,i\,as \geq 0$, goal-cases)
case 1
from this(5,7) $\text{len-decomp\,add-mono}$ 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 $\text{Cons}(2)$ by auto
have set $?xs \subseteq \text{set}\,xs$ using $\text{Cons}(5)$ by auto
moreover from $\text{Cons}(1)[\text{OF}\,1(2,3)\,2 - 4]$ have $\exists\,ys.\,\text{set}\,ys \subseteq \text{set}\,?xs$
\wedge \text{len}\,m\,i\,i\,ys < 0$ by auto
ultimately show $?\text{case\,by\,blast}$
next
case 2
from this(5,7) show $?\text{case\,by\,auto}$
qed
qed

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

35
from add-mono\(\text{OF this, of } b \ b \} \langle a + b < b \rangle \) show False by auto

dvd

lemma remove-cycles-neg-cycles-aux\(\uparrow\):
\begin{align*}
&\text{fixes } j \ \text{xs ys} \\
&\text{assumes } j \notin \text{set } \text{ys} \\
&\text{assumes } j \in \text{set } \text{xs} \\
&\text{assumes } \text{xs} = \text{ys} @ j \# \text{concat } (\text{map } (\lambda \text{xs. } \text{xs} @ [j]) \text{ xss}) @ \text{as} \\
&\text{assumes } \text{len } \text{m} i j \text{ys} > \text{len } \text{m} i j \text{xs} \\
&\text{shows } \exists \text{ys. set } \text{ys} \subseteq \text{set } \text{xs} \land \text{len } \text{m} j j \text{ys} < 0 \text{ using } \text{assms} \\
\end{align*}

proof (induction \text{xss} arbitrary: \text{xs} as)
\begin{align*}
&\text{case Nil} \\
&\text{show } ?\text{case} \\
&\text{proof (cases } \text{len } \text{m} j j \text{as } \geq 0) \\
&\text{case True} \\
&\text{from Nil(3) len-decomp[of } \text{xs } \text{ys } j \text{ as } \text{m } j \text{] } \\
&\text{have } \text{len } \text{m} i j \text{xs} = \text{len } \text{m} i j \text{ys} + \text{len } \text{m} j j \text{as} \text{ by simp} \\
&\text{with True} \\
&\text{have } \text{len } \text{m} i j \text{ys} \leq \text{len } \text{m} i j \text{xs} \text{ using add-mono by fastforce} \\
&\text{with Nil show } ?\text{thesis by auto} \\
&\text{next} \\
&\text{case False with Nil show } ?\text{thesis by auto} \\
\end{align*}

qed

next
\begin{align*}
&\text{case } (\text{Cons } \text{zs } \text{xss}) \\
&\text{let } ?\text{xs} = \text{ys} @ j \# \text{concat } (\text{map } (\lambda \text{xs. } \text{xs} @ [j]) \text{ xss}) @ \text{as} \\
&\text{let } ?t = \text{concat } (\text{map } (\lambda \text{xs. } \text{xs} @ [j]) \text{ xss}) @ \text{as} \\
&\text{show } ?\text{case} \\
&\text{proof (cases } \text{len } \text{m} i j ?\text{xs} \leq \text{len } \text{m} i j \text{xs}) \\
&\text{case True} \\
&\text{hence } 4: \text{len } \text{m} i j ?\text{xs} < \text{len } \text{m} i j \text{ys} \text{ using Cons(5) by simp} \\
&\text{have } 2:j \in \text{set } ?\text{xs} \text{ using Cons(2) by auto} \\
&\text{have set } ?\text{xs} \subseteq \text{set } \text{xs} \text{ using Cons(4) by auto} \\
&\text{moreover from Cons(1)}(\text{OF Cons(2) 2 - 4} \text{ have } \exists \text{ys. set } \text{ys} \subseteq \text{set } ?\text{xs} \\
\land \text{len } \text{m} j j \text{ys} < 0 \text{ by blast} \\
&\text{ultimately show } ?\text{thesis by blast} \\
&\text{next} \\
&\text{case False} \\
&\text{hence } \text{len } \text{m} i j \text{xs} < \text{len } \text{m} i j ?\text{xs} \text{ by auto} \\
&\text{from this len-decomp Cons(4) add-mono} \\
&\text{have } \text{len } \text{m} j j (\text{concat } (\text{map } (\lambda \text{xs. } \text{xs} @ [j]) (\text{zs } \# \text{xss}) @ \text{as}) < \text{len } \text{m} j j ?t} \\
&\text{using False } \text{locale } \text{leI by fastforce} \\
&\text{hence } \text{len } \text{m} j j (\text{zs } @ j \# ?t < \text{len } \text{m} j j ?t \text{ by simp} \\
\end{align*}
with \( \text{len-decomp}[\text{of } zs \# j \# ?t zs ?t m j j] \)

have \( \text{len } m j j zs + \text{len } m j j ?t < \text{len } m j j ?t \) by auto

hence \( \text{len } m j j zs < 0 \) using \( \text{add-lt-neutral} \) by auto

thus \( \text{thesis using } \text{Cons.prems(3)} \) by auto

qed

with \( \text{len-decomp} \) of \( zs @ j \# ?t zs j ?t m j j \)

have \( \text{len } m j j zs + \text{len } m j j ?t < \text{len } m j j ?t \) by auto

hence \( \text{len } m j j zs < 0 \) using \( \text{add-lt-neutral} \) by auto

thus \( \text{thesis using } \text{Cons.prems(3)} \) by auto

qed

lemma \( \text{add-le-impl} \): \( a + b < a + c \implies b < c \)

proof (rule \text{ccontr})

assume \( a + b < a + c \sim b < c \)

hence \( b \geq c \) by auto

from \( \text{add-mono}[OF \ this, \ of } a a \) \( (a + b < a + c) \) show \( \text{False by auto} \)

qed

lemma \( \text{start-remove-neg-cycles} \):

\( \text{len } m i j (\text{start-remove } xs k []) > \text{len } m i j xs \implies \exists \ ys. \ set \ ys \subseteq \ set \ xs \land \text{len } m k k ys < 0 \)

proof–

let \( ?xs = \text{start-remove } xs k [] \)

assume \( \text{len-lt}: \text{len } m i j ?xs > \text{len } m i j xs \)

hence \( k \in \set xs \) using \( \text{start-remove-id} \) by fastforce

from \( \text{start-remove-decomp}[OF \ this, \ of } [] \) obtain \( as \ bs \) where \( as-bs: \)

\( xs = \text{as @ } k \# \text{bs } ?xs = \text{as @ remove-cycles } bs \ k \ [k] \)

by fastforce

let \( ?xs' = \text{remove-cycles } bs \ k \ [k] \)

have \( k \in \set bs \) using \( \text{as-bs len-lt remove-cycles-id} \) by fastforce

then obtain \( ys \) where \( ?xs = \text{as @ } k \# \text{ys } ?xs' = k \# \text{ys } k \notin \set ys \)

using \( \text{as-bs(2) remove-cycles-begins-with[OF } k \in \set bs] \) by auto

have \( \text{len-lt}: \text{len } m k j bs < \text{len } m k j ys \)

using \( \text{len-decomp}[OF \ as-bs(1), \ of } m i j \] \( \text{len-decomp}[OF \ ys(1), \ of } m i j \] \( \text{len-lt add-le-impl} \) by metis

from \( \text{remove-cycles-cycles}[OF \ \text{of } k \in \set bs] \) obtain \( \text{zss } \text{as'} \)

where \( \text{as'} @ \text{concat } (\text{map } ((\#) \ k) \ \text{zss}) @ ?xs' = bs \) by fastforce

hence \( \text{as'} @ \text{concat } (\text{map } ((\#) \ k) \ \text{zss}) @ k \# \text{ys } k = bs \) using \( \text{ys(2)} \) by simp

from \( \text{remove-cycles-neg-cycles-aux}[OF \ \text{of } k \notin \set ys \] \( \text{of } k \in \set bs \) \( \text{this[symmetric]} \]

\( \text{len-lt} \)

show \( \text{thesis using } \text{as-bs(1)} \) by auto

qed

lemma \( \text{remove-all-cycles-neg-cycles} \):

\( \text{len } m i j (\text{remove-all-cycles } ys \ xs) > \text{len } m i j xs \)

\implies \( \exists \ ys k. \ set \ ys \subseteq \ set \ xs \land k \in \set xs \land \text{len } m k k ys < 0 \)

proof (induction \( ys \) arbitrary: \( xs \))

37
```plaintext

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-subs[of xs y []] by auto 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: len m i j (rem-cycles i j xs) > len m i j xs ⇒ ∃ i' ys. len m i' i' ys < 0 ∧ set ys ⊆ set xs ∧ i' ∈ 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: $j \notin \text{set } ?xs$ 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 @ concat (map ((#) $j$) $xss$) @ remove-cycles (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 $j \in \text{set } (\text{rev } ?xs)$, of []]
    have remove-cycles (rev ?xs) $j [] = j \# \text{rev } ?xsj \ j \notin \text{set } ?xsj$
    by auto
    with as(1) have xss: as @ concat (map ((#) $j$) $xss$) @ $j \# \text{rev } ?xsj$
      = rev ?xs by simp
      hence rev (as @ concat (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 $?xsj @ j \# \text{concat } (\lambda x, x @ [j]) (\text{rev } (\text{map rev } xss)))$
          @ 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': len $m \ i \ j \ ?xsj > \text{len } m \ i \ j \ ?xsj$ 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: $i \notin \text{set } ?xsj$ by (simp add: remove-all-def)
  from remove-cycles-cycles[OF True] obtain $xss$ as where as:
    as @ concat (map ((#) $i$) $xss$) @ remove-cycles $?xsj \ i \ [] = \text{rev } ?xsj \ i \notin \text{set }
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 # set ?xsij
    by auto
  with as(1) have xs: as @ concat (map ((#) i) xs) @ i # ?xsij = ?xsj by simp
from remove-cycles-neg-cycles-aux[of 1 True this[symmetric] len-lt]
  show ?thesis using remove-all-revsubs remove-all-cycles-sub by fastforce
qed
qed

1.6.2 Cycle-Freeness

lemma cycle-free-alt-def:
  cycle-free M n ←→ cycle-free-up-to M n n
unfolding cycle-free-def cycle-free-up-to-def ..

lemma negative-cycle-dest-diag:
  ¬ cycle-free-up-to m k n ⇒ k ≤ n ⇒ ∃ i xs. i ≤ n ∧ set xs ⊆ {0..k} ∧ 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 ⊆ set xs i' ∈ set (i # j # xs) by auto
from this(2,3) 1(1-4) have set ys ⊆ {0..k} i' ≤ n by auto
  with * show ?case by auto
next
  case 2 then show ?case by fastforce
qed

lemma negative-cycle-dest-diag':
  ¬ cycle-free m n ⇒ ∃ i xs. i ≤ n ∧ set xs ⊆ {0..n} ∧ len m i i xs < 0
by (rule negative-cycle-dest-diag) (auto simp: cycle-free-alt-def)

abbreviation cyc-free :: 'a mat ⇒ nat ⇒ bool where
cyc-free m n ≡ ∀ i xs. i ≤ n ∧ set xs ⊆ {0..n} → len m i i xs ≥ 0

lemma cycle-free-diag-intro:
cyc-free m n ⇒ cycle-free m n
using negative-cycle-dest-diag′ by force

lemma cycle-free-diag-equiv:
cyc-free m n ⟷ cycle-free m n using negative-cycle-dest-diag′
by (force simp: cycle-free-def)

lemma cycle-free-diag-dest:
cycle-free m n ⟹ cyc-free m n
using cycle-free-diag-equiv by blast

lemma cycle-free-up-to-diag-eqv:
cycle-free-up-to m k n ⟷ cyc-free-subs n {0..k} m if k ≤ 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-subs n {0..k} m i ≤ n j ≤ n k ≤ 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-subs n {0..k} (fw m n k) cyc-free-subs 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-subs-len)
next
  case 6
  from fw-len[OF that(1) that(2-)] show ?case by blast
qed (rule that cycle-free)+

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

lemma
  cyc-free m n ⟷ cyc-free-subs n {0..n} m ..

lemma cycle-free-cyc-free-up-to:
cycle-free m n ⟹ k ≤ n ⟹ 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 ⟹ i ≤ n ⟹ 0 ≤ m i i
using cycle-free-up-to-diag[of cycle-free-cycle-free-up-to] by blast
corollary fw-shortest-path:
cyc-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
by (rule fw-shortest-path-up-to; force)

corollary fw-shortest:
assumes cyc-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-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-cycle-free-dest: \neg cycle-free m n \implies \exists k \leq n \cdot \neg 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 |xs. set xs \subseteq \{0..k\} \land i \notin set xs \land j \notin set xs \land distinct xs}
\implies D m i j k \leq x using Min-le[OF D-base-finite'] by (auto simp add: D-def)

lemma D-not-diag-le': set xs \subseteq \{0..k\} \implies i \notin set xs \implies j \notin set xs \implies distinct xs
\implies D m i j k \leq len m i j xs using Min-le[OF D-base-finite'] by (fastforce simp add: D-def)

lemma nat-upto-sub-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-sub-top-removal:
S \subseteq \{0..n::nat\} \implies n \notin S \implies S \subseteq \{0..n - 1\}
using nat-upto-sub-top-removal' by (cases n; simp)
Monotonicity with respect to $k$.

**Lemma** $fw$-invariant:

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

**Proof** (induction $k$)

- case 0
  - then show ?case by (auto intro: $fw$-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 $i \leq n$ $j \leq n$ have $fw \ m \ n \ (Suc \ k) \ i \ j \leq fw \ m \ n \ k \ i \ j$
              - by (auto intro: $fw$-mono)
            - ultimately show ?thesis by auto
    - qed
  - qed

**Lemma** negative-len-shortest:

$\text{length} \ xs = n \Rightarrow \text{len} \ m \ i \ i \ xs < 0$

$\Rightarrow \exists \ j \ ys. \ \text{distinct} \ (j \# \ ys) \ \land \ \text{len} \ m \ j \ j \ ys < 0 \ \land \ j \ \in \ \text{set} \ (i \# \ xs) \ \land \ \text{set} \ ys \ \subseteq \ \text{set} \ xs$

**Proof** (induction $n$ arbitrary; $xs$ i rule: less-induct)

- case (less $n$)
  - show ?case
    - proof (cases $xs$)
      - case Nil
        - thus ?thesis using less.prems by auto
  - next
    - case (Cons $y$ $ys$)
      - then have $\text{length} \ xs \geq 1$ by auto
      - show ?thesis
        - proof (cases $i \in \ \text{set} \ xs$)
          - assume $i: \ i \in \ \text{set} \ xs$
            - then obtain $as \ bs$ where $xs: \ xs = as \ @ \ i \# \ bs$ by (meson split-list)
          - show ?thesis
            - proof (cases $\text{len} \ m \ i \ i \ as < 0$)
              - case True
from \( xs \) less \( \text{prems} \) have length as < \( n \) by auto \\
from less.IH[\( \text{OF} \ this \ True \)] \( xs \) show ?thesis by auto \\
next \\
\hspace{1em} \text{case } False \\
\hspace{1em} from len-decomp[\( \text{OF} \ xs \)] have len \( m \ i \ i \ xs \) = len \( m \ i \ i \ as \) + len \( m \ i \ bs \) by auto \\
\hspace{1em} with False less \( \text{prems} \) have * : len \( m \ i \ i \ bs \) < 0 \\
\hspace{1.5em} by (metis add-it-neutral local.dual-order.strict-trans local.neqE) \\
from \( xs \) less \( \text{prems} \) have length \( bs \) < \( n \) by auto \\
from less.IH[\( \text{OF} \ this \ * \)] \( xs \) show ?thesis by auto \\
qed \\
next \\
\hspace{1em} assume \( i : i \notin \text{set} \ xs \) \\
show ?thesis \\
proof \( (\text{cases} \ \text{distinct} \ xs) \) \\
\hspace{1em} \text{case } True \\
\hspace{2em} with i less \( \text{prems} \) show ?thesis by auto \\
next \\
\hspace{1em} case False \\
\hspace{1.5em} from not-distinct-decomp[\( \text{OF} \ this \)] obtain \( a \) as \( bs \) cs where \( xs \) : \\
\hspace{2em} \( xs = \text{as} \ @ \ a \ # \ \text{bs} \ @ \ a \ # \ \text{cs} \) \\
\hspace{1.5em} by auto \\
show ?thesis \\
proof \( (\text{cases} \ \text{len} \ m \ a \ a \ bs < 0 \) \\
\hspace{1em} \text{case } True \\
\hspace{2em} from \( xs \) less \( \text{prems} \) have length \( bs \) < \( n \) by auto \\
\hspace{2em} from less.IH[\( \text{OF} \ this \ True \)] \( xs \) show ?thesis by auto \\
next \\
\hspace{1em} case False \\
\hspace{1.5em} from len-decomp[\( \text{OF} \ xs \), \( \text{of} \ m \ i \ i \)] len-decomp[\( \text{of} \ bs \ @ \ a \ # \ \text{cs} \ \text{bs} \ a \] \\
\hspace{2em} \text{cs} m a i \] \\
\hspace{2em} have * : len \( m \ i \ i \ xs \) = len \( m \ i \ a \ as \) + (len \( m \ a \ a \ bs \) + len \( m \ a \ i \ cs \)) \\
\hspace{2em} by auto \\
\hspace{2em} from False have len \( m \ a \ a \ bs \) ≥ 0 by auto \\
\hspace{2em} with add-mono have len \( m \ a \ a \ bs \) + len \( m \ a \ i \ cs \) ≥ len \( m \ a \ i \ cs \) \\
\hspace{3em} by fastforce \\
\hspace{2em} with * have \( \text{len} \ m \ i \ i \ xs \geq \text{len} \ m \ i \ a \ as \) + \( \text{len} \ m \ a \ i \ cs \) by (simp add: add-mono) \\
\hspace{3em} with less \( \text{prems}(2) \) have \( \text{len} \ m \ i \ a \ as \) + \( \text{len} \ m \ a \ i \ cs \) < 0 by auto \\
\hspace{3em} with len-comp have \( \text{len} \ m \ i \ i \ (a@a\#cs) < 0 \) by auto \\
\hspace{2em} from less.IH[\( \text{OF} \ - \ this, \ \text{of} \ \text{length} \ (a@a\#cs) \)] \( xs \) less \( \text{prems} \) \\
show ?thesis by auto \\
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:
  ¬ cyc-free m n ⇒ ∃ i ≤ n. fw m n n i i < 0
proof –
  assume A: ¬ cyc-free m n
  let ?K = \{k. k ≤ n ∧ ¬ cyc-free-subs n \{0..k\} m\}
  define k where k = Min ?K
  have not-empty-K: ?K ≠ {} using A by auto
  have finite ?K by auto
  with not-empty-K have *:
    ∀ k' < k. cyc-free-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
    ¬ cyc-free-subs n \{0..k\} m k ≤ n
  unfolding k-def by auto
  then have ∃ xs j. j ≤ n ∧ len m j j xs < 0 ∧ set xs ⊆ \{0..k\}
    by force
  then obtain a as where a-as: a ≤ n ∧ len m a a as < 0 ∧ set as ⊆ \{0..k\} by auto
  with negative-len-shortest[of as length as m a] obtain j xs where j-xs:
    distinct (j ≠ xs) ∧ len m j j xs < 0 ∧ j ∈ set (a ≠ as) ∧ set xs ⊆ set as
    by auto
  with a-as (k ≤ n) have cyc: j ≤ n set xs ⊆ \{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 \( n \{0..k - 1\} m \) by blast \\
have \( k \in \text{set } xs \) \\
proof (rule contr, goal-cases) \\
  case \( 1 \) \\
  with \( \{\text{set } xs \subseteq \{0..k\}\} \) nat upto subs top removal have \( \text{set } xs \subseteq \{0..k-1\} \) by auto \\
  with \( \text{cyc-free-sub } n \{0..k - 1\} \) \( m \) \( j \leq n \) have \( 0 \leq \text{len } m \ j \ j \) xs by blast \\
  with cyc(3) show \(?\) case by simp \\
qed \\
with \( \text{cyc}(4)\) have \( j \neq k \) by auto \\
from \( \{k \in \text{set } xs\} \) obtain \( ys \ zs \) where \( xs = ys \ @ k \ # zs \) by (meson split-list) \\
  with \( \{\text{distinct } (j \ # xs)\} \)
  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 \( xs(1,4)\) \( \{\text{set } xs \subseteq \{0..k\}\} \) nat upto subs top removal have \( ys: \text{set } ys \subseteq \{0..k-1\} \) by auto \\
from \( xs(1,5)\) \( \{\text{set } xs \subseteq \{0..k\}\} \) nat upto subs 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 \( \{k \leq n\} \) \( j \leq n\) \( \text{fw-shortest-path-up-to}[OF **] \) by auto \\
moreover have \( D \ m \ j \ k \ (k - 1) = \text{fw } m \ n \ (k - 1) \ k \ j \)
  using \( \{k \leq n\} \) \( j \leq n\) \( \text{fw-shortest-path-up-to}[OF **] \) by auto \\
ultimately have \( \text{fw } m \ n \ (k - 1) \ j \ k + \text{fw } m \ n \ (k - 1) \ k \ j \leq \text{len } m \ j \ k \ ys + \text{len } m \ j \ k \ zs \)
  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 \( \text{neg: } \text{fw } m \ n \ (k - 1) \ j \ k + \text{fw } m \ n \ (k - 1) \ k \ j \leq 0 \)
  using \( xs(1) \) \( \text{len } m \ j \ j \ xs < 0\) \( \text{len-comp} \) 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 \( k > 0 \) \( \) have \( \ast : \text{fw } m \ n \ k = \text{fw } m \ n \ (k - 1) \) \( n \ k \ n \)
  by (cases \( k \)) auto \\
from \( \text{fw-cyc-free-sub}[OF **, \) \( \text{THEN} \) \( \text{cyc-free-sub-diag} \) \( k \leq n\) \ have \( \text{fw } m \ n \ (k - 1) \ k \ k \geq 0 \)
  by auto \\
from \( \text{fw-step}[\) \( \text{of } \text{fw } m \ n \ (k - 1), \ \text{OF} \ this \) \( k \leq n \) \( j \leq n\) \ show 
  ?thesis 
  by (auto intro: min.cobounded2 simp: \ast \)
qed

with neg have \( \text{fw} \ m \ n \ k \ j < 0 \) by auto

moreover from \( \text{fw-invariant} \) \( \langle j \leq n \rangle \langle k \leq n \rangle \) have \( \text{fw} \ m \ n \ n \ j \ j \leq \text{fw} \ m \ n \ k \ j \ j \)
  by blast

ultimately have \( \text{thesis} \) using \( \langle j \leq n \rangle \) by auto

moreover

\{ assume \( k = 0 \)

  with cyc(2,4) have \( x = \emptyset \lor x = [0] \)
  apply safe
  apply (case-tac xs)
  apply fastforce
  apply (rename-tac ys)
  apply (case-tac ys)
  apply auto
done

then have \( \text{thesis} \)

proof

assume \( x = \emptyset \)

with cyc have \( m \ j \ j < 0 \) by auto

with \( \text{fw-mono}[\text{of} \ n \ j \ n \ m] \langle j \leq n \rangle \) have \( \text{fw} \ m \ n \ n \ j \ j < 0 \) by auto

with \( \langle j \leq n \rangle \) show \( \text{thesis} \) by auto

next

assume \( x: x = [0] \)

with cyc have \( m \ j \ 0 + m \ 0 \ j < 0 \) by auto

moreover from \( \langle j \leq n \rangle \) have \( \text{fw} \ m \ n \ 0 \ j \ j \leq \text{fw-upd} \ m \ 0 \ j j \ j \)
  by (auto intro: order_trans[OF \( \text{fw-invariant} \) \( \text{fw-fw-upd-mono} \])

ultimately have \( \text{fw} \ m \ n \ 0 \ j \ j < 0 \)

  unfolding \( \text{fw-upd-def} \) \( \text{upd-def} \) by auto

  then have \( \text{fw} \ m \ n \ 0 \ j \ j < 0 \) by (metis cyc(1) less-or-eq-imp-le
  single-iteration-inv)

  with \( \langle j \leq n \rangle \) have \( \text{fw} \ m \ n \ n \ j \ j < 0 \) using \( \text{fw-invariant}[\text{of} \ n \ j \ n \ j \ m] \) by auto

  with \( \langle j \leq n \rangle \) show \( \text{thesis} \) by blast

qed

\}

ultimately show \( \text{thesis} \) by auto

qed
1.8 More on Canonical Matrices

abbreviation
canonical M n ≡ ∀ i j k. i ≤ n ∧ j ≤ n ∧ k ≤ n → M i k ≤ M i j + M j k

lemma canonical-alt-def:
canonical M n ←→ canonical-subsn {0..n} M

unfolding canonical-subsn-def by auto

lemma fw-canonical:
canonical (fw m n n) n if cyc-free m n
using fw-canonical-subs[of cyc-free m n] unfolding canonical-alt-def by auto

lemma canonical-len:
canonical M n ⇒ i ≤ n ⇒ j ≤ n ⇒ set xs ⊆ {0..n} ⇒ M i j ≤ 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 ≤ len M x j xs by auto
from Cons.prems (canonical M n) have M i j ≤ M i x + M x j by simp
also with Cons have ... ≤ M i x + 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
⇒ ∀ i ≤ n. ∀ j ≤ n. D m i j n = m’ i j ⇒ i ≤ n ⇒ j ≤ n ⇒ set xs ⊆ {0..n} ⇒ ∃ ys. set ys ⊆ {0..n} ∧ len m’ i j xs = 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 set ys ⊆ {0..n} len m’ i j [] = len m i j ys by auto
then show ?case by auto
next
case (Cons y ys)
from Cons.IH(OF Cons.prems(1,2) - \( j \leq n \), of y) Cons.prems(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(5)(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 Cons.prems(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 \( \subseteq \{0..n\} \land len m' i i xs = len m i i ys \) 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 > 0
by (rule cyc-free-subs-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)] \langle i \leq n \rangle]
  show (FW M n) i i = 0 by auto
qed

lemma FW-fixed-preservation:
  fixes M :: ('a::linordered_ab_monoid_add) 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 canonical ?M' n \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:
  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 = this
  then have i \leq n \land 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:
  canonical-subs n \{k\} M \implies i \leq n \implies j \leq n \implies 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 $M_{ij} \leq M_{ik} + M_{kj}$ unfolding canonical-subs-def by auto
then have $\min (M_{ij}) (M_{ik} + M_{kj}) = M_{ij}$ by (simp split; split-min)
thus ?case by force
qed

lemma fw-upd-canonical-id:
  canonical $M \equiv i \leq n \Rightarrow j \leq n \Rightarrow k \leq n \Rightarrow fw-upd M_{ikj} = M$
using $fw-upd-canonical-subs-id[of n k M_{ij}]$ unfolding canonical-subs-def by auto

lemma fwi-canonical-id:
  $fwi M_{n k j} = M$ if canonical-subs $n \{k\}$ $M_{i j} \leq n j \leq n k \leq n$
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 $n (I \cup \{k::nat\}) (FWI M_{n k}) \lor (\exists i \leq n. FWI M_{n k i} < 0)$ 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 $\langle k \leq n \rangle$] show ?thesis unfolding FWI-def..
next
  case False
  with $\langle k \leq n \rangle$ fwi-mono[OF $\langle k \leq n \rangle \langle k \leq n \rangle$, of $M_{k n n}$] show ?thesis
    unfolding FWI-def by fastforce
qed
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] (λ x. True) (λ k a. (f a k)) a0

fun for-rec :: ('a ⇒ nat ⇒ 'a nres) ⇒ 'a ⇒ nat ⇒ 'a nres where
  for-rec f a 0 = f a 0 |
  for-rec f a (Suc n) = for-rec f a n ≫ (λ 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 (λa i. for-rec (λ a. f a i) a n) a
  (i - 1))
  ≫ (λ a. for-rec (λ a. f a i) a j)

fun for-rec2 :: ('a ⇒ nat ⇒ nat ⇒ 'a nres) ⇒ 'a ⇒ nat ⇒ nat ⇒ nat ⇒ '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 ≫ (λ a. f a (Suc i) 0) |
  for-rec2 f a n i (Suc j) = for-rec2 f a n i j ≫ (λ 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

end
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 (λa 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 (λa 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 (λa 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] (λx. True) (λk a. f a k) a
using for-comb-for-rec[unfolded for-comb-def, symmetric].
**Theorem** for-rec2-eq:

\[
\text{for-rec2 } f \ a \ n \ n \ n = \nfoldli \{0..<n+1\} (\lambda x. \text{True}) \nfoldli \{0..<n+1\} (\lambda x. \text{True}) (\lambda j \ a \ i j) \ a
\]

using

\[
\text{for-rec2'}-\text{for-rec}[
\text{unfolded for-rec2-for-rec2}[\text{symmetric}], \text{unfolded for-comb-for-rec}[\text{symmetric}]
\text{for-comb-def}
\]

**Theorem** for-rec3-eq:

\[
\text{for-rec3 } f \ a \ n \ n \ n \ n = \nfoldli \{0..<n+1\} (\lambda x. \text{True}) \nfoldli \{0..<n+1\} (\lambda x. \text{True}) \nfoldli \{0..<n+1\} (\lambda x. \text{True}) (\lambda j \ a \ k \ i j)
\]

using

\[
\text{for-rec3'}-\text{for-rec}[
\text{unfolded for-rec3-for-rec3}[\text{symmetric}], \text{unfolded for-comb-for-rec}[\text{symmetric}]
\text{for-comb-def}
\]

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-upd' :: ('a::linordered_ab_monoid_add) mtx ⇒ nat ⇒ nat ⇒ nat ⇒ 'a mtx nres where
fw-upd' m k i j =
RETURN (op-mtx-set m (i, j) (min (op-mtx-get m (i, j)) (op-mtx-get m (i, k) + op-mtx-get m (k, j))))

lemma fw-upd'-alt-def:
fw-upd' m k i j =
RETURN (let
  e = op-mtx-get m (i, k) + op-mtx-get m (k, j)
in if e < op-mtx-get m (i, j) then op-mtx-set m (i, j) e else m )
unfolding fw-upd'-def min-def Let-def by auto

definition fwi' :: ('a::linordered_ab_monoid_add) mtx ⇒ nat ⇒ nat ⇒ nat ⇒ nat ⇒ 'a mtx nres where
fwi' m n k i j = RECT (λ fw (m, k, i, j). case (i, j) of
  (0, 0) ⇒ fw-upd' m k 0 0 |
  (Suc i, 0) ⇒ do {m' ← fw (m, k, i, n); fw-upd' m' k (Suc i) 0} |
  (i, Suc j) ⇒ do {m' ← fw (m, k, i, j); fw-upd' m' k i (Suc j)}
) (m, k, i, j)

lemma fwi'-simps:
fwi' m n k 0 0 = fw-upd' m k 0 0
fwi' m n k (Suc i) 0 = do {m' ← fwi' m n k i n; fw-upd' m' k (Suc i) 0}
fwi' m n k i (Suc j) = do {m' ← fwi' m n k i j; fw-upd' m' k i (Suc j)}
unfolding fwi'-def by (subst RECT-unfold, (refine-mono; fail), (auto split: nat.split; fail))+

lemma fwi' m n k i j ≤ SPEC (λ r. r = uncurry (fwi (curry m) n k i j))
by (induction curry m n k i j arbitrary: m rule: fwi.induct)
  (fastforce simp add: fw-upd'-def fw-upd-def upd-def fwi'-simps pw-le-iff refine-pw-simps)+

lemma fw-upd'-spec:
fw-upd' M k i j ≤ SPEC (λ M'. M' = uncurry (fwi-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 (λ M. fw-upd' M k) M n i j \leq SPEC (λ M'. M' = uncurry (fwi (curry M) n k i j))
using fw-upd'-spec
by (induction λ M. fw-upd' (M :: (nat × nat ⇒ '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 ⇒ nat ⇒ nat ⇒ 'a
  where
  \begin{align*}
  fw' m n k = & nfoldli [0..<k + 1] (λ - . True) (λ k M. for-rec2 (λ M. fw-upd' M k) M n n) m
  \end{align*}

lemma fw'-spec:
  fw' m n k \leq SPEC (λ 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⇒-) ∈ Id → Id → Id by simp
lemma [sepref-import-param]: (min,min::'a⇒-) ∈ Id → Id → Id by simp

abbreviation node-assn ≡ nat-assn
abbreviation mtx-assn ≡ asmtx-assn (Suc n) id-assn::('a mtx ⇒-)

sepref-definition fw-upd-impl1 is
uncurry2 (uncurry fw-upd') ::
  [λ (((,-),i),j). k \leq n ∧ i \leq n ∧ j \leq n]_a mtx-assn^d *_{a} node-assn^k *_{a}
node-assn^k *_{a} node-assn^k
  → mtx-assn
  unfolding fw-upd'-def by sepref

sepref-definition fw-upd-impl is
uncurry2 (uncurry fw-upd') ::
  [λ (((,-),i),j). k \leq n ∧ i \leq n ∧ j \leq n]_a mtx-assn^d *_{a} node-assn^k *_{a}
node-assn^k *_{a} node-assn^k

56
\[ \to \text{mtx-assn} \]

**unfolding** \( \text{fw-upd'}\)-alt-def by sepref

**sepref-register** \( \text{fw-upd'} :: \ 'a \ i\text{-}mtx \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \ 'a \ i\text{-}mtx \text{nres} \)

**definition**

\( \text{fwi-impl'} \ (M :: \ 'a \ \text{mtx}) \ k = \text{for-rec2} \ (\lambda \ M. \ \text{fw-upd'} \ M \ k) \ M \ n \ n \ n \)

**definition**

\( \text{fw-impl'} \ (M :: \ 'a \ \text{mtx}) = \text{fw'} \ M \ n \ n \)

**context**

\[ \text{notes} [\text{id-rules}] = \text{itypeI[of n \ TYPE \ (nat)]} \]

and \[ \text{[sepref-import-param]} = \text{IdI[of n]} \]

**begin**

**sepref-definition** \( \text{fw-impl} \) is

\( \text{fw-impl'} :: \text{mtx-assn}^d \rightarrow_a \text{mtx-assn} \)

**unfolding** \( \text{fw-impl'}\)-def[abs-def] \( \text{fw'}\)-def for-rec2-eq

**supply** \[ \text{[sepref-fr-rules]} = \text{fw-upd-impl}\text{.refine} \]

by sepref

**sepref-definition** \( \text{fw-impl1} \) is

\( \text{fw-impl'} :: \text{mtx-assn}^d \rightarrow_a \text{mtx-assn} \)

**unfolding** \( \text{fw-impl'}\)-def[abs-def] \( \text{fw'}\)-def for-rec2-eq

**supply** \[ \text{[sepref-fr-rules]} = \text{fw-upd-impl1}\text{.refine} \]

by sepref

**sepref-definition** \( \text{fwi-impl} \) is

uncurry \( \text{fwi-impl'} :: [\lambda \ (-,k). \ k \leq n]_a \text{mtx-assn}^d \ast_a \text{node-assn}^k \rightarrow \text{mtx-assn} \)

**unfolding** \( \text{fwi-impl'}\)-def[abs-def] for-rec2-eq

**supply** \[ \text{[sepref-fr-rules]} = \text{fw-upd-impl}\text{.refine} \]

by sepref

**sepref-definition** \( \text{fwi-impl1} \) is

uncurry \( \text{fwi-impl'} :: [\lambda \ (-,k). \ k \leq n]_a \text{mtx-assn}^d \ast_a \text{node-assn}^k \rightarrow \text{mtx-assn} \)

**supply** \[ \text{[sepref-fr-rules]} = \text{fw-upd-impl1}\text{.refine} \]

by sepref

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

**Definition fw-spec**

\[
\text{fw-spec } n \ M \equiv \text{SPEC \ (} \lambda \ M'. \begin{align*}
& \text{if } (\exists \ i \leq n. \ M' \ i \ i < 0) \\
& \text{then } \neg \text{cyc-free } M \ n \\
& \text{else } \forall i \leq n. \forall j \leq n. \ M' \ i \ j = D \ M \ i \ j \ n \land \text{cyc-free } M \ n
\end{align*}
\]

**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:**

\[
\text{RETURN } (FW \ M \ n) \leq \text{fw-spec } n \ M
\]

**Unfolding** fw-spec-def cycle-free-diag-eqv

**Proof** (simp, safe, goal-cases)

**Case prem:** (1 i)

**With** fw-shortest-path[unfolded cycle-free-diag-eqv, 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-eqv]

**By** (force intro: fw-shortest-path[symmetric, unfolded cycle-free-diag-eqv])

**Next**

**Case** 3 then show ?case **Using** FW-neg-cycle-detect[unfolded cycle-free-diag-eqv]

**By** blast

**Qed**

**Definition**

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

**Definition**

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

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

**Lemma fw-impl'-correct:**

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

**Unfolding** fw-impl'-def[abs-def] **Using** fw'-spec fw-fw-spec
1.10.1 Main Result

This is one way to state that the \(fw\text{-}\text{impl}\) fulfills the specification \(fw\text{-}\text{spec}\).

**theorem** \(fw\text{-}\text{impl}\text{-}\text{correct}\):

\[
(fw\text{-}\text{impl} n, fw\text{-}\text{spec} n) \in (\text{mtx\text{-}\text{curry\text{-}\text{assn} n})^d} \to_a \text{mtx\text{-}\text{curry\text{-}\text{assn} n}
\]

**using** \(fw\text{-}\text{impl}\text{.}\text{refine}[FCOMP \text{fw}\text{-}\text{impl}\text{'-}\text{correct}[\text{THEN fun-relD, OF IdI}]].

An alternative version: a Hoare triple for total correctness.

**corollary**

\(<\text{mtx\text{-}\text{curry\text{-}\text{assn} n M Mi}>) \text{fw}\text{-}\text{impl} n Mi <\lambda Mi'. \exists A M'. \text{mtx\text{-}\text{curry\text{-}\text{assn} n M'}} Mi' ^+\>

\[
(\text{if } (\exists \ i \leq n. M' i i < 0) \\
\text{then } \neg \text{cyc-free } M n \\
\text{else } \forall i \leq n. \forall j \leq n. M' i j = D M i j n \land \text{cyc-free } M n) >_1
\]

**unfolding** \(\text{cyc-free-diag-equiv}\)

**by** (rule cons-rule(OF \text{-} \text{fw}\text{-}\text{impl}\text{-}\text{correct}[THEN hfrejD, THEN hn-refineD]])

\((\text{sep-auto simp: \text{fw}\text{-}\text{spec-def[unfolded cyc-free-diag-equiv]})} +

1.10.2 Alternative versions for Uncurried Matrices.

**definition** \(FW I' = \text{uncurry ooo FWI o curry}\)

**lemma** \(fwi\text{-}\text{impl}'\text{-}\text{refine-FWI}'\):

\[
(fwi\text{-}\text{impl}' n, RETURN oo \text{PR-CONST (\lambda M. FWI'} M n)) \in \text{Id} \to \text{Id} \to \text{(Id) nres-rel}
\]

**unfolding** \(fwi\text{-}\text{impl}'\text{-}\text{def[abs-def]} FWI-def[abs-def] FWI'-def\)

**using** \(\text{for-rec2-fwi}\)

**by** (force simp: \(\text{pw}\text{-}\text{le-iff} \text{pw-nres-rel-iff refine-pw-simps})

**lemmas** \(fwi\text{-}\text{impl}\text{-}\text{refine-FWI}'; = fwi\text{-}\text{impl}\text{.refine}[FCOMP fwi\text{-}\text{impl}'\text{-}\text{refine-FWI}']\)

**definition** \(FW' = \text{uncurry oo FW o curry}\)

**definition** \(FW'' n M = FW' M n\)

**lemma** \(fw\text{-}\text{impl}'\text{-}\text{refine-FW}''\):

\[
(fw\text{-}\text{impl}' n, RETURN o \text{PR-CONST (FW}'' n)) \in \text{Id} \to (\text{Id) nres-rel}
\]

**unfolding** \(fw\text{-}\text{impl}'\text{-}\text{def[abs-def]} FW''-def[abs-def] FW'\text{-}\text{def}\)

**using** \(\text{fw}'\text{-}\text{spec}\)

**by** (force simp: \(\text{pw}\text{-}\text{le-iff} \text{pw-nres-rel-iff refine-pw-simps})

**lemmas** \(fw\text{-}\text{impl}\text{-}\text{refine-FW}'' = fw\text{-}\text{impl}\text{.refine}[FCOMP fw\text{-}\text{impl}'\text{-}\text{refine-FW}''\]

**lemmas** \(fw\text{-}\text{impl1-refine-FW}'' = fw\text{-}\text{impl1}\text{.refine}[FCOMP fw\text{-}\text{impl}'\text{-}\text{refine-FW}''\]

59
References

