

The Floyd-Warshall Algorithm for Shortest Paths

Simon Wimmer and Peter Lammich

August 26, 2024

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	6
1.4	Result Under The Absence of Negative Cycles	9
1.5	Definition of Shortest Paths	12
1.6	Intermezzo: Equivalent Characterizations of Cycle-Freeness	15
1.7	Result Under the Presence of Negative Cycles	17
1.8	More on Canonical Matrices	18
1.9	Additional Theorems	19
1.10	Refinement to Efficient Imperative Code	23

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

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

1.1 Introduction

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

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

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

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

1.2 Preliminaries

1.2.1 Cycles in Lists

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

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

where

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

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

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

lemma $\text{cnt-rev}: \text{cnt } x \text{ } (\text{rev } xs) = \text{cnt } x \text{ } xs$ $\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma $\text{remove-cycles-self}:$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *remove-cycles-cycles*:

$\exists xs\ as.\ as\ @\ concat\ (map\ (\lambda\ xs.\ x\ \#\ xs)\ xs)\ @\ remove-cycles\ xs\ x\ ys$
 $=\ xs\ \wedge\ x\ \notin\ set\ as$
if $x \in set\ xs$
 $\langle proof \rangle$

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

where

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

lemma *start-remove-decomp*:

$x \in set\ xs \implies \exists\ as\ bs.\ xs = as\ @\ x\ \#\ bs \wedge\ start-remove\ xs\ x\ ys = rev\ ys$
 $@\ as\ @\ remove-cycles\ bs\ x\ [x]$
 $\langle proof \rangle$

lemma *start-remove-removes*: cnt x (start-remove xs x ys) \leq Suc (cnt x ys)
 $\langle proof \rangle$

lemma *start-remove-id[simp]*: $x \notin set\ xs \implies start-remove\ xs\ x\ ys = rev\ ys$
 $@\ xs$
 $\langle proof \rangle$

lemma *start-remove-cnt-id*:

$x \neq y \implies cnt\ y\ (start-remove\ xs\ x\ ys) \leq cnt\ y\ ys + cnt\ y\ xs$
 $\langle proof \rangle$

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*: cnt y (remove-all-cycles xs ys) \leq max 1 (cnt
y ys)
 $\langle proof \rangle$

lemma *cnt-remove-all-cycles*: $x \in set\ xs \implies cnt\ x\ (remove-all-cycles\ xs\ ys)$
 ≤ 1
 $\langle proof \rangle$

lemma *cnt-mono*:

$cnt\ a\ (b\ \#\ xs) \leq cnt\ a\ (b\ \#\ c\ \#\ xs)$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *remove-cycles-subs*:

$set\ (remove-cycles\ xs\ x\ ys) \subseteq set\ xs \cup set\ ys$
 $\langle proof \rangle$

lemma *start-remove-subs*:

$set\ (start-remove\ xs\ x\ ys) \subseteq set\ xs \cup set\ ys$
 $\langle proof \rangle$

lemma *remove-all-cycles-subs*:

$set\ (remove-all-cycles\ xs\ ys) \subseteq set\ ys$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

definition

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

definition

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

lemma *remove-all-distinct*:

$distinct\ xs \implies distinct\ (x\ \#\ remove-all\ x\ xs)$
 $\langle proof \rangle$

lemma *remove-all-removes*:

$x \notin set\ (remove-all\ x\ xs)$
 $\langle proof \rangle$

lemma *remove-all-subs*:

$set\ (remove-all\ x\ xs) \subseteq set\ xs$

<proof>

lemma *remove-all-rev-distinct: distinct xs \implies distinct (x # remove-all-rev x xs)*

<proof>

lemma *remove-all-rev-removes: x \notin set (remove-all-rev x xs)*

<proof>

lemma *remove-all-rev-subst: set (remove-all-rev x xs) \subseteq set xs*

<proof>

abbreviation *rem-cycles i j xs \equiv remove-all i (remove-all-rev j (remove-all-cycles xs xs))*

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

<proof>

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

<proof>

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

<proof>

lemma *rem-cycles-subst: set (rem-cycles i j xs) \subseteq set xs*

<proof>

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 <proof>*

end

subclass (**in** *linordered-ab-group-add*) *linordered-ab-monoid-add <proof>*

context *linordered-ab-monoid-add*

begin

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

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

where

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

definition $\text{fw-upd} :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$ **where**

$$\text{fw-upd } m \ k \ i \ j \equiv \text{upd } m \ i \ j \ (\min (m \ i \ j) (m \ i \ k + m \ k \ j))$$

Recursive version of the two inner loops.

fun $\text{fwi} :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$ **where**

$$\begin{aligned} \text{fwi } m \ n \ k \ 0 \quad 0 &= \text{fw-upd } m \ k \ 0 \ 0 \ | \\ \text{fwi } m \ n \ k \ (\text{Suc } i) \ 0 &= \text{fw-upd } (\text{fwi } m \ n \ k \ i \ n) \ k \ (\text{Suc } i) \ 0 \ | \\ \text{fwi } m \ n \ k \ i \quad (\text{Suc } j) &= \text{fw-upd } (\text{fwi } m \ n \ k \ i \ j) \ k \ i \ (\text{Suc } j) \end{aligned}$$

Recursive version of the full algorithm.

fun $\text{fw} :: 'a \text{ mat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$ **where**

$$\begin{aligned} \text{fw } m \ n \ 0 &= \text{fwi } m \ n \ 0 \ n \ n \ | \\ \text{fw } m \ n \ (\text{Suc } k) &= \text{fwi } (\text{fw } m \ n \ k) \ n \ (\text{Suc } k) \ n \ n \end{aligned}$$

1.3.2 Elementary Properties

lemma fw-upd-mono :

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

$\langle \text{proof} \rangle$

lemma $\text{fw-upd-out-of-bounds1}$:

assumes $i' > i$
shows $(\text{fw-upd } M \ k \ i \ j) \ i' \ j' = M \ i' \ j'$
 $\langle \text{proof} \rangle$

lemma $\text{fw-upd-out-of-bounds2}$:

assumes $j' > j$
shows $(\text{fw-upd } M \ k \ i \ j) \ i' \ j' = M \ i' \ j'$
 $\langle \text{proof} \rangle$

lemma $\text{fwi-out-of-bounds1}$:

assumes $i' > n \ i \leq n$
shows $(\text{fwi } M \ n \ k \ i \ j) \ i' \ j' = M \ i' \ j'$
 $\langle \text{proof} \rangle$

lemma fw-out-of-bounds1 :

assumes $i' > n$
shows $(fw\ M\ n\ k)\ i'\ j' = M\ i'\ j'$
 $\langle proof \rangle$

lemma *fwi-out-of-bounds2*:
assumes $j' > n\ j \leq n$
shows $(fwi\ M\ n\ k\ i\ j)\ i'\ j' = M\ i'\ j'$
 $\langle proof \rangle$

lemma *fw-out-of-bounds2*:
assumes $j' > n$
shows $(fw\ M\ n\ k)\ i'\ j' = M\ i'\ j'$
 $\langle proof \rangle$

lemma *fwi-invariant-aux-1*:
 $j'' \leq j \implies fwi\ m\ n\ k\ i\ j\ i'\ j' \leq fwi\ m\ n\ k\ i\ j''\ i'\ j'$
 $\langle proof \rangle$

lemma *fwi-invariant*:
 $j \leq n \implies i'' \leq i \implies j'' \leq j$
 $\implies fwi\ m\ n\ k\ i\ j\ i'\ j' \leq fwi\ m\ n\ k\ i''\ j''\ i'\ j'$
 $\langle proof \rangle$

lemma *single-row-inv*:
 $j' < j \implies fwi\ m\ n\ k\ i'\ j\ i'\ j' = fwi\ m\ n\ k\ i'\ j'\ i'\ j'$
 $\langle proof \rangle$

lemma *single-iteration-inv'*:
 $i' < i \implies j' \leq n \implies fwi\ m\ n\ k\ i\ j\ i'\ j' = fwi\ m\ n\ k\ i'\ j'\ i'\ j'$
 $\langle proof \rangle$

lemma *single-iteration-inv*:
 $i' \leq i \implies j' \leq j \implies j \leq n \implies fwi\ m\ n\ k\ i\ j\ i'\ j' = fwi\ m\ n\ k\ i'\ j'\ i'\ j'$
 $\langle proof \rangle$

lemma *fwi-innermost-id*:
 $i' < i \implies fwi\ m\ n\ k\ i'\ j'\ i\ j = m\ i\ j$
 $\langle proof \rangle$

lemma *fwi-middle-id*:
 $j' < j \implies i' \leq i \implies fwi\ m\ n\ k\ i'\ j'\ i\ j = m\ i\ j$
 $\langle proof \rangle$

lemma *fwi-outermost-mono*:

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

lemma *fwi-mono*:

$\text{fwi } m \ n \ k \ i' \ j' \ i \ j \leq m \ i \ j$ **if** $i \leq n \ j \leq n$
 ⟨proof⟩

lemma *Suc-innermost-mono*:

$i \leq n \implies j \leq n \implies \text{fw } m \ n \ (\text{Suc } k) \ i \ j \leq \text{fw } m \ n \ k \ i \ j$
 ⟨proof⟩

lemma *fw-mono*:

$i \leq n \implies j \leq n \implies \text{fw } m \ n \ k \ i \ j \leq m \ i \ j$
 ⟨proof⟩

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

lemma *fwi-step*:

$m \ k \ k \geq 0 \implies i \leq n \implies j \leq n \implies k \leq n \implies \text{fwi } m \ n \ k \ i \ j \ i \ j = \min$
 $(m \ i \ j) (m \ i \ k + m \ k \ j)$
 ⟨proof⟩

1.4 Result Under The Absence of Negative Cycles

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

1.4.1 Length of Paths

fun *len* :: 'a mat ⇒ nat ⇒ nat ⇒ nat list ⇒ 'a **where**

$\text{len } m \ u \ v \ [] = m \ u \ v \ |$

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

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

⟨proof⟩

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

⟨proof⟩

1.4.2 Canonicity

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

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

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

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

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

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

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

lemma *canonical-subs-len*:

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

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

lemma *fwi-step'*:

$\text{fwi } m \ n \ k \ i' \ j' \ i \ j = \min (m \ i \ j) (m \ i \ k + m \ k \ j)$ **if**
 $m \ k \ k \geq 0 \ i' \leq n \ j' \leq n \ k \leq n \ i \leq i' \ j \leq j'$
 $\langle \text{proof} \rangle$

An invocation of fwi extends canonical forms.

lemma *fwi-canonical-extend*:

$\text{canonical-subs } n \ (I \cup \{k\}) (\text{fwi } m \ n \ k \ n \ n)$ **if**
 $\text{canonical-subs } n \ I \ m \ I \subseteq \{0..n\} \ 0 \leq m \ k \ k \ k \leq n$
 $\langle \text{proof} \rangle$

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

lemma *fwi-cyc-free-diag*:

fwi m n k n n i i ≥ 0 **if**
cyc-free Subs n I m 0 ≤ m k k k ≤ n k ∈ I i ≤ n
 ⟨proof⟩

lemma *cyc-free Subs-diag*:

m i i ≥ 0 **if** *cyc-free Subs n I m i ≤ n*
 ⟨proof⟩

lemma *fwi-cyc-free Subs'*:

cyc-free Subs n (I ∪ {k}) (fwi m n k n n) **if**
cyc-free Subs n I m canonical Subs n I m I ⊆ {0..n} k ≤ n
 $\forall i \leq n. \text{fwi } m \text{ n } k \text{ n } n \text{ i } i \geq 0$
 ⟨proof⟩

lemma *fwi-cyc-free Subs*:

cyc-free Subs n (I ∪ {k}) (fwi m n k n n) **if**
cyc-free Subs n (I ∪ {k}) m canonical Subs n I m I ⊆ {0..n} k ≤ n
 ⟨proof⟩

lemma *canonical Subs-empty [simp]*:

canonical Subs n {} m
 ⟨proof⟩

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

$\exists i \leq n. \exists xs. \text{set } xs \subseteq \{0..k\} \wedge \text{len } m \text{ i } i \text{ xs} < 0$ **if** *fwi m n k n n i i < 0*
 $i \leq n \ k \leq n$
 ⟨proof⟩

fwi preserves the length of paths.

lemma *fwi-len*:

$\exists ys. \text{set } ys \subseteq \text{set } xs \cup \{k\} \wedge \text{len } (fwi \ m \ n \ k \ n \ n) \ i \ j \ xs = \text{len } m \ i \ j \ ys$
if $i \leq n \ j \leq n \ k \leq n \ m \ k \ k \geq 0 \ \text{set } xs \subseteq \{0..n\}$
 ⟨proof⟩

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

$\exists i \leq n. \exists ys. \text{set } ys \subseteq \text{set } xs \cup \{k\} \wedge \text{len } m \ i \ i \ ys < 0$ **if**
 $\text{len } (fwi \ m \ n \ k \ n \ n) \ i \ i \ xs < 0 \ i \leq n \ k \leq n \ \text{set } xs \subseteq \{0..n\}$
 ⟨proof⟩

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

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

$\exists i \leq n. \exists ys. \text{set } ys \subseteq \text{set } xs \cup \{0..k\} \wedge \text{len } m \text{ } i \text{ } ys < 0$ **if**
 $\text{len } (fw \ m \ n \ k) \ i \ i \ xs < 0 \ i \leq n \ k \leq n \ \text{set } xs \subseteq \{0..n\}$
 ⟨proof⟩

Main theorem under the absence of negative cycles.

theorem *fw-correct*:

canonical-subs $n \ \{0..k\} \ (fw \ m \ n \ k) \wedge \text{cyc-free-subs} \ n \ \{0..k\} \ (fw \ m \ n \ k)$
if *cyc-free-subs* $n \ \{0..k\} \ m \ k \leq n$
 ⟨proof⟩

lemmas *fw-canonical-subs* = *fw-correct*[*THEN* *conjunct1*]

lemmas *fw-cyc-free-subs* = *fw-correct*[*THEN* *conjunct2*]

lemmas *cyc-free-diag* = *cyc-free-subs-diag*

1.5 Definition of Shortest Paths

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

definition *D* :: 'a mat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a **where**

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

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

⟨proof⟩

lemma *finite-distinct*: *finite* $s \Longrightarrow \text{finite } \{xs . \text{set } xs \subseteq s \wedge \text{distinct } xs\}$

⟨proof⟩

lemma *D-base-finite*:

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

⟨proof⟩

lemma *D-base-finite'*:

finite $\{\text{len } m \ i \ j \ xs \mid xs. \text{set } xs \subseteq \{0..k\} \wedge \text{distinct } (i \# j \# xs)\}$

⟨proof⟩

lemma *D-base-finite''*:

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

⟨proof⟩

definition *cycle-free* :: 'a mat \Rightarrow nat \Rightarrow bool **where**

$cycle\text{-}free\ m\ n \equiv \forall\ i\ xs.\ i \leq n \wedge set\ xs \subseteq \{0..n\} \longrightarrow$
 $(\forall\ j.\ j \leq n \longrightarrow len\ m\ i\ j\ (rem\text{-}cycles\ i\ j\ xs) \leq len\ m\ i\ j\ xs) \wedge len\ m\ i\ i\ xs \geq 0$

lemma *D-eqI*:

fixes $m\ n\ i\ j\ k$
defines $A \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\}\}$
defines $A\text{-}distinct \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
assumes $cycle\text{-}free\ m\ n\ i \leq n\ j \leq n\ k \leq n\ (\bigwedge y.\ y \in A\text{-}distinct \implies x \leq y)\ x \in A$
shows $D\ m\ i\ j\ k = x$ *<proof>*

lemma *D-base-not-empty*:

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

lemma *Min-elem-dest*: $finite\ A \implies A \neq \{\} \implies x = Min\ A \implies x \in A$
<proof>

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

$x \in \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..Suc\ k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *cycle-free-dest*:

$cycle\text{-}free\ m\ n \implies i \leq n \implies j \leq n \implies set\ xs \subseteq \{0..n\}$
 $\implies len\ m\ i\ j\ (rem\text{-}cycles\ i\ j\ xs) \leq len\ m\ i\ j\ xs$

<proof>

definition *cycle-free-up-to* :: $'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

$cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n \equiv \forall\ i\ xs.\ i \leq n \wedge set\ xs \subseteq \{0..k\} \longrightarrow$

$(\forall j. j \leq n \longrightarrow \text{len } m \ i \ j \ (\text{rem-cycles } i \ j \ xs) \leq \text{len } m \ i \ j \ xs) \wedge \text{len } m \ i \ i \ xs \geq 0$

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

$i \leq n \implies \text{set } xs \subseteq \{0..k\} \implies \text{cycle-free-up-to } m \ k \ n \implies \text{len } m \ i \ i \ xs \geq 0$
 $\langle \text{proof} \rangle$

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

assumes *cycle-free-up-to* $m \ k \ n \ i \leq n$
shows $m \ i \ i \geq 0$
 $\langle \text{proof} \rangle$

lemma *D-eqI2*:

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

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* $n \ \{0..k\} \ m \ i \leq n \ j \leq n \ k \leq n$
shows $D \ m \ i \ j \ k = m \ i \ j$
 $\langle \text{proof} \rangle$

theorem *fw-subs-len*:

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

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'*:

$\exists xs. \text{set } xs \subseteq \{k\} \wedge fwi \ m \ n \ k \ i' \ j' \ i \ j = \text{len } m \ i \ j \ xs$ **if**
 $m \ k \ k \geq 0 \ i' \leq n \ j' \leq n \ k \leq n \ i \leq i' \ j \leq j'$
 $\langle \text{proof} \rangle$

The same result for fw .

lemma *fw-len*:

$\exists xs. \text{set } xs \subseteq \{0..k\} \wedge fw\ m\ n\ k\ i\ j = len\ m\ i\ j\ xs$ **if**
 $cyc\text{-free}\text{-subs}\ n\ \{0..k\}\ m\ i \leq n\ j \leq n\ k \leq n$
 $\langle proof \rangle$

1.6 Intermezzo: Equivalent Characterizations of Cycle-Freeness

1.6.1 Shortening Negative Cycles

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

fixes $i\ xs\ ys$
defines $xs' \equiv i \# ys$
assumes $i \notin \text{set } ys$
assumes $i \in \text{set } xs$
assumes $xs = as @ \text{concat} (\text{map} ((\#)\ i)\ xss) @ xs'$
assumes $len\ m\ i\ j\ ys > len\ m\ i\ j\ xs$
shows $\exists ys. \text{set } ys \subseteq \text{set } xs \wedge len\ m\ i\ i\ ys < 0$ $\langle proof \rangle$

lemma *add-lt-neutral*: $a + b < b \implies a < 0$

$\langle proof \rangle$

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

fixes $j\ xs\ ys$
assumes $j \notin \text{set } ys$
assumes $j \in \text{set } xs$
assumes $xs = ys @ j \# \text{concat} (\text{map} (\lambda xs. xs @ [j])\ xss) @ as$
assumes $len\ m\ i\ j\ ys > len\ m\ i\ j\ xs$
shows $\exists ys. \text{set } ys \subseteq \text{set } xs \wedge len\ m\ j\ j\ ys < 0$ $\langle proof \rangle$

lemma *add-le-impl*: $a + b < a + c \implies b < c$

$\langle proof \rangle$

lemma *start-remove-neg-cycles*:

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

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

$len\ m\ i\ j\ (\text{remove}\text{-all}\text{-cycles}\ ys\ xs) > len\ m\ i\ j\ xs$
 $\implies \exists ys\ k. \text{set } ys \subseteq \text{set } xs \wedge k \in \text{set } xs \wedge len\ m\ k\ k\ ys < 0$
 $\langle proof \rangle$

lemma *concat-map-cons-rev*:

$rev (concat (map ((\#) j) xss)) = concat (map (\lambda xs. xs @ [j]) (rev (map rev xss)))$
 <proof>

lemma *negative-cycle-dest*: $len\ m\ i\ j\ (rem\ cycles\ i\ j\ xs) > len\ m\ i\ j\ xs$
 $\implies \exists\ i'\ ys. len\ m\ i'\ i'\ ys < 0 \wedge set\ ys \subseteq set\ xs \wedge i' \in set\ (i\ \# j\ \# xs)$
 <proof>

1.6.2 Cycle-Freeness

lemma *cycle-free-alt-def*:
 $cycle\ free\ M\ n \iff cycle\ free\ up\ to\ M\ n\ n$
 <proof>

lemma *negative-cycle-dest-diag*:
 $\neg\ cycle\ free\ up\ to\ m\ k\ n \implies k \leq n \implies \exists\ i\ xs. i \leq n \wedge set\ xs \subseteq \{0..k\}$
 $\wedge len\ m\ i\ i\ xs < 0$
 <proof>

lemma *negative-cycle-dest-diag'*:
 $\neg\ cycle\ free\ m\ n \implies \exists\ i\ xs. i \leq n \wedge set\ xs \subseteq \{0..n\} \wedge len\ m\ i\ i\ xs < 0$
 <proof>

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

lemma *cycle-free-diag-intro*:
 $cyc\ free\ m\ n \implies cycle\ free\ m\ n$
 <proof>

lemma *cycle-free-diag-equiv*:
 $cyc\ free\ m\ n \iff cycle\ free\ m\ n$ <proof>

lemma *cycle-free-diag-dest*:
 $cycle\ free\ m\ n \implies cyc\ free\ m\ n$
 <proof>

lemma *cycle-free-upto-diag-equiv*:
 $cycle\ free\ up\ to\ m\ k\ n \iff cyc\ free\ subs\ n\ \{0..k\}\ m$ **if** $k \leq n$
 <proof>

theorem *fw-shortest-path-up-to*:
 $D\ m\ i\ j\ k = fw\ m\ n\ k\ i\ j$ **if** $cyc\ free\ subs\ n\ \{0..k\}\ m$ $i \leq n\ j \leq n\ k \leq n$

<proof>

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

lemma

$cyc\text{-}free\ m\ n \longleftrightarrow cyc\text{-}free\text{-}subs\ n\ \{0..n\}\ m$ *<proof>*

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

$cycle\text{-}free\ m\ n \implies k \leq n \implies cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n$
<proof>

lemma *cycle-free-diag:*

$cycle\text{-}free\ m\ n \implies i \leq n \implies 0 \leq m\ i\ i$
<proof>

corollary *fw-shortest-path:*

$cycle\text{-}free\ m\ n \implies i \leq n \implies j \leq n \implies k \leq n \implies D\ m\ i\ j\ k = fw\ m\ n\ k\ i$
<proof>

corollary *fw-shortest:*

assumes $cycle\text{-}free\ m\ n\ i \leq n\ j \leq n\ k \leq n$
shows $fw\ m\ n\ n\ i\ j \leq fw\ m\ n\ n\ i\ k + fw\ m\ n\ n\ k\ j$
<proof>

1.7 Result Under the Presence of Negative Cycles

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

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

lemma *D-not-diag-le:*

$(x :: 'a) \in \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
 $\implies D\ m\ i\ j\ k \leq x$ *<proof>*

lemma *D-not-diag-le':* $set\ xs \subseteq \{0..k\} \implies i \notin set\ xs \implies j \notin set\ xs \implies distinct\ xs$
 $\implies D\ m\ i\ j\ k \leq len\ m\ i\ j\ xs$ *<proof>*

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

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

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

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

Monotonicity with respect to k .

lemma *fw-invariant*:

$k' \leq k \implies i \leq n \implies j \leq n \implies k \leq n \implies fw\ m\ n\ k\ i\ j \leq fw\ m\ n\ k'\ i\ j$
 ⟨proof⟩

lemma *negative-len-shortest*:

$length\ xs = n \implies len\ m\ i\ i\ xs < 0$
 $\implies \exists j\ ys. distinct\ (j\ \# \ ys) \wedge len\ m\ j\ j\ ys < 0 \wedge j \in set\ (i\ \# \ xs) \wedge set\ ys \subseteq set\ xs$
 ⟨proof⟩

lemma *fw-upd-leI*:

$fw\ upd\ m'\ k\ i\ j\ i\ j \leq fw\ upd\ m\ k\ i\ j\ i\ j$ **if**
 $m'\ i\ k \leq m\ i\ k\ m'\ k\ j \leq m\ k\ j\ m'\ i\ j \leq m\ i\ j$
 ⟨proof⟩

lemma *fwi-fw-upd-mono*:

$fwi\ m\ n\ k\ i\ j\ i\ j \leq fw\ upd\ m\ k\ i\ j\ i\ j$ **if** $k \leq n\ i \leq n\ j \leq n$
 ⟨proof⟩

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

theorem *FW-neg-cycle-detect*:

$\neg\ cyc\ free\ m\ n \implies \exists i \leq n. fw\ m\ n\ n\ i\ i < 0$
 ⟨proof⟩

end

1.8 More on Canonical Matrices

abbreviation

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

lemma *canonical-alt-def*:

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

lemma *fw-canonical*:

canonical $(fw\ m\ n\ n)\ n$ **if** *cyc-free* $m\ n$
 ⟨proof⟩

lemma *canonical-len*:

canonical $M\ n \implies i \leq n \implies j \leq n \implies \text{set } xs \subseteq \{0..n\} \implies M\ i\ j \leq \text{len } M\ i\ j\ xs$
 ⟨proof⟩

1.9 Additional Theorems

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

cycle-free $m\ n$
 $\implies \forall i \leq n. \forall j \leq n. D\ m\ i\ j\ n = m'\ i\ j \implies i \leq n \implies j \leq n \implies \text{set } xs \subseteq \{0..n\}$
 $\implies \exists ys. \text{set } ys \subseteq \{0..n\} \wedge \text{len } m'\ i\ j\ xs = \text{len } m\ i\ j\ ys$
 ⟨proof⟩

lemma *D-cyc-free-preservation*:

cyc-free $m\ n \implies \forall i \leq n. \forall j \leq n. D\ m\ i\ j\ n = m'\ i\ j \implies \text{cyc-free } m'\ n$
 ⟨proof⟩

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

lemma *FW-out-of-bounds1*:

assumes $i > n$
shows $(FW\ M\ n)\ i\ j = M\ i\ j$
 ⟨proof⟩

lemma *FW-out-of-bounds2*:

assumes $j > n$
shows $(FW\ M\ n)\ i\ j = M\ i\ j$
 ⟨proof⟩

lemma *FW-cyc-free-preservation*:

cyc-free $m\ n \implies \text{cyc-free } (FW\ m\ n)\ n$
 ⟨proof⟩

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

cyc-free $m\ n \implies i \leq n \implies m\ i\ i \geq 0$
 ⟨proof⟩

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>

lemma *FW-fixed-preservation:*

fixes $M :: ('a::\text{linordered-ab-monoid-add}) \text{ mat}$
assumes $A: i \leq n \implies M 0 i + M i 0 = 0$ *canonical* $(FW M n) n$ *cyc-free*
 $(FW M n) n$
shows $FW M n 0 i + FW M n i 0 = 0$ *<proof>*

lemma *diag-cyc-free-neutral:*

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

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

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

lemma *fw-upd-canonical-id:*

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

lemma *fwi-canonical-id:*

$\text{fwi } M n k i j = M$ **if** $\text{canonical-subs } n \{k\} M i \leq n j \leq n k \leq n$
<proof>

lemma *fw-canonical-id:*

$\text{fw } M n k = M$ **if** $\text{canonical-subs } n \{0..k\} M k \leq n$
<proof>

lemmas $FW\text{-canonical-id} = \text{fw-canonical-id}[\text{OF - order.refl, unfolded canonical-alt-def[symmetric]]$

definition $FWI M n k \equiv \text{fwi } M n k n n$

The characteristic property of *fwi*.

theorem *fwi-characteristic:*

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

end

```

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

context
begin

private definition for-comb where
  for-comb f a 0 n = nfoldli [0..n + 1] (λ x. True) (λ k a. (f a k)) a 0

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
  ⟨proof⟩ definition for-rec2' where
  for-rec2' f a n i j =
    (if i = 0 then RETURN a else for-rec (λ a 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
  ⟨proof⟩

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 $for-rec (\lambda a k. for-rec2' (\lambda a. f a k) a n$
 $n) a (k - 1)$)
 $\gg (\lambda a. for-rec2' (\lambda a. f a k) a n i j)$

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

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

<proof> **lemma** *for-rec2'-for-rec*:

$for-rec2' f a n n n =$

$for-rec (\lambda a i. for-rec (\lambda a. f a i) a n) a n$

<proof> **lemma** *for-rec3'-for-rec*:

$for-rec3' f a n n n n =$

$for-rec (\lambda a k. for-rec (\lambda a i. for-rec (\lambda a. f a k i) a n) a n) a n$

<proof>

theorem *for-rec-eq*:

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

<proof>

theorem *for-rec2-eq*:

$for-rec2 f a n n n =$

$nfoldli [0..<n + 1] (\lambda x. True)$

$(\lambda i. nfoldli [0..<n + 1] (\lambda x. True) (\lambda j a. f a i j)) a$

<proof>

theorem *for-rec3-eq*:

$for-rec3 f a n n n n =$

$nfoldli [0..<n + 1] (\lambda x. True)$

$(\lambda k. nfoldli [0..<n + 1] (\lambda x. True)$

$(\lambda i. nfoldli [0..<n + 1] (\lambda x. True) (\lambda j a. f a k i j)))$

a

<proof>

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
)
 ⟨proof⟩

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)}
 ⟨proof⟩

lemma

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

lemma *fw-upd'-spec:*

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

lemma *for-rec2-fwi:*

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

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

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

lemma *fw'-spec:*

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

context

fixes $n :: nat$

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

begin

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

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

abbreviation $node\text{-}assn \equiv nat\text{-}assn$

abbreviation $mtx\text{-}assn \equiv asmtx\text{-}assn (Suc n) id\text{-}assn\::('a mtx \Rightarrow-)$

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

$uncurry2 (uncurry fw\text{-}upd') ::$
 $[\lambda (((-,k),i),j). k \leq n \wedge i \leq n \wedge j \leq n]_a mtx\text{-}assn^d *_a node\text{-}assn^k *_a$
 $node\text{-}assn^k *_a node\text{-}assn^k$
 $\rightarrow mtx\text{-}assn$
 $\langle proof \rangle$

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

$uncurry2 (uncurry fw\text{-}upd') ::$

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

$$\text{node-assn}^k *_a \text{node-assn}^k$$

$$\rightarrow \text{mtx-assn}$$

$$\langle \text{proof} \rangle$$

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

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

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

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

begin

sepref-definition fw-impl **is**

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

$\langle \text{proof} \rangle$

sepref-definition fw-impl1 **is**

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

$\langle \text{proof} \rangle$

sepref-definition fwi-impl **is**

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

$\langle \text{proof} \rangle$

sepref-definition fwi-impl1 **is**

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

$\langle \text{proof} \rangle$

end

end

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

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

definition fw-spec **where**

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

lemma *D-diag-nonnegI*:
assumes *cycle-free* $M \ n \ i \leq n$
shows $D \ M \ i \ i \ n \geq 0$
 $\langle \text{proof} \rangle$

lemma *fw-fw-spec*:
 $RETURN (FW \ M \ n) \leq fw\text{-spec } n \ M$
 $\langle \text{proof} \rangle$

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

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

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

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

1.10.1 Main Result

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

theorem *fw-impl-correct*:
 $(fw\text{-impl } n, fw\text{-spec } n) \in (mtx\text{-curry-assn } n)^d \rightarrow_a \text{mtx-curry-assn } n$
 $\langle \text{proof} \rangle$

An alternative version: a Hoare triple for total correctness.

corollary
 $\langle \text{mtx-curry-assn } n \ M \ Mi \rangle \text{fw-impl } n \ Mi \langle \lambda Mi'. \exists_A M'. \text{mtx-curry-assn}$
 $n \ M' \ Mi' \ * \ \uparrow$
 $(\text{if } (\exists i \leq n. M' \ i \ i < 0)$
 $\text{then } \neg \text{cyc-free } M \ n$
 $\text{else } \forall i \leq n. \forall j \leq n. M' \ i \ j = D \ M \ i \ j \ n \wedge \text{cyc-free } M \ n) \rangle_t$
 $\langle \text{proof} \rangle$

1.10.2 Alternative versions for Uncurried Matrices.

definition $FWI' = \text{uncurry } \circ \circ FWI \circ \text{curry}$

lemma $\text{fwi-impl}'\text{-refine-FWI}'$:

$(\text{fwi-impl}' n, \text{RETURN } \circ \circ PR\text{-CONST } (\lambda M. FWI' M n)) \in Id \rightarrow Id \rightarrow$
 $\langle Id \rangle \text{ nres-rel}$
 $\langle \text{proof} \rangle$

lemmas $\text{fwi-impl-refine-FWI}' = \text{fwi-impl.refine}[FCOMP \text{fwi-impl}'\text{-refine-FWI}']$

definition $FW' = \text{uncurry } \circ \circ FW \circ \text{curry}$

definition $FW'' n M = FW' M n$

lemma $\text{fw-impl}'\text{-refine-FW}''$:

$(\text{fw-impl}' n, \text{RETURN } \circ PR\text{-CONST } (FW'' n)) \in Id \rightarrow \langle Id \rangle \text{ nres-rel}$
 $\langle \text{proof} \rangle$

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

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

end

References

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