The Floyd-Warshall Algorithm for Shortest Paths

Simon Wimmer and Peter Lammich

December 7, 2022

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

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 $\{0 \ldots k+1\}$, 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 we 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 *cnt x xs* ≡ *length* (*filter* (λ*y*. *x* = *y*) *xs*)

fun *remove-cycles* :: *'a list* ⇒ *'a* ⇒ *'a list* ⇒ *'a list*
where
remove-cycles [] - *acc* = *rev acc* |
remove-cycles (*x*#*xs*) *y acc* =
(*if x* = *y then remove-cycles xs y* [*x*] *else remove-cycles xs y* (*x*#*acc*))

lemma *cnt-rev*: *cnt x* (*rev xs*) = *cnt x xs* ⟨*proof*⟩

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

lemma *remove-cycles-removes*: *cnt x* (*remove-cycles xs x ys*) ≤ *max 1* (*cnt x ys*)
⟨*proof*⟩

lemma *remove-cycles-id*: *x* ∉ *set xs* ⟹ *remove-cycles xs x ys* = *rev ys* @ *xs*
⟨*proof*⟩

lemma *remove-cycles-cnt-id*:
x ≠ *y* ⟹ *cnt y* (*remove-cycles xs x ys*) ≤ *cnt y ys* + *cnt y xs*
⟨*proof*⟩

lemma *remove-cycles-ends-cycle*: *remove-cycles xs x ys* ≠ *rev ys* @ *xs* ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *remove-cycles-begins-with*: *x* ∈ *set xs* ⟹ ∃ *zs*. *remove-cycles xs x ys* = *x* # *zs* ∧ *x* ∉ *set zs*
⟨*proof*⟩

lemma *remove-cycles-self*:
x ∈ *set xs* ⟹ *remove-cycles* (*remove-cycles xs x ys*) *x zs* = *remove-cycles xs x ys*
⟨*proof*⟩

lemma *remove-cycles-one*: *remove-cycles* (*as* @ *x* # *xs*) *x ys* = *remove-cycles* (*x*#*xs*) *x ys*
⟨*proof*⟩

lemma *remove-cycles-cycles*:
  $\exists$ *xxs as*. *as* @ *concat* (*map* (λ *xs*. *x* # *xs*) *xxs*) @ *remove-cycles xs x ys* = *xs* $\wedge$ *x* $\notin$ *set as*
  if *x* $\in$ *set xs*
⟨*proof*⟩

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* $\Longrightarrow$ $\exists$ *as bs*. *xs* = *as* @ *x* # *bs* $\wedge$ *start-remove xs x ys* = *rev ys* @ *as* @ *remove-cycles bs x* [*x*]
⟨*proof*⟩

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

lemma *start-remove-id*[*simp*]: *x* $\notin$ *set xs* $\Longrightarrow$ *start-remove xs x ys* = *rev ys* @ *xs*
⟨*proof*⟩

lemma *start-remove-cnt-id*:
  x $\neq$ *y* $\Longrightarrow$ *cnt y* (*start-remove xs x ys*) $\leq$ *cnt y ys* + *cnt y xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cnt-remove-all-cycles*: *x* $\in$ *set xs* $\Longrightarrow$ *cnt x* (*remove-all-cycles xs ys*) $\leq$ *1*
⟨*proof*⟩

lemma *cnt-mono*:
cnt a (b # xs) ≤ cnt a (b # c # xs)
⟨*proof*⟩

lemma *cnt-distinct-intro*: *∀ x ∈ set xs. cnt x xs ≤ 1 ⟹ distinct xs*
⟨*proof*⟩

lemma *remove-cycles-subs*:
set (remove-cycles xs x ys) ⊆ set xs ∪ set ys
⟨*proof*⟩

lemma *start-remove-subs*:
set (start-remove xs x ys) ⊆ set xs ∪ set ys
⟨*proof*⟩

lemma *remove-all-cycles-subs*:
set (remove-all-cycles xs ys) ⊆ set ys
⟨*proof*⟩

lemma *remove-all-cycles-distinct*: *set ys ⊆ set xs ⟹ distinct (remove-all-cycles xs ys)*
⟨*proof*⟩

lemma *distinct-remove-cycles-inv*: *distinct (xs @ ys) ⟹ distinct (remove-cycles xs x ys)*
⟨*proof*⟩

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

definition
remove-all-rev x xs = (if x ∈ set xs then rev (tl (remove-cycles (rev xs) x [])) else xs)

lemma *remove-all-distinct*:
distinct xs ⟹ distinct (x # remove-all x xs)
⟨*proof*⟩

lemma *remove-all-removes*:
x ∉ set (remove-all x xs)
⟨*proof*⟩

lemma *remove-all-subs*:
set (remove-all x xs) ⊆ set xs

⟨*proof*⟩

lemma *remove-all-rev-distinct*: *distinct xs* ⟹ *distinct* (*x* # *remove-all-rev x xs*)
⟨*proof*⟩

lemma *remove-all-rev-removes*: *x* ∉ *set* (*remove-all-rev x xs*)
⟨*proof*⟩

lemma *remove-all-rev-subs*: *set* (*remove-all-rev x xs*) ⊆ *set xs*
⟨*proof*⟩

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

lemma *rem-cycles-distinct'*: *i* ≠ *j* ⟹ *distinct* (*i* # *j* # *rem-cycles i j xs*)
⟨*proof*⟩

lemma *rem-cycles-removes-last*: *j* ∉ *set* (*rem-cycles i j xs*)
⟨*proof*⟩

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

lemma *rem-cycles-subs*: *set* (*rem-cycles i j xs*) ⊆ *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 $'c\ mat = nat \Rightarrow nat \Rightarrow 'c$

definition $upd :: 'c\ mat \Rightarrow nat \Rightarrow nat \Rightarrow 'c \Rightarrow 'c\ mat$
where
$upd\ m\ x\ y\ v = m\ (x := (m\ x)\ (y := v))$

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

Recursive version of the two inner loops.

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

Recursive version of the full algorithm.

fun $fw :: 'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ mat$ **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*:
$fw\text{-}upd\ m\ k\ i\ j\ i'\ j' \leq m\ i'\ j'$
⟨*proof*⟩

lemma *fw-upd-out-of-bounds1*:
assumes $i' > i$
shows $(fw\text{-}upd\ M\ k\ i\ j)\ i'\ j' = M\ i'\ j'$
⟨*proof*⟩

lemma *fw-upd-out-of-bounds2*:
assumes $j' > j$
shows $(fw\text{-}upd\ M\ k\ i\ j)\ i'\ j' = M\ i'\ j'$
⟨*proof*⟩

lemma *fwi-out-of-bounds1*:
assumes $i' > n\ i \leq n$
shows $(fwi\ M\ n\ k\ i\ j)\ i'\ j' = M\ i'\ j'$
⟨*proof*⟩

lemma *fw-out-of-bounds1*:

assumes $i' > n$
shows $(fw\ M\ n\ k)\ i'\ j' = M\ i'\ j'$
⟨*proof*⟩

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

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

lemma *fwi-invariant-aux-1*:
$j'' \leq j \Longrightarrow fwi\ m\ n\ k\ i\ j\ i'\ j' \leq fwi\ m\ n\ k\ i\ j''\ i'\ j'$
⟨*proof*⟩

lemma *fwi-invariant*:
$j \leq n \Longrightarrow i'' \leq i \Longrightarrow j'' \leq j$
$\Longrightarrow fwi\ m\ n\ k\ i\ j\ i'\ j' \leq fwi\ m\ n\ k\ i''\ j''\ i'\ j'$
⟨*proof*⟩

lemma *single-row-inv*:
$j' < j \Longrightarrow fwi\ m\ n\ k\ i'\ j\ i'\ j' = fwi\ m\ n\ k\ i'\ j'\ i'\ j'$
⟨*proof*⟩

lemma *single-iteration-inv'*:
$i' < i \Longrightarrow j' \leq n \Longrightarrow fwi\ m\ n\ k\ i\ j\ i'\ j' = fwi\ m\ n\ k\ i'\ j'\ i'\ j'$
⟨*proof*⟩

lemma *single-iteration-inv*:
$i' \leq i \Longrightarrow j' \leq j \Longrightarrow j \leq n \Longrightarrow fwi\ m\ n\ k\ i\ j\ i'\ j' = fwi\ m\ n\ k\ i'\ j'\ i'\ j'$
⟨*proof*⟩

lemma *fwi-innermost-id*:
$i' < i \Longrightarrow fwi\ m\ n\ k\ i'\ j'\ i\ j = m\ i\ j$
⟨*proof*⟩

lemma *fwi-middle-id*:
$j' < j \Longrightarrow i' \leq i \Longrightarrow fwi\ m\ n\ k\ i'\ j'\ i\ j = m\ i\ j$
⟨*proof*⟩

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*⟩

lemma *fwi-mono*:
$fwi\ m\ n\ k\ i'\ j'\ i\ j \leq m\ i\ j$ **if** $i \leq n\ j \leq n$
⟨*proof*⟩

lemma *Suc-innermost-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ (Suc\ k)\ i\ j \leq fw\ m\ n\ k\ i\ j$
⟨*proof*⟩

lemma *fw-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ k\ i\ j \leq m\ i\ j$
⟨*proof*⟩

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

lemma *fwi-step*:
$m\ k\ k \geq 0 \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow fwi\ m\ n\ k\ i\ j\ i\ j = min\ (m\ i\ j)\ (m\ i\ k + m\ k\ j)$
⟨*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 $len :: 'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow 'a$ **where**
$len\ m\ u\ v\ [] = m\ u\ v\ |$
$len\ m\ u\ v\ (w\#ws) = m\ u\ w + len\ m\ w\ v\ ws$

lemma *len-decomp*: $xs = ys\ @\ y\ \#\ zs \Longrightarrow len\ m\ x\ z\ xs = len\ m\ x\ y\ ys + len\ m\ y\ z\ zs$
⟨*proof*⟩

lemma *len-comp*: $len\ m\ a\ c\ (xs\ @\ b\ \#\ ys) = len\ m\ a\ b\ xs + len\ m\ b\ c\ ys$
⟨*proof*⟩

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* :: *nat* ⇒ *nat set* ⇒ *'a mat* ⇒ *bool* **where**
canonical-subs n I m = (∀ *i j k*. *i* ≤ *n* ∧ *k* ≤ *n* ∧ *j* ∈ *I* ⟶ *m i k* ≤ *m i j* + *m j k*)

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

abbreviation *cyc-free-subs* :: *nat* ⇒ *nat set* ⇒ *'a mat* ⇒ *bool* **where**
cyc-free-subs n I m ≡ ∀ *i xs*. *i* ≤ *n* ∧ *set xs* ⊆ *I* ⟶ *len m i i xs* ≥ *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-subs-len*:
M i j ≤ *len M i j xs* **if** *canonical-subs n I M i* ≤ *n j* ≤ *n set xs* ⊆ *I I* ⊆ {*0*..*n*}
⟨*proof*⟩

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

lemma *fwi-step'*:
fwi m n k i' j' i j = *min* (*m i j*) (*m i k* + *m k j*) **if**
m k k ≥ *0 i'* ≤ *n j'* ≤ *n k* ≤ *n i* ≤ *i' j* ≤ *j'*
⟨*proof*⟩

An invocation of *fwi* extends canonical forms.

lemma *fwi-canonical-extend*:
canonical-subs n (*I* ∪ {*k*}) (*fwi m n k n n*) **if**
canonical-subs n I m I ⊆ {*0*..*n*} *0* ≤ *m k k k* ≤ *n*
⟨*proof*⟩

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 $\geq$ *0* **if**
cyc-free-subs n I m 0 $\leq$ *m k k k* $\leq$ *n k* $\in$ *I i* $\leq$ *n*
⟨*proof*⟩

lemma *cyc-free-subs-diag*:
m i i $\geq$ *0* **if** *cyc-free-subs n I m i* $\leq$ *n*
⟨*proof*⟩

lemma *fwi-cyc-free-subs′*:
cyc-free-subs n (*I* $\cup$ {*k*}) (*fwi m n k n n*) **if**
cyc-free-subs n I m canonical-subs n I m I $\subseteq$ {*0..n*} *k* $\leq$ *n*
$\forall$ *i* $\leq$ *n*. *fwi m n k n n i i* $\geq$ *0*
⟨*proof*⟩

lemma *fwi-cyc-free-subs*:
cyc-free-subs n (*I* $\cup$ {*k*}) (*fwi m n k n n*) **if**
cyc-free-subs n (*I* $\cup$ {*k*}) *m canonical-subs n I m I* $\subseteq$ {*0..n*} *k* $\leq$ *n*
⟨*proof*⟩

lemma *canonical-subs-empty* [*simp*]:
canonical-subs n {} *m*
⟨*proof*⟩

lemma *fwi-neg-diag-neg-cycle*:
$\exists$ *i* $\leq$ *n*. $\exists$ *xs*. *set xs* $\subseteq$ {*0..k*} $\wedge$ *len m i i xs* $<$ *0* **if** *fwi m n k n n i i* $<$ *0 i* $\leq$ *n k* $\leq$ *n*
⟨*proof*⟩

fwi preserves the length of paths.

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

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

Main theorem under the absence of negative cycles.

theorem *fw-correct*:
canonical-subs $n\ \{0..k\}\ (fw\ m\ n\ k) \wedge$ *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 $\{0 \ldots k\}$.

definition $D :: 'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a$ **where**
$D\ m\ i\ j\ k \equiv Min\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$

lemma *distinct-length-le*:*finite* $s \Longrightarrow set\ xs \subseteq s \Longrightarrow distinct\ xs \Longrightarrow length\ xs \leq card\ s$
⟨*proof*⟩

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

lemma *D-base-finite*:
finite $\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ xs\}$
⟨*proof*⟩

lemma *D-base-finite′*:
finite $\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ (i\ \#\ j\ \#\ xs)\}$
⟨*proof*⟩

lemma *D-base-finite″*:
finite $\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
⟨*proof*⟩

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

cycle-free m n ≡ ∀ *i xs*. *i* ≤ *n* ∧ *set xs* ⊆ {*0..n*} ⟶
(∀ *j*. *j* ≤ *n* ⟶ *len m i j* (*rem-cycles i j xs*) ≤ *len m i j xs*) ∧ *len m i i xs* ≥ *0*

lemma *D-eqI*:
fixes *m n i j k*
defines *A* ≡ {*len m i j xs* | *xs*. *set xs* ⊆ {*0..k*}}
defines *A-distinct* ≡ {*len m i j xs* |*xs*. *set xs* ⊆ {*0..k*} ∧ *i* ∉ *set xs* ∧ *j* ∉ *set xs* ∧ *distinct xs*}
assumes *cycle-free m n i* ≤ *n j* ≤ *n k* ≤ *n* (⋀*y*. *y* ∈ *A-distinct* ⟹ *x* ≤ *y*) *x* ∈ *A*
shows *D m i j k* = *x* ⟨*proof*⟩

lemma *D-base-not-empty*:
{*len m i j xs* |*xs*. *set xs* ⊆ {*0..k*} ∧ *i* ∉ *set xs* ∧ *j* ∉ *set xs* ∧ *distinct xs*} ≠ {}
⟨*proof*⟩

lemma *Min-elem-dest*: *finite A* ⟹ *A* ≠ {} ⟹ *x* = *Min A* ⟹ *x* ∈ *A*
⟨*proof*⟩

lemma *D-dest*: *x* = *D m i j k* ⟹
x ∈ {*len m i j xs* |*xs*. *set xs* ⊆ {*0..Suc k*} ∧ *i* ∉ *set xs* ∧ *j* ∉ *set xs* ∧ *distinct xs*}
⟨*proof*⟩

lemma *D-dest′*: *x* = *D m i j k* ⟹ *x* ∈ {*len m i j xs* |*xs*. *set xs* ⊆ {*0..Suc k*}}
⟨*proof*⟩

lemma *D-dest″*: *x* = *D m i j k* ⟹ *x* ∈ {*len m i j xs* |*xs*. *set xs* ⊆ {*0..k*}}
⟨*proof*⟩

lemma *cycle-free-loop-dest*: *i* ≤ *n* ⟹ *set xs* ⊆ {*0..n*} ⟹ *cycle-free m n* ⟹ *len m i i xs* ≥ *0*
⟨*proof*⟩

lemma *cycle-free-dest*:
cycle-free m n ⟹ *i* ≤ *n* ⟹ *j* ≤ *n* ⟹ *set xs* ⊆ {*0..n*}
⟹ *len m i j* (*rem-cycles i j xs*) ≤ *len m i j xs*
⟨*proof*⟩

definition *cycle-free-up-to* :: *′a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
cycle-free-up-to m k n ≡ ∀ *i xs*. *i* ≤ *n* ∧ *set xs* ⊆ {*0..k*} ⟶

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

lemma *cycle-free-up-to-loop-dest*:
$i \leq n \Longrightarrow set\ xs \subseteq \{0..k\} \Longrightarrow cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n \Longrightarrow len\ m\ i\ i\ xs \geq 0$
⟨*proof*⟩

lemma *cycle-free-up-to-diag*:
assumes *cycle-free-up-to m k n i* ≤ *n*
shows *m i i* ≥ *0*
⟨*proof*⟩

lemma *D-eqI2*:
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-free-up-to m k n i* ≤ *n j* ≤ *n k* ≤ *n*
$(\bigwedge y.\ y \in A\text{-}distinct \Longrightarrow x \leq y)\ x \in A$
shows *D m i j k* = *x* ⟨*proof*⟩

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*
⟨*proof*⟩

theorem *fw-subs-len*:
(*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*⟩

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′*:
∃ *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′*
⟨*proof*⟩

The same result for *fw*.

lemma *fw-len*:
$\exists\ xs.\ set\ xs \subseteq \{0..k\} \wedge fw\ m\ n\ k\ i\ j = len\ m\ i\ j\ xs$ **if**
cyc-free-subs $n\ \{0..k\}\ m\ i \leq n\ j \leq n\ k \leq n$
⟨*proof*⟩

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 $i \notin set\ ys$
assumes $i \in set\ xs$
assumes $xs = as\ @\ concat\ (map\ ((\#)\ i)\ xss)\ @\ xs'$
assumes $len\ m\ i\ j\ ys > len\ m\ i\ j\ xs$
shows $\exists\ ys.\ set\ ys \subseteq set\ xs \wedge len\ m\ i\ i\ ys < 0$ ⟨*proof*⟩

lemma *add-lt-neutral*: $a + b < b \Longrightarrow a < 0$
⟨*proof*⟩

lemma *remove-cycles-neg-cycles-aux'*:
fixes *j xs ys*
assumes $j \notin set\ ys$
assumes $j \in set\ xs$
assumes $xs = ys\ @\ j\ \#\ concat\ (map\ (\lambda\ xs.\ xs\ @\ [j])\ xss)\ @\ as$
assumes $len\ m\ i\ j\ ys > len\ m\ i\ j\ xs$
shows $\exists\ ys.\ set\ ys \subseteq set\ xs \wedge len\ m\ j\ j\ ys < 0$ ⟨*proof*⟩

lemma *add-le-impl*: $a + b < a + c \Longrightarrow b < c$
⟨*proof*⟩

lemma *start-remove-neg-cycles*:
$len\ m\ i\ j\ (start\text{-}remove\ xs\ k\ []) > len\ m\ i\ j\ xs \Longrightarrow \exists\ ys.\ set\ ys \subseteq set\ xs \wedge$
$len\ m\ k\ k\ ys < 0$
⟨*proof*⟩

lemma *remove-all-cycles-neg-cycles*:
$len\ m\ i\ j\ (remove\text{-}all\text{-}cycles\ ys\ xs) > len\ m\ i\ j\ xs$
$\Longrightarrow \exists\ ys\ k.\ set\ ys \subseteq set\ xs \wedge k \in set\ xs \wedge len\ m\ k\ k\ ys < 0$
⟨*proof*⟩

lemma *concat-map-cons-rev*:

rev (*concat* (*map* ((#) *j*) *xss*)) = *concat* (*map* (λ *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*
⟹ ∃ *i' ys*. *len m i' i' ys* < *0* ∧ *set ys* ⊆ *set xs* ∧ *i'* ∈ *set* (*i* # *j* # *xs*)
⟨*proof*⟩

1.6.2 Cycle-Freeness

lemma *cycle-free-alt-def*:
cycle-free M n ⟷ *cycle-free-up-to M n n*
⟨*proof*⟩

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*⟩

lemma *negative-cycle-dest-diag'*:
¬ *cycle-free m n* ⟹ ∃ *i xs*. *i* ≤ *n* ∧ *set xs* ⊆ {*0*..*n*} ∧ *len m i i xs* < *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cycle-free-diag-equiv*:
cyc-free m n ⟷ *cycle-free m n* ⟨*proof*⟩

lemma *cycle-free-diag-dest*:
cycle-free m n ⟹ *cyc-free m n*
⟨*proof*⟩

lemma *cycle-free-upto-diag-equiv*:
cycle-free-up-to m k n ⟷ *cyc-free-subs n* {*0*..*k*} *m* **if** *k* ≤ *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* ≤ *n j* ≤ *n k* ≤ *n*

⟨*proof*⟩

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

lemma
cyc-free m n ⟷ *cyc-free-subs n* {*0..n*} *m* ⟨*proof*⟩

lemma *cycle-free-cycle-free-up-to*:
cycle-free m n ⟹ *k* ≤ *n* ⟹ *cycle-free-up-to m k n*
⟨*proof*⟩

lemma *cycle-free-diag*:
cycle-free m n ⟹ *i* ≤ *n* ⟹ *0* ≤ *m i i*
⟨*proof*⟩

corollary *fw-shortest-path*:
cyc-free m n ⟹ *i* ≤ *n* ⟹ *j* ≤ *n* ⟹ *k* ≤ *n* ⟹ *D m i j k* = *fw m n k i j*
⟨*proof*⟩

corollary *fw-shortest*:
assumes *cyc-free m n i* ≤ *n j* ≤ *n k* ≤ *n*
shows *fw m n n i j* ≤ *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*: ¬ *cycle-free m n* ⟹ ∃ *k* ≤ *n*. ¬ *cycle-free-up-to m k n*
⟨*proof*⟩

lemma *D-not-diag-le*:
(*x* :: *'a*) ∈ {*len m i j xs* |*xs*. *set xs* ⊆ {*0..k*} ∧ *i* ∉ *set xs* ∧ *j* ∉ *set xs* ∧ *distinct xs*}
⟹ *D m i j k* ≤ *x* ⟨*proof*⟩

lemma *D-not-diag-le'*: *set xs* ⊆ {*0..k*} ⟹ *i* ∉ *set xs* ⟹ *j* ∉ *set xs* ⟹ *distinct xs*
⟹ *D m i j k* ≤ *len m i j xs* ⟨*proof*⟩

lemma *nat-upto-subs-top-removal'*:
S ⊆ {*0..Suc n*} ⟹ *Suc n* ∉ *S* ⟹ *S* ⊆ {*0..n*}
⟨*proof*⟩

lemma *nat-upto-subs-top-removal*:
$S \subseteq \{0..n::nat\} \Longrightarrow n \notin S \Longrightarrow S \subseteq \{0..n - 1\}$
⟨*proof*⟩

Monotonicity with respect to k.

lemma *fw-invariant*:
$k' \leq k \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow fw\ m\ n\ k\ i\ j \leq fw\ m\ n\ k'\ i\ j$
⟨*proof*⟩

lemma *negative-len-shortest*:
$length\ xs = n \Longrightarrow len\ m\ i\ i\ xs < 0$
$\Longrightarrow \exists\ j\ ys.\ distinct\ (j\ \#\ ys) \wedge len\ m\ j\ j\ ys < 0 \wedge j \in set\ (i\ \#\ xs) \wedge set\ ys \subseteq set\ xs$
⟨*proof*⟩

lemma *fw-upd-leI*:
$fw\text{-}upd\ m'\ k\ i\ j\ i\ j \leq fw\text{-}upd\ m\ k\ i\ j\ i\ j$ **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*:
$fwi\ m\ n\ k\ i\ j\ i\ j \leq fw\text{-}upd\ m\ k\ i\ j\ i\ j$ **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 $\{0 \ldots k\}$. 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*:
$\neg\ cyc\text{-}free\ m\ n \Longrightarrow \exists\ i \leq n.\ fw\ m\ n\ n\ i\ i < 0$
⟨*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 $\longleftrightarrow$ *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 $\Longrightarrow$ $i \leq n \Longrightarrow j \leq n \Longrightarrow$ *set xs* $\subseteq \{0..n\} \Longrightarrow$ *M i j* $\leq$ *len M i j xs*
⟨*proof*⟩

1.9 Additional Theorems

lemma *D-cycle-free-len-dest*:
cycle-free m n
$\Longrightarrow \forall\ i \leq n.\ \forall\ j \leq n.$ *D m i j n* = *m′ i j* $\Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow$ *set xs* $\subseteq \{0..n\}$
$\Longrightarrow \exists$ *ys. set ys* $\subseteq \{0..n\} \wedge$ *len m′ i j xs* = *len m i j ys*
⟨*proof*⟩

lemma *D-cyc-free-preservation*:
cyc-free m n $\Longrightarrow \forall\ i \leq n.\ \forall\ j \leq n.$ *D m i j n* = *m′ i j* $\Longrightarrow$ *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 $\Longrightarrow$ *cyc-free* (*FW m n*) *n*
⟨*proof*⟩

lemma *cyc-free-diag-dest′*:
cyc-free m n $\Longrightarrow i \leq n \Longrightarrow$ *m i i* ≥ 0
⟨*proof*⟩

lemma *FW-diag-neutral-preservation*:
$\forall\ i \leq n.\ M\ i\ i = 0 \Longrightarrow$ *cyc-free* $M\ n \Longrightarrow \forall\ i{\leq}n.\ (FW\ M\ n)\ i\ i = 0$
⟨*proof*⟩

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 $FW\ M\ n\ 0\ i + FW\ M\ n\ i\ 0 = 0$ ⟨*proof*⟩

lemma *diag-cyc-free-neutral*:
cyc-free $M\ n \Longrightarrow \forall\, k{\leq}n.\ M\ k\ k \leq 0 \Longrightarrow \forall\, i{\leq}n.\ M\ i\ i = 0$
⟨*proof*⟩

lemma *fw-upd-canonical-subs-id*:
canonical-subs $n\ \{k\}\ M \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow$ *fw-upd* $M\ k\ i\ j = M$
⟨*proof*⟩

lemma *fw-upd-canonical-id*:
canonical $M\ n \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow$ *fw-upd* $M\ k\ i\ j = M$
⟨*proof*⟩

lemma *fwi-canonical-id*:
fwi $M\ n\ k\ i\ j = M$ **if** *canonical-subs* $n\ \{k\}\ M\ i \leq n\ j \leq n\ k \leq n$
⟨*proof*⟩

lemma *fw-canonical-id*:
fw $M\ n\ k = M$ **if** *canonical-subs* $n\ \{0..k\}\ M\ k \leq n$
⟨*proof*⟩

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

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

The characteristic property of *fwi*.

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

end

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

context
begin

private definition *for-comb* **where**
for-comb f a0 n = *nfoldli* [*0*..<*n* + *1*] (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*
⟨*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 (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
⟨*proof*⟩ **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
⟨*proof*⟩ **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
⟨*proof*⟩

theorem *for-rec-eq*:
for-rec f a n = nfoldli [0..<n + 1] (λx. True) (λk a. f a k) a
⟨*proof*⟩

theorem *for-rec2-eq*:
for-rec2 f a n n n =
nfoldli [0..<n + 1] (λx. True)
(λi. nfoldli [0..<n + 1] (λx. True) (λj a. f a i j)) a
⟨*proof*⟩

theorem *for-rec3-eq*:
for-rec3 f a n n n n =
nfoldli [0..<n + 1] (λx. True)
(λk. nfoldli [0..<n + 1] (λx. True)
(λi. nfoldli [0..<n + 1] (λx. True) (λj 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 Imperative Refinement 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*)))
)

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
fwi' m n k i j ≤ *SPEC* (λ *r*. *r* = *uncurry* (*fwi* (*curry m*) *n k i j*))
⟨*proof*⟩

lemma *fw-upd'-spec*:
fw-upd' M k i j ≤ *SPEC* (λ *M'*. *M'* = *uncurry* (*fw-upd* (*curry M*) *k i j*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*] (λ -. *True*) (λ *k M*. *for-rec2* (λ *M*. *fw-upd'* *M k*) *M n n n*) *m*

lemma *fw'-spec*:
fw' m n k $\leq$ *SPEC* (λ *M'*. *M'* = *uncurry* (*fw* (*curry m*) *n k*))
⟨*proof*⟩

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

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

sepref-definition *fw-upd-impl* **is**
uncurry2 (*uncurry fw-upd'*) ::
[λ (((-,*k*),*i*),*j*). *k* $\leq$ *n* $\wedge$ *i* $\leq$ *n* $\wedge$ *j* $\leq$ *n*]$_a$ *mtx-assn*^{d} $*_a$ *node-assn*^{k} $*_a$ *node-assn*^{k} $*_a$ *node-assn*^{k}
$\rightarrow$ *mtx-assn*
⟨*proof*⟩

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

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

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

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

context
  notes [*id-rules*] = *itypeI*[*of n TYPE* (*nat*)]
    and [*sepref-import-param*] = *IdI*[*of n*]
begin

sepref-definition *fw-impl* **is**
  fw-impl′ :: *mtx-assn*^{d} →$_a$ *mtx-assn*
⟨*proof*⟩

sepref-definition *fwi-impl* **is**
  uncurry fwi-impl′ :: [λ (-,*k*). *k* ≤ *n*]$_a$ *mtx-assn*^{d} ∗$_a$ *node-assn*^{k} → *mtx-assn*
⟨*proof*⟩

end

end

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

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

definition *fw-spec* **where**
  fw-spec n M ≡ *SPEC* (λ *M′*.
    if (∃ *i* ≤ *n*. *M′ i i* < *0*)
    then ¬ *cyc-free M n*
    else ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M′ i j* = *D M i j n* ∧ *cyc-free M n*)

lemma *D-diag-nonnegI*:
  assumes *cycle-free M n i* ≤ *n*
  shows *D M i i n* ≥ *0*
⟨*proof*⟩

lemma *fw-fw-spec*:
  RETURN (*FW M n*) ≤ *fw-spec n M*
⟨*proof*⟩

definition
  mat-curry-rel = {(*Mu*, *Mc*). *curry Mu* = *Mc*}

definition
  mtx-curry-assn n = *hr-comp* (*mtx-assn n*) (*br curry* (λ-. *True*))

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

lemma *fw-impl′-correct*:
(*fw-impl′*, *fw-spec*) ∈ *Id* → *br curry* (λ -. *True*) → ⟨*br curry* (λ -. *True*)⟩ *nres-rel*
⟨*proof*⟩

1.10.1 Main Result

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

theorem *fw-impl-correct*:
(*fw-impl n*, *fw-spec n*) ∈ (*mtx-curry-assn n*)^d →$_a$ *mtx-curry-assn n*
⟨*proof*⟩

An alternative version: a Hoare triple for total correctness.

corollary
<*mtx-curry-assn n M Mi*> *fw-impl n Mi* <λ *Mi′*. ∃$_A$ *M′*. *mtx-curry-assn n M′ Mi′* ∗ ↑
(*if* (∃ *i* ≤ *n*. *M′ i i* < *0*)
then ¬ *cyc-free M n*
else ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M′ i j* = *D M i j n* ∧ *cyc-free M n*)>$_t$
⟨*proof*⟩

1.10.2 Alternative Versions for Uncurried Matrices.

definition *FWI′* = *uncurry ooo FWI o curry*

lemma *fwi-impl′-refine-FWI′*:
(*fwi-impl′ n*, *RETURN oo PR-CONST* (λ *M*. *FWI′ M n*)) ∈ *Id* → *Id* → ⟨*Id*⟩ *nres-rel*
⟨*proof*⟩

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

definition *FW′* = *uncurry oo FW o curry*

definition *FW″ n M* = *FW′ M n*

lemma *fw-impl′-refine-FW″*:
(*fw-impl′ n*, *RETURN o PR-CONST* (*FW″ n*)) ∈ *Id* → ⟨*Id*⟩ *nres-rel*
⟨*proof*⟩

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

end

References

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[Roy59] Bernard Roy. Transitivité et connexité. In *Extrait des comptes rendus des séances de lAcadémie 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.