

Travelling Salesman Problem

Maksym Bortin

October 28, 2026

Abstract

Travelling Salesman Problem (TSP) is well-known to be NP-hard. The computational solution, derived in this theory, follows the Bellman-Held-Karp [1, 2] approach utilising the power set of locations to be visited by the salesman.

Contents

1	The Bellman-Held-Karp approach to TSP	1
1.1	Hamiltonian paths	1
1.2	Optimal Hamiltonian paths	2
1.3	Minimal distance extensions	4
1.4	<i>opt-Hamiltonian-paths</i> as a fixed point	4
2	An implementation using dynamic programming	4
2.1	A few preparations	4
2.2	Indexing subsets of $\{0, \dots, n\}$ represented by sorted lists . . .	5
2.3	Computing minimal distance extensions	9
2.4	Tables	11
2.5	Summing up	13
2.6	An example	13

```
theory TSP-Dyn-Prog
imports Main
begin
```

1 The Bellman-Held-Karp approach to TSP

1.1 Hamiltonian paths

A Hamiltonian path from a vertex s to a vertex t via a (finite) set V has to pass through each vertex in V exactly once. This can be captured by a list

that starts with s , ends with t and has a distinct list containing all vertices from V (i.e. essentially the same as a permutation on V) in-between.

In particular, with $s = t$ one gets a representation of a Hamiltonian cycle from s via V .

Also note that s or t might in principle occur in V as well.

definition

Hamiltonian-paths $s \ V \ t = \{s \# vs @ [t] \mid vs. \text{ distinct } vs \wedge \text{ set } vs = V\}$

By the above definition, a Hamiltonian path exists only for a finite V :

lemma *Hamiltonian-paths-not-empty* :

finite $V = (\exists p. p \in \text{Hamiltonian-paths } s \ V \ t)$

<proof>

lemma *Hamiltonian-paths-finite* :

finite(*Hamiltonian-paths* $s \ V \ t$)

<proof>

lemma *Hamiltonian-paths0* :

Hamiltonian-paths $s \ \{\} \ t = \{[s, t]\}$

<proof>

If V is not empty then the tail of any Hamiltonian path from s to t via V is in turn a Hamiltonian path starting with a vertex in V :

lemma *Hamiltonian-pathsD* :

assumes $p \in \text{Hamiltonian-paths } s \ V \ t$

and $V \neq \{\}$

shows $\exists v \in V. \exists p' \in \text{Hamiltonian-paths } v \ (V - \{v\}) \ t. p = s \# p'$

<proof>

lemma *Hamiltonian-pathsI* :

assumes $p \in \text{Hamiltonian-paths } v \ (V - \{v\}) \ t$

and $v \in V$

shows $s \# p \in \text{Hamiltonian-paths } s \ V \ t$

<proof>

1.2 Optimal Hamiltonian paths

Extending an edge distance function δ to paths:

fun *path-distance* **where**

path-distance $\delta \ [] = (0 :: 'a :: \text{linordered-ab-group-add}) \mid$

path-distance $\delta \ [v] = 0 \mid$

path-distance $\delta \ (v1 \# v2 \# vs) = \delta \ v1 \ v2 + \text{path-distance } \delta \ (v2 \# vs)$

lemma *Hamiltonian-path-distance-Cons-le* :
assumes $p \in \text{Hamiltonian-paths } s \ V \ t$
and $p' \in \text{Hamiltonian-paths } s \ V \ t$
shows $(\text{path-distance } \delta \ p \leq \text{path-distance } \delta \ p') =$
 $(\text{path-distance } \delta \ (v\#p) \leq \text{path-distance } \delta \ (v\#p'))$
 $\langle \text{proof} \rangle$

A Hamiltonian path is optimal with respect to the given edge distance function δ if its *path-distance* is minimal:

definition
 $\text{opt-Hamiltonian-paths } \delta \ s \ V \ t = \{p. p \in \text{Hamiltonian-paths } s \ V \ t \wedge$
 $(\forall p' \in \text{Hamiltonian-paths } s \ V \ t. \text{path-distance } \delta \ p \leq \text{path-distance } \delta \ p')\}$

lemma *opt-Hamiltonian-paths0* :
 $\text{opt-Hamiltonian-paths } \delta \ s \ \{\} \ t = \{[s, t]\}$
 $\langle \text{proof} \rangle$

lemma *opt-Hamiltonian-pathsD* :
 $\text{opt-Hamiltonian-paths } \delta \ s \ V \ t \subseteq \text{Hamiltonian-paths } s \ V \ t$
 $\langle \text{proof} \rangle$

lemma *opt-Hamiltonian-paths-same-distance* :
assumes $p \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t$
and $p' \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t$
shows $\text{path-distance } \delta \ p = \text{path-distance } \delta \ p'$
 $\langle \text{proof} \rangle$

lemma *opt-Hamiltonian-paths-not-empty* :
assumes *finite* V
shows $\exists p. p \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t$
 $\langle \text{proof} \rangle$

lemma *opt-Hamiltonian-pathsI* :
assumes $p \in \text{Hamiltonian-paths } s \ V \ t$
and $\forall p' \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t. \text{path-distance } \delta \ p \leq \text{path-distance } \delta \ p'$
shows $p \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t$
 $\langle \text{proof} \rangle$

1.3 Minimal distance extensions

definition *Min-distance-ext* **where**

Min-distance-ext $\delta \ v \ S = \{v\#p \mid p. p \in S \wedge$
 $(\forall p' \in S. \text{path-distance } \delta \ (v\#p) \leq \text{path-distance } \delta \ (v\#p'))\}$

lemma *Min-distance-ext* :

$(p \in \text{Min-distance-ext } \delta \ v \ S) =$
 $(\exists p' \in S. p = v\#p' \wedge (\forall p'' \in S. \text{path-distance } \delta \ (v\#p') \leq \text{path-distance } \delta \ (v\#p'')))$
 $\langle \text{proof} \rangle$

lemma *Min-distance-ext-eq'* :

assumes $S \subseteq T$
and $\forall p' \in T - S. \exists p \in S. \text{path-distance } \delta \ (v\#p) < \text{path-distance } \delta \ (v\#p')$
shows $(p \in \text{Min-distance-ext } \delta \ v \ S) = (p \in \text{Min-distance-ext } \delta \ v \ T)$
(is ?L = ?R)
 $\langle \text{proof} \rangle$

corollary *Min-distance-ext-eq* :

assumes $S \subseteq T$
and $\forall p' \in T - S. \exists p \in S. \text{path-distance } \delta \ (v\#p) < \text{path-distance } \delta \ (v\#p')$
shows $\text{Min-distance-ext } \delta \ v \ S = \text{Min-distance-ext } \delta \ v \ T$
 $\langle \text{proof} \rangle$

1.4 opt-Hamiltonian-paths as a fixed point

theorem *opt-Hamiltonian-paths-fix* :

assumes $V \neq \{\}$
shows $\text{opt-Hamiltonian-paths } \delta \ s \ V \ t =$
 $\text{Min-distance-ext } \delta \ s \ (\bigcup v \in V. \text{opt-Hamiltonian-paths } \delta \ v \ (V - \{v\}) \ t)$
(is ?L = ?R)
 $\langle \text{proof} \rangle$

2 An implementation using dynamic programming

2.1 A few preparations

Membership test for lists with a (strictly) descending order of elements:

fun *memS* **where**

memS $a \ [] = \text{False}$ |
memS $a \ (x\#xs) = (a \leq x \wedge (a = x \vee \text{memS } a \ xs))$

lemma *memS* :

assumes *sorted-wrt* $(>) \ xs$
shows $\text{memS } (a :: 'a :: \text{linorder}) \ xs = (a \in \text{set } xs)$

<proof>

Removal from lists with a strictly descending order of elements:

fun *rem1S* **where**

rem1S (*a* :: 'a :: linorder) [] = [] |

rem1S *a* (*x*#*xs*) = (if *a* = *x* then *xs* else (if *a* < *x* then *x*#*rem1S* *a* *xs* else *x*#*xs*))

lemma *rem1S* :

assumes *sorted-wrt* (>) *xs*

shows *set*(*rem1S* *a* *xs*) = *set* *xs* - {*a*}

<proof>

lemma *rem1S-length* :

assumes *sorted-wrt* (>) *xs*

and *a* ∈ *set* *xs*

shows *length*(*rem1S* *a* *xs*) + 1 = *length* *xs*

<proof>

corollary *rem1S-length'* :

assumes *sorted-wrt* (>) *xs*

and *a* ∈ *set* *xs*

shows *length*(*rem1S* *a* *xs*) < *length* *xs*

<proof>

lemma *sorted-wrt-rem1S* :

assumes *sorted-wrt* (>) *xs*

shows *sorted-wrt* (>) (*rem1S* *a* *xs*)

<proof>

lemma *sorted-lists-eq* :

assumes *sorted-wrt* (>) (*as* :: 'a :: linorder list)

and *sorted-wrt* (>) *bs*

and *set* *as* ⊆ *set* *bs*

and *length* *as* = *length* *bs*

shows *as* = *bs*

<proof>

2.2 Indexing subsets of $\{0, \dots, n\}$ represented by sorted lists

fun *nat2list* :: *nat* ⇒ *nat* ⇒ *nat* list

where

nat2list *c* *ix* = (if *ix* = 0 then [] |

else (if even *ix* then *nat2list* (*c*+1) (*ix* div 2)

else *nat2list* (*c*+1) (*ix* div 2) @ [*c*]))

lemma *nat2list-Nil* :
 $(\text{nat2list } c \text{ } ix = []) = (ix = 0)$
 $\langle \text{proof} \rangle$

abbreviation *pow2* :: $\text{nat} \Rightarrow \text{nat}$ ($2^{\cdot}$ 80)
where $\text{pow2 } n \equiv 2^{\wedge n}$

lemma *nat2list-ub* :
assumes $ix < 2^b$
shows $\forall i \in \text{set}(\text{nat2list } c \text{ } ix). i < b + c$
 $\langle \text{proof} \rangle$

lemma *nat2list-lb* :
 $\forall i \in \text{set}(\text{nat2list } c \text{ } ix). c \leq i$
 $\langle \text{proof} \rangle$

lemma *nat2list-sorted* :
 $\text{sorted-wrt } (>) (\text{nat2list } c \text{ } ix)$
 $\langle \text{proof} \rangle$

lemma *nat2list-pow2*[*OF refl*] :
assumes $b = 2^n$
shows $\text{nat2list } c \text{ } b = [c+n]$
 $\langle \text{proof} \rangle$

lemma *pow2-div2* :
assumes $n > 0$
shows $(2^n + x) \text{ div } 2 = 2^{n-1} + x \text{ div } 2$
 $\langle \text{proof} \rangle$

lemma *pow2pred-div2* :
 $(2^n - 1) \text{ div } 2 = 2^{n-1} - 1$
 $\langle \text{proof} \rangle$

lemma *nat2list-pred-pow2* :
assumes $b = 2^n - 1$

shows $\text{nat2list } c \ b = \text{rev } [c..<n+c]$
 $\langle \text{proof} \rangle$

corollary $\text{nat2list-pred-pow2}' :$
 $\text{nat2list } 0 \ (2^n - 1) = \text{rev } [0..<n]$
 $\langle \text{proof} \rangle$

lemma $\text{nat2list-shift} :$
assumes $ix < 2^b$
shows $\text{nat2list } c \ (2^b + ix) = (c + b) \# (\text{nat2list } c \ ix)$
 $\langle \text{proof} \rangle$

lemma $\text{nat2list-inj} :$
assumes $ix \neq ix'$
shows $\text{nat2list } c \ ix \neq \text{nat2list } c \ ix'$
 $\langle \text{proof} \rangle$

fun $\text{list2nat} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat}$
where
 $\text{list2nat } r \ b \ [] = 2^b * r \mid$
 $\text{list2nat } r \ b \ (e \# l) = (\text{if } b > 0 \text{ then}$
 $\quad (\text{if } b-1 = e \text{ then } \text{list2nat } (2*r + 1) \ (b-1) \ l$
 $\quad \text{else } \text{list2nat } (2*r) \ (b-1) \ (e \# l))$
 $\text{else } r)$

lemma $\text{list2nat-eg} :$
 $\text{list2nat } r \ b \ l = 2^b * r + \text{list2nat } 0 \ b \ l$
 $\langle \text{proof} \rangle$

lemma $\text{list2nat-gr0} :$
assumes $\text{sorted-wrt } (>) \ l$
and $\text{length } l > 0$
and $\forall x \in \text{set } l. \ x < b$
shows $\text{list2nat } r \ b \ l > 0$
 $\langle \text{proof} \rangle$

```

lemma list2nat-ub :
  assumes  $\forall x \in \text{set } l. x < b$ 
    and sorted-wrt ( $>$ ) l
  shows  $\text{list2nat } r \ b \ l < 2^b * r + 2^b$ 
   $\langle \text{proof} \rangle$ 

lemmas list2nat-ub0 = list2nat-ub [where  $r=0$ , simplified]

lemma list2nat-inj :
  assumes  $\forall x \in \text{set } l. x < b$ 
    and sorted-wrt ( $>$ ) l
  shows  $\text{nat2list } 0 \ (\text{list2nat } 0 \ b \ l) = l$ 
   $\langle \text{proof} \rangle$ 

corollary list2nat-surj :
  assumes  $ix < 2^b$ 
  shows  $\text{list2nat } 0 \ b \ (\text{nat2list } 0 \ ix) = ix$ 
   $\langle \text{proof} \rangle$ 

lemma list2nat-mono :
  assumes  $\forall x \in \text{set } l. x < b$ 
    and sorted-wrt ( $>$ ) l
    and  $\text{length } l' < \text{length } l$ 
    and  $\text{set } l' \subseteq \text{set } l$ 
    and sorted-wrt ( $>$ ) l'
  shows  $\text{list2nat } r \ b \ l' < \text{list2nat } r \ b \ l$ 
   $\langle \text{proof} \rangle$ 

declare list2nat.simps[simp del] nat2list.simps[simp del]

corollary list2nat-rem1S :
  assumes sorted-wrt ( $>$ ) l
    and  $\forall x \in \text{set } l. x < b$ 
    and  $u \in \text{set } l$ 
  shows  $\text{list2nat } r \ b \ (\text{rem1S } u \ l) < \text{list2nat } r \ b \ l$ 
   $\langle \text{proof} \rangle$ 

corollary list2nat-rem1S-not-in :

```

assumes $u \in \text{set } (\text{nat2list } 0 \text{ } ix)$
and $ix < 2^N$
shows $u \notin \text{set}(\text{nat2list } 0 \text{ } (\text{list2nat } 0 \text{ } N \text{ } (\text{rem1S } u \text{ } (\text{nat2list } 0 \text{ } ix))))$
 $\langle \text{proof} \rangle$

corollary *list2nat-rem1S-less* :
assumes $u \in \text{set } (\text{nat2list } 0 \text{ } ix)$
and $ix < 2^N$
shows $\text{list2nat } 0 \text{ } N \text{ } (\text{rem1S } u \text{ } (\text{nat2list } 0 \text{ } ix)) < ix$
 $\langle \text{proof} \rangle$

2.3 Computing minimal distance extensions

fun *min-distance-ext'*
where
 $\text{min-distance-ext}' \delta \text{ } v \text{ } (md :: 'a :: \text{linordered-ab-group-add}) \text{ } mps \text{ } [] = (md, mps) \mid$
 $\text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } mps \text{ } ((h, (d, ps)) \# xs) =$
 $(\text{let } dp = \delta \text{ } v \text{ } h + d \text{ in}$
 $\text{if } md < dp \text{ then } \text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } mps \text{ } xs$
 $\text{else } (\text{if } md = dp \text{ then } \text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } ((\text{map } ((\#) \text{ } h) \text{ } ps) @ mps) \text{ } xs$
 $\text{else } \text{min-distance-ext}' \delta \text{ } v \text{ } dp \text{ } (\text{map } ((\#) \text{ } h) \text{ } ps) \text{ } xs))$

definition *min-distance-ext* $\delta \text{ } v \text{ } xs =$
 $(\text{case } xs \text{ of}$
 $[] \Rightarrow (0, [])$
 $\mid (h, (d, ps)) \# xs' \Rightarrow \text{min-distance-ext}' \delta \text{ } v \text{ } (\delta \text{ } v \text{ } h + d) \text{ } (\text{map } ((\#) \text{ } h) \text{ } ps) \text{ } xs')$

lemma *min-distance-ext'1* :
assumes $\delta \text{ } v \text{ } h + d > md$
shows $\text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } mps \text{ } ((h, d, ps) \# xs) =$
 $\text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } mps \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *min-distance-ext'2* :
assumes $\delta \text{ } v \text{ } h + d = md$
shows $\text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } mps \text{ } ((h, d, ps) \# xs) =$
 $\text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } (\text{map } ((\#) \text{ } h) \text{ } ps @ mps) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *min-distance-ext'3* :
assumes $\delta \text{ } v \text{ } h + d < md$
shows $\text{min-distance-ext}' \delta \text{ } v \text{ } md \text{ } mps \text{ } ((h, d, ps) \# xs) =$
 $\text{min-distance-ext}' \delta \text{ } v \text{ } (\delta \text{ } v \text{ } h + d) \text{ } (\text{map } ((\#) \text{ } h) \text{ } ps) \text{ } xs$
 $\langle \text{proof} \rangle$

declare *min-distance-ext'.simps*(2)[*simp del*]

definition

min-distance-ext-inv δ $h = (\lambda(d, ps). \text{length } ps > 0 \wedge (\forall p \in \text{set } ps. \text{path-distance } \delta (h \# p) = d))$

lemma *min-distance-ext-invD* :

assumes *min-distance-ext-inv* δ h (d, ps)

shows $\exists p. p \in \text{set } ps$ **and** $\forall p \in \text{set } ps. \text{path-distance } \delta (h \# p) = d$
 $\langle \text{proof} \rangle$

lemma *min-distance-ext'-inv* :

assumes *min-distance-ext'* δ v md mps $xs = (d, ps)$

and *min-distance-ext-inv* δ v (md, mps)

and $\forall (h, d, ps) \in \text{set } xs. \text{min-distance-ext-inv } \delta$ h (d, ps)

shows *min-distance-ext-inv* δ v (d, ps)

$\langle \text{proof} \rangle$

lemma *min-distance-ext'-eq* :

assumes *min-distance-ext'* δ v md mps $xs = (d, ps)$

and *min-distance-ext-inv* δ v (md, mps)

and $\forall (h, d', l) \in \text{set } xs. \text{min-distance-ext-inv } \delta$ h (d', l)

shows $(\#) v \text{ ' set } ps = \text{Min-distance-ext } \delta v \{p. p \in \text{set } mps \vee (\exists (h, d', l) \in \text{set } xs. p \in (\#) h \text{ ' set } l)\}$

(**is** $- = \text{Min-distance-ext } \delta v$ ($?s$ mps xs))

$\langle \text{proof} \rangle$

lemma *min-distance-ext* :

assumes *min-distance-ext* δ v $xs = (d, rs)$

and $\forall (h, d', l) \in \text{set } xs. \text{min-distance-ext-inv } \delta$ h (d', l)

and $\text{length } xs > 0$

shows *min-distance-ext-inv* δ v (d, rs) $\wedge$

$(\#) v \text{ ' set } rs = \text{Min-distance-ext } \delta v \{p. \exists (h, d', l) \in \text{set } xs. p \in (\#) h \text{ ' set } l\}$

(**is** $?A \wedge ?B$)

$\langle \text{proof} \rangle$

2.4 Tables

type-synonym *vertex* = *nat*

For simplicity distances shall be represented by integers below:

type-synonym *distance* = *int*

type-synonym *table* = *vertex* $\times$ *nat* $\Rightarrow$ *distance* $\times$ (*vertex list*) *list*

The table invariant condition:

definition *table-inv* :: (*vertex* $\Rightarrow$ *vertex* $\Rightarrow$ *distance*) $\Rightarrow$ *nat* $\Rightarrow$ *vertex* $\Rightarrow$ *table* $\Rightarrow$ *nat* $\Rightarrow$ *bool*

where

table-inv δ *N* *t* *T* *ix* =
 $(\forall i \leq ix. \forall v \in \{0..<N\} - \text{set}(\text{nat2list } 0 \ i). \\ \text{min-distance-ext-inv } \delta \ v \ (T(v, \ i)) \wedge \\ (\#) \ v \ ' \ \text{set}(\text{snd}(T(v, \ i))) = \text{opt-Hamiltonian-paths } \delta \ v \ (\text{set}(\text{nat2list } 0 \ i)) \ t)$

Initialising a table with edge distances from each of $\{0, \dots, N-1\}$ to the destination *t*:

fun *init-tab* :: (*vertex* $\Rightarrow$ *vertex* $\Rightarrow$ *distance*) $\Rightarrow$ *nat* $\Rightarrow$ *vertex* $\Rightarrow$ *table*

where

init-tab δ *N* *t* = (if *N* = 0 then ($\lambda x. (0, [])$)
 else (*init-tab* δ (*N*-1) *t*)((*N*-1, 0) := (δ (*N*-1) *t*, [[*t*]]))

lemma *table-inv-init* :

table-inv δ *N* *t* (*init-tab* δ *N* *t*) 0

<proof>

declare *init-tab.simps*[*simp del*]

Update a table along the column *ix* using the values in the preceding columns:

definition *upd-tab* :: (*vertex* $\Rightarrow$ *vertex* $\Rightarrow$ *distance*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *table* $\Rightarrow$ *table*

where

upd-tab δ *N* *ix* =
 (let *vs* = *nat2list* 0 *ix* in
 let *V* = [0..*N*] in
 foldr ($\lambda v \ T. \ T((v, \ ix) := \text{min-distance-ext } \delta \ v \ (\text{map } (\lambda w. (w, \ T(w, \ \text{list2nat } 0 \ N \\ (\text{rem1S } w \ vs)))) \ vs)))$ *vs*))
 (*filter* ($\lambda v. \neg \text{memS } v \ vs$) *V*)

lemma *foldr-upd1* :

assumes $\forall v \in \text{set } xs. f \ v \neq x$

shows $\text{foldr } (\lambda v \ T. \ T(f \ v := g \ v \ T)) \ xs \ T \ x = T \ x$
 $\langle \text{proof} \rangle$

lemma *foldr-upd2* :
assumes $x \in \text{set } xs$
and $\forall xs. m \ (\text{foldr } (\lambda v \ T. \ T((v, \ vs) := g \ v \ (m \ T))) \ xs \ T) = m \ T$
shows $\text{foldr } (\lambda v \ T. \ T((v, \ vs) := g \ v \ (m \ T))) \ xs \ T \ (x, \ vs) = g \ x \ (m \ T)$
 $\langle \text{proof} \rangle$

lemma *table-inv-upd* :
assumes $\text{table-inv } \delta \ N \ t \ T \ ix$
and $ix < ix'$
shows $\text{table-inv } \delta \ N \ t \ (\text{upd-tab } \delta \ N \ ix' \ T) \ ix$
 $\langle \text{proof} \rangle$

lemma *upd-tab-app* :
assumes $vs = \text{nat2list } 0 \ ix$
and $v \in \{0..<N\} - \text{set } vs$
and $ix < 2^N$
shows $\text{upd-tab } \delta \ N \ ix \ T \ (v, \ ix) =$
 $\text{min-distance-ext } \delta \ v \ (\text{map } (\lambda w. \ (w, \ T(w, \ \text{list2nat } 0 \ N \ (\text{rem1S } w \ vs)))) \ vs)$
 $\langle \text{proof} \rangle$

lemma *table-inv-upd-ext* :
assumes $\text{table-inv } \delta \ N \ t \ T \ ix$
and $ix + 1 < 2^N$
shows $\text{table-inv } \delta \ N \ t \ (\text{upd-tab } \delta \ N \ (ix+1) \ T) \ (ix+1)$
 $\langle \text{proof} \rangle$

corollary *table-inv-upd-ext'* :
assumes $\text{table-inv } \delta \ N \ t \ T \ (ix-1)$
and $ix < 2^N$
and $ix > 0$
shows $\text{table-inv } \delta \ N \ t \ (\text{upd-tab } \delta \ N \ ix \ T) \ ix$
 $\langle \text{proof} \rangle$

Successively update the columns up to $2^n - 1$ using *upd-tab*:

function *fill-tab* :: $(\text{vertex} \Rightarrow \text{vertex} \Rightarrow \text{distance}) \Rightarrow \text{nat} \Rightarrow \text{table} \Rightarrow \text{nat} \Rightarrow \text{table}$
where
 $\text{fill-tab } \delta \ N \ T \ ix = (\text{if } ix < 2^N - 1 \text{ then } \text{fill-tab } \delta \ N \ (\text{upd-tab } \delta \ N \ ix \ T) \ (ix+1)$
 $\text{else } T)$
 $\langle \text{proof} \rangle$

termination

<proof>

declare *fill-tab.simps*[*simp del*]

lemma *fill-tab* :

assumes $N > 0$

and $ix > 0$

and $\forall ix' < ix. \text{table-inv } \delta \ N \ t \ T \ ix'$

shows $\text{table-inv } \delta \ N \ t \ (\text{fill-tab } \delta \ N \ T \ ix) \ (2^N - 2)$

<proof>

2.5 Summing up

definition

tsp $\delta \ N =$

$(\text{if } N = 0 \text{ then } (\delta \ N \ N, [[N, N]])$
 $\text{else } (\text{let } T = \text{init-tab } \delta \ N \ N \text{ in}$
 $\text{let } T' = \text{fill-tab } \delta \ N \ T \ 1 \text{ in}$
 $\text{let } vs = \text{nat2list } 0 \ (2^N - 1) \text{ in}$
 $\text{let } (d, ps) = \text{min-distance-ext } \delta \ N \ (\text{map } (\lambda w. (w, T'(w, \text{list2nat } 0 \ N \ (\text{rem1S } w \ vs)))) \ vs) \text{ in}$
 $(d, \text{map } ((\#) \ N) \ ps)))$

Without loss of generality, the salesman is assumed to reside in the location/vertex N and has to visit each of the locations $0 \dots N - 1$ returning subsequently to N . Now the list(s) forming the value *tsp* $\delta \ N$ represent all of such Hamiltonian cycles that are minimal with respect to δ as the following theorem shows:

theorem *tsp* :

assumes *tsp* $\delta \ N = (d, ps)$

shows $\text{set } ps = \text{opt-Hamiltonian-paths } \delta \ N \ \{0..<N\} \ N \wedge$
 $(\forall p \in \text{set } ps. \text{path-distance } \delta \ p = d)$

<proof>

2.6 An example

An asymmetric edge distance function:

definition $\delta \ (u :: \text{vertex}) \ (v :: \text{vertex}) =$

$(\text{if } u = 0 \text{ then } (\text{if } v = 1 \text{ then } 9$
 $\text{else } (\text{if } v = 2 \text{ then } 10$
 $\text{else } (\text{if } v = 3 \text{ then } 5 \text{ else } 0)))$
 $\text{else } (\text{if } u = 1 \text{ then } (\text{if } v = 0 \text{ then } 13$
 $\text{else } (\text{if } v = 2 \text{ then } 12$
 $\text{else } (\text{if } v = 3 \text{ then } 6 \text{ else } 0)))$
 $\text{else } (\text{if } u = 2 \text{ then } (\text{if } v = 0 \text{ then } 8$
 $\text{else } (\text{if } v = 1 \text{ then } 9$

```

else (if v = 3 then 8 else 0)))
else (if u = 3 then (if v = 0 then 10
else (if v = 1 then 15
else (if v = 2 then 20 else 0)))
else 0))))

```

For this δ there is exactly one optimal Hamiltonian cycle that starts/ends in 3 passing through each of the vertices 0, 1, 2 once:

value *tsp* δ 3

end

References

- [1] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, 1962.
- [2] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. In T. C. Rowan, editor, *Proceedings of the 16th ACM national meeting, ACM 1961, USA*, page 71. ACM, 1961.