

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	3
1.3	Minimal distance extensions	4
1.4	<i>opt-Hamiltonian-paths</i> as a fixed point	5
2	An implementation using dynamic programming	6
2.1	A few preparations	6
2.2	Indexing subsets of $\{0, \dots, n\}$ represented by sorted lists . . .	8
2.3	Computing minimal distance extensions	19
2.4	Tables	23
2.5	Summing up	28
2.6	An example	30

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

using *Hamiltonian-paths-def finite-distinct-list* **by** *fastforce*

lemma *Hamiltonian-paths-finite* :

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

proof(*cases finite V*)

case $a: \text{True}$

show *?thesis*

proof(*rule-tac* $B = (\lambda vs. s \# vs @ [t]) \ ' \{vs. \text{ set } vs \subseteq V \wedge \text{ distinct } vs\}$ **in** *finite-subset*)

show *Hamiltonian-paths* $s \ V \ t \subseteq (\lambda vs. s \# vs @ [t]) \ ' \{vs. \text{ set } vs \subseteq V \wedge \text{ distinct } vs\}$

by(*clarsimp simp: Hamiltonian-paths-def*)

next

from a **show** *finite* $((\lambda vs. s \# vs @ [t]) \ ' \{vs. \text{ set } vs \subseteq V \wedge \text{ distinct } vs\})$

by (*simp add: finite-subset-distinct*)

qed

next

case False

then show *?thesis*

by (*metis Hamiltonian-paths-not-empty equals0I finite.emptyI*)

qed

lemma *Hamiltonian-paths0* :

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

by(*clarsimp simp: Hamiltonian-paths-def*)

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

using *assms Hamiltonian-paths-def hd-Cons-tl set-empty*

proof(*clarsimp simp: Hamiltonian-paths-def*)

qed(*case-tac vs; clarsimp+*)

lemma *Hamiltonian-pathsI* :
assumes $p \in \text{Hamiltonian-paths } v \ (V - \{v\}) \ t$
and $v \in V$
shows $s \# p \in \text{Hamiltonian-paths } s \ V \ t$
using *assms*
by(*clarsimp simp: Hamiltonian-paths-def insert-absorb*)

1.2 Optimal Hamiltonian paths

Extending an edge distance function δ to paths:

fun *path-distance* **where**
 $\text{path-distance } \delta \ [] = (0 :: 'a :: \text{linordered-ab-group-add}) \mid$
 $\text{path-distance } \delta \ [v] = 0 \mid$
 $\text{path-distance } \delta \ (v1 \# v2 \# vs) = \delta \ v1 \ v2 + \text{path-distance } \delta \ (v2 \# vs)$

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 $(\text{path-distance } \delta \ p \leq \text{path-distance } \delta \ p') =$
 $(\text{path-distance } \delta \ (v \# p) \leq \text{path-distance } \delta \ (v \# p'))$
using *assms by (clarsimp simp: Hamiltonian-paths-def)*

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* :
 $\text{opt-Hamiltonian-paths } \delta \ s \ \{\} \ t = \{[s, t]\}$
by(*fastforce simp: opt-Hamiltonian-paths-def Hamiltonian-paths0*)

lemma *opt-Hamiltonian-pathsD* :
 $\text{opt-Hamiltonian-paths } \delta \ s \ V \ t \subseteq \text{Hamiltonian-paths } s \ V \ t$
by(*clarsimp simp: opt-Hamiltonian-paths-def*)

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 $\text{path-distance } \delta \ p = \text{path-distance } \delta \ p'$
using *assms*

by (*simp add: Orderings.order-eq-iff opt-Hamiltonian-paths-def*)

lemma *opt-Hamiltonian-paths-not-empty* :
assumes *finite V*
shows $\exists p. p \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t$
proof –
have *Hamiltonian-paths s V t $\neq$ {}*
by (*metis Hamiltonian-paths-not-empty assms empty-iff*)
with *obtains-MIN[where f=path-distance δ] show ?thesis*
by (*smt (verit, best) Hamiltonian-paths-finite Min.bounded-iff empty-iff finite-imageI image-eqI*
mem-Collect-eq nle-le opt-Hamiltonian-paths-def)
qed

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 $p \in \text{opt-Hamiltonian-paths } \delta \ s \ V \ t$
by (*smt (z3) Hamiltonian-paths-not-empty assms mem-Collect-eq opt-Hamiltonian-paths-def*
opt-Hamiltonian-paths-not-empty order-trans)

1.3 Minimal distance extensions

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

lemma *Min-distance-ext* :
 $(p \in \text{Min-distance-ext } \delta \ v \ S) =$
 $(\exists p' \in S. p = v\#p' \wedge (\forall p'' \in S. \text{path-distance } \delta \ (v\#p') \leq \text{path-distance } \delta \ (v\#p'')))$
proof(*rule iffI*)
qed(*clarsimp simp: Min-distance-ext-def*)+

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)
proof
assume *?L with assms show ?R*
by (*smt (verit, del-insts) DiffI Min-distance-ext nless-le order.strict-trans1*
subset-iff)
next

assume ?R **with** *assms* **show** ?L
by (*smt* (*verit*, *del-insts*) *DiffI* *Min-distance-ext* *nle-le* *nless-le* *subset-iff*)
qed

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 $\text{Min-distance-ext } \delta v S = \text{Min-distance-ext } \delta v T$
proof(*rule set-eqI*)
qed(*simp add: Min-distance-ext-eq' assms*)

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)
proof
show ?L $\subseteq$?R
proof
fix p **assume** $a: p \in \text{opt-Hamiltonian-paths } \delta s V t$
with *assms* **obtain** $v p'$ **where** $b: v \in V p' \in \text{Hamiltonian-paths } v (V - \{v\})$
 $t p = s \# p'$
by (*meson* *Hamiltonian-pathsD* *opt-Hamiltonian-pathsD* *subsetD*)
have $p' \in \text{opt-Hamiltonian-paths } \delta v (V - \{v\}) t$
proof(*rule opt-Hamiltonian-pathsI*)
show $p' \in \text{Hamiltonian-paths } v (V - \{v\}) t$ **by**(*rule b*)
next
from a **and** b
show $\forall p'' \in \text{opt-Hamiltonian-paths } \delta v (V - \{v\}) t. \text{path-distance } \delta p' \leq$
 $\text{path-distance } \delta p''$
by (*metis* (*no-types*, *lifting*) *Hamiltonian-path-distance-Cons-le* *Hamiltonian-pathsI*
mem-Collect-eq opt-Hamiltonian-paths-def)
qed
with a **show** $p \in \text{Min-distance-ext } \delta s (\bigcup v \in V. \text{opt-Hamiltonian-paths } \delta v (V - \{v\}) t)$
proof(*clarsimp simp: Min-distance-ext*)
qed(*metis* (*no-types*, *lifting*) *Hamiltonian-pathsI* $b(1,3)$ *mem-Collect-eq opt-Hamiltonian-paths-def*)
qed
next
show ?R $\subseteq$?L
proof(*clarsimp simp: Min-distance-ext*)
fix $v p$
assume $1: v \in V$
and $2: p \in \text{opt-Hamiltonian-paths } \delta v (V - \{v\}) t$
and $3: \forall v \in V. \forall p' \in \text{opt-Hamiltonian-paths } \delta v (V - \{v\}) t.$

```

      path-distance  $\delta$  (s#p)  $\leq$  path-distance  $\delta$  (s#p')
show s#p  $\in$  opt-Hamiltonian-paths  $\delta$  s V t
proof(rule opt-Hamiltonian-pathsI)
  from 1 and 2 show s#p  $\in$  Hamiltonian-paths s V t
  by (meson Hamiltonian-pathsI opt-Hamiltonian-pathsD subsetD)
next
show  $\forall p' \in$  opt-Hamiltonian-paths  $\delta$  s V t. path-distance  $\delta$  (s#p)  $\leq$  path-distance
 $\delta$  p'
  proof
    fix p' assume a:p'  $\in$  opt-Hamiltonian-paths  $\delta$  s V t
    with 1 obtain v' p'' where p''  $\in$  Hamiltonian-paths v' (V - {v'}) t v'  $\in$ 
    V p' = s#p''
    by (metis Hamiltonian-pathsD emptyE in-mono opt-Hamiltonian-pathsD)
    with 3 and a show path-distance  $\delta$  (s#p)  $\leq$  path-distance  $\delta$  p'
    by (smt (verit) Hamiltonian-path-distance-Cons-le Hamiltonian-pathsI
        mem-Collect-eq opt-Hamiltonian-paths-def)
  qed
qed
qed
qed
qed

```

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

```

```

lemma memS :
  assumes sorted-wrt (>) xs
  shows memS (a :: 'a :: linorder) xs = (a  $\in$  set xs)
  using assms
proof(induct xs)
qed(fastforce+)

```

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}
  using assms
proof(induct xs)

```

qed(*force*+))

lemma *rem1S-length* :
 assumes *sorted-wrt* ($>$) *xs*
 and $a \in \text{set } xs$
 shows $\text{length}(\text{rem1S } a \text{ } xs) + 1 = \text{length } xs$
 using *assms*
proof(*induct xs*)
qed(*fastforce*+))

corollary *rem1S-length'* :
 assumes *sorted-wrt* ($>$) *xs*
 and $a \in \text{set } xs$
 shows $\text{length}(\text{rem1S } a \text{ } xs) < \text{length } xs$
 using *assms rem1S-length* **by** *fastforce*

lemma *sorted-wrt-rem1S* :
 assumes *sorted-wrt* ($>$) *xs*
 shows *sorted-wrt* ($>$) (*rem1S a xs*)
 using *assms*
proof(*induct xs*)
qed(*clarsimp simp: rem1S*)+

lemma *sorted-lists-eq* :
 assumes *sorted-wrt* ($>$) (*as* :: '*a* :: *linorder list*')
 and *sorted-wrt* ($>$) *bs*
 and $\text{set } as \subseteq \text{set } bs$
 and $\text{length } as = \text{length } bs$
 shows $as = bs$
 using *assms*
proof(*induct bs arbitrary: as*)
 case *Nil*
 then show ?*case*
 by *simp*
next
 case 1:(*Cons b bs*)
 then show ?*case*
 proof(*cases as*)
 case *Nil*
 then show ?*thesis*
 using 1(5) **by** *fastforce*
 next
 case (*Cons a as'*)
 with 1 **show** ?*thesis*

```

    proof(clarsimp)
    qed(metis not-less-iff-gr-or-eq subset-insert-iff)
  qed
qed

```

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

```

fun nat2list :: nat  $\Rightarrow$  nat  $\Rightarrow$  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 :
  (nat2list c ix = []) = (ix = 0)
proof(induct c ix rule: nat2list.induct)
  case (1 c ix)
  then show ?case
  proof(cases ix)
    case 0
    then show ?thesis
    by simp
  next
  case (Suc n)
  with 1 show ?thesis
  proof(subst nat2list.simps)
  qed(clarsimp simp del: nat2list.simps)
  qed
qed

```

```

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

```

```

lemma nat2list-ub :
  assumes ix <  $2^b$ 
  shows  $\forall i \in \text{set}(\text{nat2list } c \text{ ix}). i < b + c$ 
  using assms
proof(induct c ix arbitrary: b rule: nat2list.induct)
  case (1 c ix)
  show ?case
  proof(cases b)
    case 0
    with 1 show ?thesis by simp
  next
  case (Suc n)

```

```

    with 1 show ?thesis
    proof(subst nat2list.simps)
    qed(clarsimp simp del: nat2list.simps)
  qed
qed

```

```

lemma nat2list-lb :
   $\forall i \in \text{set}(\text{nat2list } c \text{ } ix). c \leq i$ 
proof(induct c ix rule: nat2list.induct)
  case (1 c ix)
  hence  $ix > 0 \implies \forall i \in \text{set}(\text{nat2list } (c + 1) (ix \text{ div } 2)). c \leq i$  by fastforce
  then show ?case
  proof(subst nat2list.simps)
  qed(simp del: nat2list.simps)
qed

```

```

lemma nat2list-sorted :
  sorted-wrt (>) (nat2list c ix)
proof(induct c ix rule: nat2list.induct)
  case (1 c ix)
  have  $ix > 0 \implies \forall i \in \text{set}(\text{nat2list } (c + 1) (ix \text{ div } 2)). c < i$ 
    by (meson less-iff-succ-less-eq nat2list-lb)
  with 1 show ?case
  proof(subst nat2list.simps)
  qed(simp add: sorted-wrt-append del: nat2list.simps)
qed

```

```

lemma nat2list-pow2[OF refl] :
  assumes  $b = 2^n$ 
  shows  $\text{nat2list } c \text{ } b = [c+n]$ 
  using assms
proof(induct c b arbitrary: n rule: nat2list.induct)
  case (1 c ix)
  then show ?case
  proof(cases n)
    case 0
    with 1(3) and nat2list-Nil show ?thesis
    by simp
  next
    case (Suc m)
    with 1 show ?thesis
    proof(subst nat2list.simps)
    qed(simp del: nat2list.simps)
  qed
qed

```

```

lemma pow2-div2 :
  assumes  $n > 0$ 
  shows  $(2^n + x) \text{ div } 2 = 2^{n-1} + x \text{ div } 2$ 
  by (metis assms add-self-div-2 div-plus-div-distrib-dvd-left dvd-power mult-2-right
power-minus-mult)

```

```

lemma pow2pred-div2 :
 $(2^n - 1) \text{ div } 2 = 2^{n-1} - 1$ 
  by (metis mask-half mask-nat-def)

```

```

lemma nat2list-pred-pow2 :
  assumes  $b = 2^n - 1$ 
  shows  $\text{nat2list } c \ b = \text{rev } [c..<n+c]$ 
  using assms
proof(induct  $c \ b$  arbitrary: n rule: nat2list.induct)
  case ( $1 \ c \ ix$ )
  then show ?case
  proof(cases  $n = 0$ )
    case True
    with  $1$  show ?thesis by simp
  next
  case a:False
  then show ?thesis
  proof(cases  $n = 1$ )
    case True
    with  $1$  show ?thesis by simp
  next
  from  $a$  have  $b : 2^n > 1$  by (simp add: less-trans-Suc)
  case False
  with  $1$  and  $a$  and  $b$  and pow2pred-div2[of n] show ?thesis
  proof(subst nat2list.simps)
  qed(simp add: bot-nat-0.not-eq-extremum upt-rec)
  qed
qed
qed

```

```

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

```

```

lemma nat2list-shift :

```

```

assumes  $ix < 2^b$ 
shows  $\text{nat2list } c (2^b + ix) = (c + b) \# (\text{nat2list } c ix)$ 
using assms
proof(induct c ix arbitrary: b rule: nat2list.induct)
  case ( $1\ c\ ix$ )
  then show ?case
  proof(cases b = 0)
    case True
    with 1 show ?thesis by simp
  next
  case  $2\text{:False}$ 
  show ?thesis
  proof(cases even ix)
    case  $3\text{:True}$ 
    show ?thesis
    proof(cases ix = 0)
      case True
      with 2 show ?thesis
      by (metis add.right-neutral nat2list-Nil nat2list-pow2)
    next
    case  $4\text{:False}$ 
    note pow2-div2[where  $x=ix$  and  $n=b$ ]
    with 1(1)[where  $b=b-1$ ] and 1(3) and 2 and 3 and 4 show ?thesis
    by (simp add: div-greater-zero-iff div-less-iff-less-mult dvd-add-left-iff
power-eq-if)
    qed
  next
  case  $3\text{:False}$ 
  hence  $ix > 0$  by (meson odd-pos)

  with 1(2)[where  $b=b-1$ ] and 1(3) and 2 and 3
  have  $\text{nat2list } (c+1) (2^{b-1} + ix \text{ div } 2) @ [c] = (c + b) \# \text{nat2list } c ix$ 
  proof(subst nat2list.simps[where  $ix=ix$ ])
  qed(smt (verit, best) One-nat-def Suc-pred append-Cons even-add even-power
group-cancel.add1
less-mult-imp-div-less mult-2-right plus-1-eq-Suc power-commutes power-eq-if)

  with 2 and 3 show ?thesis
  by (simp add: pow2-div2)
  qed
qed
qed

```

```

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

```

```

using assms
proof(induct c ix arbitrary: ix' rule: nat2list.induct)
  case (1 c ix)
  then show ?case
  proof(cases even ix)
    case 2:True
    then show ?thesis
    proof(cases even ix')
      case True
      with 1 and 2 show ?thesis
      by (metis dvd-div-eq-iff nat2list.simps nat2list-Nil)
    next
    case False
    with 1 and 2 show ?thesis
    by (metis add-le-same-cancel1 in-set-conv-decomp nat2list.simps nat2list-lb
not-one-le-zero)
  qed
next
  case 2:False
  show ?thesis
  proof(cases even ix')
    case True
    with 1 and 2 show ?thesis
    by (metis add-le-same-cancel1 even-zero in-set-conv-decomp nat2list.simps
nat2list-lb not-one-le-zero)
  next
  have 3:  $\bigwedge n m. n \neq m \implies \text{odd } n \implies \text{odd } m \implies n \text{ div } 2 \neq m \text{ div } 2$ 
  by (metis mod2-eq-if mult-div-mod-eq)
  case False
  with 1(2)[where ix'=ix' div 2] and 1(3) and 2 show ?thesis
  by(simp add: 3)
  qed
qed
qed

```

```

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

```

lemma *list2nat-eq* :

```

list2nat r b l = 2b*r + list2nat 0 b l
proof(induct b arbitrary:l r rule: less-induct)
  case (less x)
  then show ?case
  proof(cases l)
    case Nil
    then show ?thesis by simp
  next
  case (Cons a xs)
  then show ?thesis
  proof(cases x - 1 = a)
    case 1:True
    note less[of x-1 2*r+1 xs] and less[of x-1 1 xs]
    with Cons and 1 show ?thesis
    by (simp add: power-eq-if)
  next
  case 1:False
  note less[of x-1 2*r a#xs]
  with Cons and 1 show ?thesis
  by (simp add: power-eq-if)
  qed
qed
qed

```

```

lemma list2nat-gr0 :
  assumes sorted-wrt (>) l
  and length l > 0
  and  $\forall x \in \text{set } l. x < b$ 
  shows list2nat r b l > 0
  using assms
proof(induct r b l rule: list2nat.induct)
  case (1 r b)
  then show ?case by simp
next
  case (2 r b x l)
  then show ?case
  proof(cases l = [])
    case True
    with 2(5) show ?thesis
  proof(clarsimp simp del: list2nat.simps)
    have  $x < b \implies 0 < \text{list2nat } 0 \text{ } b \text{ } [x]$ 
    proof(induct b rule: less-induct)
      case (less v)
      then show ?case
      proof(subst list2nat.simps)
        qed(simp del: list2nat.simps(2))
      qed

```

```

      thus  $x < b \implies 0 < \text{list2nat } r \ b \ [x]$ 
      by (metis add-is-0 gr-zeroI list2nat-eq)
    qed
  next
    from 2 have  $a:b - 1 \neq x \implies 0 < \text{list2nat } (2 * r) \ (b - 1) \ (x \# l)$  by fastforce
    case False
    with 2 and a show ?thesis
    proof(subst list2nat.simps)
      qed(simp del: list2nat.simps)
    qed
  qed

```

```

lemma list2nat-ub :
  assumes  $\forall x \in \text{set } l. x < b$ 
    and sorted-wrt ( $>$ )  $l$ 
  shows  $\text{list2nat } r \ b \ l < 2^b * r + 2^b$ 
  using assms
proof(induct r b l rule: list2nat.induct)
  case (1 r b)
  then show ?case by simp
next
  case (2 r b x l)
  then show ?case
  proof(cases  $b - 1 = x$ )
    case True
    with 2 show ?thesis
    proof(subst list2nat.simps)
      qed(clarsimp simp del: list2nat.simps simp add: power-eq-if)
    next
      case a:False
      with 2(4) have  $b:\forall x \in \text{set } (x \# l). x < b - 1$  using 2(3) by fastforce
      hence  $c:b > 0$ 
      by (simp add: less-diff-conv)
      hence  $\text{list2nat } (2 * r) \ (b - 1) \ (x \# l) < 2^{b-1} * (2 * r) + 2^{b-1}$ 
      using 2 and a and b by blast
      with a and c show ?thesis
      by (smt (z3) ab-semigroup-mult-class.mult-ac(1) add.commute left-add-mult-distrib
        list2nat.simps(2) mult-2-right mult-numeral-1-right power-minus-mult
        trans-less-add2)
    qed
  qed

```

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  $\text{nat2list } 0 (\text{list2nat } 0 \ b \ l) = l$ 
  using assms
proof(induct b arbitrary: l rule: less-induct)
  case (less b)
  then show ?case
  proof(cases l)
    case Nil
    then show ?thesis by simp
  next
  case (Cons y ls)
  with less(2) have  $a:b > 0$  by simp
  show ?thesis
  proof(cases b - 1 = y)
    case b:True
    with less(2, 3) and Cons and a have  $c:\forall x \in \text{set } ls. x < b-1 \wedge \text{sorted-wrt } (>) \ ls$ 
    by (meson sorted-wrt.simps(2))

    with Cons and a and b
    have  $d:\text{nat2list } 0 (\text{list2nat } 1 \ (b - 1) \ ls) =$ 
       $(b - 1) \# \text{nat2list } 0 (\text{list2nat } 0 \ (b - 1) \ ls)$ 
    proof(subst list2nat-eq)
    qed(metis add.commute add-0 lambda-one less.prem(2) list2nat-ub mult-0-right
      mult-Suc-right
      nat2list-shift sorted-wrt.simps(2))

    note less(1)[where y=b-1 and l=ls]
    with a and b and c have  $\text{nat2list } 0 (\text{list2nat } 0 \ (b - 1) \ ls) = ls$ 
    by (metis diff-less less-numeral-extra(1) nth-mem sorted-wrt-iff-nth-less)
    with a and b and d and Cons and list2nat.simps show ?thesis
    by simp
  next
  case b:False
  have  $\text{nat2list } 0 (\text{list2nat } 0 \ (b - 1) \ (y \# ls)) = y \# ls$ 
  proof(rule less(1))
    from a show  $b - 1 < b$  by simp
  next
    from less(2,3) and a and b and Cons show  $\forall x \in \text{set } (y \# ls). x < b - 1$ 
by fastforce
  next
    from less(3) Cons show sorted-wrt ( $>$ ) ( $y \# ls$ ) by simp
  qed
  with a and b and Cons and list2nat.simps show ?thesis
  by simp
qed
qed

```

qed

corollary *list2nat-surj* :
 assumes $ix < 2^b$
 shows $list2nat\ 0\ b\ (nat2list\ 0\ ix) = ix$
 by (metis assms add.right-neutral list2nat-inj nat2list-inj nat2list-sorted nat2list-ub)

lemma *list2nat-mono* :
 assumes $\forall x \in set\ l. x < b$
 and sorted-wrt $(>)\ l$
 and $length\ l' < length\ l$
 and $set\ l' \subseteq set\ l$
 and sorted-wrt $(>)\ l'$
 shows $list2nat\ r\ b\ l' < list2nat\ r\ b\ l$
 using assms
proof(induct r b l arbitrary: l' rule: list2nat.induct)
 case (1 r b)
 then show ?case by simp
next
 case (2 r b x l)
 hence $a : b > 0$ by simp
 show ?case
proof(cases $b - 1 = x$)
 case b:True
 with 2(3, 4) and a have $c : (\forall x \in set\ l. x < b - 1) \wedge sorted-wrt\ (>)\ l$ by simp
 have $list2nat\ r\ b\ l' < list2nat\ (2 * r + 1)\ x\ l$
proof(cases l')
 case Nil
 with 2 and a and b show ?thesis
proof(subst list2nat-eq[where l=l])
 qed(smt (verit) ab-semigroup-mult-class.mult-ac(1) less-add-same-cancel1
 list2nat.simps(1) mult.commute
 mult-less-mono1 power-minus-mult trans-less-add1 zero-less-numeral
 zero-less-one zero-less-power)
next
 case (Cons a l'')
 show ?thesis
proof(cases $b - 1 = a$)
 case d:True
 have $list2nat\ (2 * r + 1)\ (b - 1)\ l'' < list2nat\ (2 * r + 1)\ (b - 1)\ l$
proof(rule 2(1))
 from a show $b > 0$.
next
 from b show $b - 1 = x$.
next
 from c show $\forall x' \in set\ l. x' < b - 1$ by simp

```

next
  from c show sorted-wrt (>) l by simp
next
  from 2(5) and Cons show length l'' < length l by simp
next
  from 2(6, 7) and Cons and b and d show set l''  $\subseteq$  set l
    using set-subset-Cons by fastforce
next
  from 2(7) and Cons show sorted-wrt (>) l'' by simp
qed
with a and b and d and Cons show ?thesis by simp
next
case d:False
show ?thesis
proof(cases length l'' + 1 = length l)
  case e:True
  have l' = l
  proof(rule sorted-lists-eq)
    show sorted-wrt (>) l using 2(4) by fastforce
  next
    show set l'  $\subseteq$  set l using b c d Cons 2(6) 2(7) by fastforce
  next
    show length l' = length l using Cons e by fastforce
  qed(simp add: 2)
  with a and b and d and Cons show ?thesis
  by (smt (verit, del-Insts) add.commute add-less-cancel-left less-add-same-cancel1
list2nat.simps(2)
list2nat-eq mult.commute mult-less-mono1 zero-less-numeral zero-less-one
zero-less-power)
next
case e:False
have f:list2nat (2 * r + 1) (b - 1) (a # l'') < list2nat (2 * r + 1) (b
- 1) l
proof(rule 2(1))
  from a show b > 0 .
next
  from b show b - 1 = x .
next
  from c show  $\forall x \in \text{set } l. x < b - 1$  by simp
next
  from c show sorted-wrt (>) l by simp
next
  from Cons and e and 2(5) show length (a # l'') < length l by fastforce
next
  from 2(6, 7) and Cons and b c d e show set (a # l'')  $\subseteq$  set l
  proof(clarsimp)
    qed(metis less-imp-add-positive not-add-less1 subset-iff subset-insert)
  next
    from 2(7) and Cons show sorted-wrt (>) (a # l'') by simp

```

```

    qed

    have list2nat (2 * r) (b - 1) (a # l'') < list2nat (2 * r + 1) (b - 1) l
    proof(rule less-trans)
      from list2nat-eq show list2nat (2 * r) (b - 1) (a # l'') < list2nat (2 *
r + 1) (b - 1) (a # l'')
      by (metis Suc-eq-plus1 add.commute add-less-cancel-left lessI mult.commute

          mult-less-mono1 zero-less-numeral zero-less-power)
    next
      from f show list2nat (2 * r + 1) (b - 1) (a # l'') < list2nat (2 * r +
1) (b - 1) l
      by simp
    qed

    with a and b and d and Cons show ?thesis by simp
  qed
qed
qed
with a and b show ?thesis by simp
next
case b:False
with 2(3, 4) and a have c:∀ x∈set(x#l). x < b - 1
  using less-imp-diff-less by force

have list2nat r b l' < list2nat (2 * r) (b - 1) (x # l)
proof(cases l')
  case Nil
  with 2 and a and b and c show ?thesis
  by (smt (verit, ccfv-threshold) One-nat-def Suc-le-eq ab-semigroup-mult-class.mult-ac(1)

      le-add-diff-inverse2 list2nat.simps(1) power-Suc0-right power-add)
next
case (Cons a l'')
then show ?thesis
proof(cases b-1 = a)
  case True
  with a and b and c and Cons and 2(6) show ?thesis
  by (metis less-not-refl list.set-intros(1) subset-iff)
next
case False
with a and b and c and Cons and 2 show ?thesis
  by (metis list2nat.simps(2))
qed
qed
with a b c and 2 show ?thesis
  by (metis list2nat.simps(2))
qed
qed

```

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 *list2nat* *r b* (*rem1S* *u l*) < *list2nat* *r b l*
proof(rule *list2nat-mono*)
qed(simp-all add: *assms rem1S sorted-wrt-rem1S rem1S-length'*)

corollary *list2nat-rem1S-not-in* :
 assumes $u \in \text{set } (\text{nat2list } 0 \text{ } ix)$
 and $ix < 2^N$
 shows $u \notin \text{set}(\text{nat2list } 0 \text{ } (list2nat \ 0 \ N \ (\text{rem1S } u \ (\text{nat2list } 0 \ ix))))$
 using *assms nat2list-sorted*[of 0 *ix*] *nat2list-ub*[of *ix* *N* 0] *sorted-wrt-rem1S rem1S list2nat-inj*
 by (metis *Diff-iff add-cancel-right-right insert-iff*)

corollary *list2nat-rem1S-less* :
 assumes $u \in \text{set } (\text{nat2list } 0 \text{ } ix)$
 and $ix < 2^N$
 shows *list2nat* 0 *N* (*rem1S* *u* (*nat2list* 0 *ix*)) < *ix*
 by (metis *assms add.right-neutral list2nat-rem1S list2nat-surj nat2list-sorted nat2list-ub*)

2.3 Computing minimal distance extensions

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

definition *min-distance-ext* δ *v* *xs* =
 (case *xs* of
 [] $\Rightarrow$ (0, [])
 | (*h*, (*d*, *ps*))#*xs'* $\Rightarrow$ *min-distance-ext'* δ *v* (δ *v* *h* + *d*) (map ((#) *h*) *ps*) *xs'*)

lemma *min-distance-ext'1* :

```

assumes  $\delta \ v \ h + d > md$ 
shows  $\text{min-distance-ext}' \ \delta \ v \ md \ mps \ ((h, d, ps) \# xs) =$ 
 $\text{min-distance-ext}' \ \delta \ v \ md \ mps \ xs$ 
by(simp add: assms)

lemma min-distance-ext'2 :
assumes  $\delta \ v \ h + d = md$ 
shows  $\text{min-distance-ext}' \ \delta \ v \ md \ mps \ ((h, d, ps) \# xs) =$ 
 $\text{min-distance-ext}' \ \delta \ v \ md \ (\text{map } ((\#) \ h) \ ps @ mps) \ xs$ 
by(simp add: assms)

lemma min-distance-ext'3 :
assumes  $\delta \ v \ h + d < md$ 
shows  $\text{min-distance-ext}' \ \delta \ v \ md \ mps \ ((h, d, ps) \# xs) =$ 
 $\text{min-distance-ext}' \ \delta \ v \ (\delta \ v \ h + d) \ (\text{map } ((\#) \ h) \ ps) \ xs$ 
using assms by(fastforce simp add: Let-def)

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

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

lemma min-distance-ext-invD :
assumes min-distance-ext-inv  $\delta \ h \ (d, ps)$ 
shows  $\exists p. p \in \text{set } ps \text{ and } \forall p \in \text{set } ps. \text{path-distance } \delta \ (h \# p) = d$ 
using assms last-in-set
unfolding min-distance-ext-inv-def
by blast+

lemma min-distance-ext'-inv :
assumes min-distance-ext'  $\delta \ v \ md \ mps \ xs = (d, ps)$ 
and min-distance-ext-inv  $\delta \ v \ (md, mps)$ 
and  $\forall (h, d, ps) \in \text{set } xs. \text{min-distance-ext-inv } \delta \ h \ (d, ps)$ 
shows min-distance-ext-inv  $\delta \ v \ (d, ps)$ 
using assms
proof(induct xs arbitrary: md mps)
case Nil
then show ?case by simp
next
case (Cons a xs)
then show ?case
proof clarsimp

```

```

fix h d' l
assume a: a = (h, d', l)
  and b: min-distance-ext-inv  $\delta$  h (d', l)
show ?thesis
proof(cases  $\delta$  v h + d' > md)
  case True
  with Cons and a and b show ?thesis
  by (metis list.set-intros(2) min-distance-ext'1)
next
  case c:False
  show ?thesis
  proof(cases  $\delta$  v h + d' < md)
    from Cons(4) and a have d: min-distance-ext-inv  $\delta$  v ( $\delta$  v h + d', map
((#) h) l)
    by (clarsimp simp: min-distance-ext-inv-def)
    case True
    with Cons and a and d show ?thesis
    by (simp add: min-distance-ext'3)
  next
    case False
    with c have d:  $\delta$  v h + d' = md by simp
    with Cons(3,4) and a have min-distance-ext-inv  $\delta$  v (md, map ((#) h) l
@ mps)
    proof(clarsimp simp: min-distance-ext-inv-def)
      qed(fastforce)
      with Cons and a and d show ?thesis
      by (clarsimp simp: min-distance-ext'2)
    qed
  qed
qed
qed
qed

```

```

lemma min-distance-ext'-eq :
  assumes min-distance-ext'  $\delta$  v md mps xs = (d, ps)
    and min-distance-ext-inv  $\delta$  v (md, mps)
    and  $\forall (h, d', l) \in \text{set } xs. \text{min-distance-ext-inv } \delta \text{ h } (d', l)$ 
  shows (#) v ' set ps = Min-distance-ext  $\delta$  v {p. p  $\in$  set mps  $\vee$  ( $\exists (h, d', l) \in \text{set}$ 
xs. p  $\in$  (#) h ' set l)}
  (is - = Min-distance-ext  $\delta$  v (?s mps xs))
  using assms
proof(induct xs arbitrary: md mps)
  case Nil
  then show ?case
  proof(clarsimp simp: min-distance-ext-inv-def Min-distance-ext-def)
    qed(fastforce)
  next

```

```

case (Cons a xs)
then show ?case
proof clarsimp
  fix h d' l
  assume a: a = (h, d', l)
    and b: min-distance-ext-inv  $\delta$  h (d', l)
  show  $(\#) v \text{ ' set } ps = \text{Min-distance-ext } \delta v \{p. p \in \text{set } mps \vee p \in (\#) h \text{ ' set } l \vee$ 
     $(\exists (h, d', l) \in \text{set } xs. p \in (\#) h \text{ ' set } l)\}$ 
  (is - = Min-distance-ext  $\delta v$  ?t)
  proof(cases  $\delta v h + d' > md$ )
    case c:True
    with Cons a and b show ?thesis
    proof(clarsimp simp: min-distance-ext'1)
      show Min-distance-ext  $\delta v$  (?s mps xs) = Min-distance-ext  $\delta v$  ?t
      proof(rule Min-distance-ext-eq, fast)
        from Cons(3,4) and b and c
        show  $\forall p' \in ?t - ?s \text{ mps } xs. \exists p \in ?s \text{ mps } xs. \text{path-distance } \delta (v \# p) <$ 
path-distance  $\delta (v \# p')$ 
        proof(clarsimp)
          qed(metis (mono-tags, opaque-lifting) min-distance-ext-invD)
        qed
      qed
    next
    case c:False
    show ?thesis
    proof(cases  $\delta v h + d' < md$ )
      case d:True
      with Cons and a and b show ?thesis
      proof(clarsimp simp: min-distance-ext'3 min-distance-ext-inv-def)
        show Min-distance-ext  $\delta v$   $\{p. p \in (\#) h \text{ ' set } l \vee (\exists (h, d', l) \in \text{set } xs. p \in$ 
 $(\#) h \text{ ' set } l)\} =$ 
        Min-distance-ext  $\delta v$  ?t (is Min-distance-ext  $\delta v$  ?u = -)
        proof(rule Min-distance-ext-eq, fast)
          from Cons(3,4) and b and d
          show  $\forall p' \in ?t - ?u. \exists p \in ?u. \text{path-distance } \delta (v \# p) < \text{path-distance } \delta$ 
 $(v \# p')$ 
          proof(clarsimp)
            qed(metis (no-types, opaque-lifting) image-eqI min-distance-ext-invD)
            path-distance.simps(3))
          qed
        qed
      next
      case False
      with c have d:  $\delta v h + d' = md$  by simp
      with Cons(3,4) and a have min-distance-ext-inv  $\delta v$  (md, map  $((\#) h) l$ 
@ mps)
      proof(clarsimp simp: min-distance-ext-inv-def)
        qed(fastforce)

```

```

    with Cons(1,2,4) and a and d show ?thesis
  proof(clarsimp simp: min-distance-ext'2)
  qed(meson)
qed
qed
qed
qed

```

```

lemma min-distance-ext :
  assumes min-distance-ext  $\delta$  v xs = (d, rs)
    and  $\forall (h, d', l) \in \text{set } xs. \text{min-distance-ext-inv } \delta h (d', l)$ 
    and length xs > 0
  shows min-distance-ext-inv  $\delta$  v (d, rs)  $\wedge$ 
    ( $\#$ ) v ' set rs = Min-distance-ext  $\delta$  v {p.  $\exists (h, d', l) \in \text{set } xs. p \in (\#) h$  '
set l}
  (is ?A  $\wedge$  ?B)
proof(cases xs)
case Nil
  with assms(3) show ?thesis by simp
next
case (Cons a ls)
  show ?thesis
  proof
    from assms(1, 2) and Cons show ?A
    proof(clarsimp simp: min-distance-ext-def)
      qed(smt (verit) Collect-cong case-prodI2 case-prod-conv image-iff length-greater-0-conv
list.distinct(1)
      list.map-disc-iff list.set-map min-distance-ext'-inv min-distance-ext-inv-def
path-distance.simps(3))
    next
      from assms(1, 2) and Cons show ?B
      proof(clarsimp simp: min-distance-ext-def)
        qed(smt (verit) Collect-cong case-prodI2 case-prod-conv image-iff length-greater-0-conv
list.distinct(1)
        list.map-disc-iff list.set-map min-distance-ext'-eq min-distance-ext-inv-def
path-distance.simps(3))
      qed
    qed
  end

```

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 $(\text{init-tab } \delta \ (N-1) \ t)((N-1, 0) := (\delta \ (N-1) \ t, [[t]]))$)

lemma *table-inv-init* :

table-inv δ N t (*init-tab* δ N t) 0

proof(*induct* N)

qed(*clarsimp simp: min-distance-ext-inv-def table-inv-def nat2list.simps opt-Hamiltonian-paths0*) +

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

lemma *foldr-upd1* :

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

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

using *assms*

proof(*induct* xs)

qed(*fastforce* +)

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  $\text{foldr } (\lambda v T. T((v, vs) := g v (m T))) xs T (x, vs) = g x (m T)$ 
using assms
proof(induct xs)
qed(clarsimp+)

```

```

lemma table-inv-upd :
  assumes table-inv  $\delta N t T ix$ 
  and  $ix < ix'$ 
  shows table-inv  $\delta N t (\text{upd-tab } \delta N ix' T) ix$ 
using assms
proof(clarsimp simp add: Let-def table-inv-def upd-tab-def)
qed(simp add: foldr-upd1 [where f= $\lambda x. (x, ix')$ ])

```

```

lemma upd-tab-app :
  assumes  $vs = \text{nat2list } 0 ix$ 
  and  $v \in \{0..<N\} - \text{set } vs$ 
  and  $ix < 2^N$ 
  shows  $\text{upd-tab } \delta N ix T (v, ix) =$ 
     $\text{min-distance-ext } \delta v (\text{map } (\lambda w. (w, T(w, \text{list2nat } 0 N (\text{rem1S } w vs)))) vs)$ 
using assms unfolding upd-tab-def Let-def
proof(subst foldr-upd2; clarsimp simp add: memS nat2list-sorted)
qed(rule foldr-upd1; metis Pair-inject list2nat-rem1S-not-in)

```

```

lemma table-inv-upd-ext :
  assumes table-inv  $\delta N t T ix$ 
  and  $ix + 1 < 2^N$ 
  shows table-inv  $\delta N t (\text{upd-tab } \delta N (ix+1) T) (ix+1)$ 
proof(clarsimp simp: table-inv-def)
  fix  $v i$ 
  define  $vs$  where  $vs = \text{nat2list } 0 i$ 
  define  $pix$  where  $pix = (\lambda w. \text{list2nat } 0 N (\text{rem1S } w vs))$ 

  assume  $1:i \leq \text{Suc } ix$ 
  and  $2:v < N$ 
  and  $3:v \notin \text{set } vs$ 
  show  $\text{min-distance-ext-inv } \delta v (\text{upd-tab } \delta N (\text{Suc } ix) T (v, i)) \wedge$ 
     $(\#) v \text{ ' set (snd (upd-tab } \delta N (\text{Suc } ix) T (v, i))) = \text{opt-Hamiltonian-paths } \delta v$ 
     $(\text{set } vs) t$ 
  proof(cases i < ix+1)
    case True
    with vs-def and table-inv-upd and assms(1) and 2 and 3 show ?thesis
    by (metis Diff-iff Suc-eq-plus1 atLeastLessThan-iff lessI less-Suc-eq-le less-eq-nat.simps(1))

```

```

      table-inv-def)
next
  case False
  with 1 have 4:  $i = ix+1$  by simp
  show ?thesis
  proof(cases upd-tab  $\delta$   $N$   $i$   $T$  ( $v$ ,  $i$ ))
    case 5:(Pair  $d$   $rs$ )
    with 4 show ?thesis
    proof(clarsimp)
      note upd-tab-app[OF refl]
      with vs-def pix-def have  $a: \text{min-distance-ext } \delta \ v \ (\text{map } (\lambda w. (w, T \ (w, \text{pix } w))) \ vs) = (d, rs)$ 
      using 2 3 4 5 assms(2) by fastforce
      from 4 and assms(2)
      have  $b: \forall x \in \text{set } vs. \text{pix } x < i \wedge \text{pix } x \leq ix \wedge x \notin \text{set}(\text{nat2list } 0 \ (\text{pix } x))$ 
      by (metis less-eq-iff-succ-less list2nat-rem1S-less list2nat-rem1S-not-in
        pix-def vs-def)

      with 4 and pix-def and vs-def have  $\text{pix-set}: \forall x. \text{set } vs - \{x\} = \text{set}(\text{nat2list } 0 \ (\text{pix } x))$ 
      by (metis Diff-iff add.right-neutral assms(2) list2nat-inj nat2list-sorted
        nat2list-ub rem1S
        sorted-wrt-rem1S)

      from b and 1 and 4 and assms have
         $c: \forall (h, d', l) \in \text{set } (\text{map } (\lambda w. (w, T(w, \text{pix } w))) \ vs). \text{min-distance-ext-inv } \delta \ h \ (d', l)$ 
      proof(clarsimp simp: table-inv-def pix-def vs-def)
        qed(metis DiffI add-cancel-right-right atLeastLessThan-iff bot-nat-0.extremum
          nat2list-ub)

      note min-distance-ext[OF a c]
      with nat2list-Nil and 4 have d:
         $\text{min-distance-ext-inv } \delta \ v \ (d, rs) \wedge (\#) \ v \text{ ' set } rs =$ 
         $\text{Min-distance-ext } \delta \ v \ \{p. \exists x \in \text{set } vs. \text{case } T(x, \text{pix } x) \text{ of } (d', l) \Rightarrow p \in (\#) \ x \text{ ' set } l\}$ 
      by(simp add: pix-def vs-def)

      have  $\text{Min-distance-ext } \delta \ v \ \{p. \exists x \in \text{set } vs. \text{case } T(x, \text{pix } x) \text{ of } (d', l) \Rightarrow p \in (\#) \ x \text{ ' set } l\} =$ 
         $\text{Min-distance-ext } \delta \ v \ (\bigcup x \in \text{set } vs. \text{opt-Hamiltonian-paths } \delta \ x \ (\text{set } vs - \{x\}) \ t)$ 
      (is  $\text{Min-distance-ext } \delta \ v \ ?l = \text{Min-distance-ext } \delta \ v \ ?r$ )

      proof -
        have  $e: \forall x \in \text{set } vs. (\#) \ x \text{ ' set}(\text{snd}(T(x, \text{pix } x))) =$ 
           $\text{opt-Hamiltonian-paths } \delta \ x \ (\text{set}(\text{nat2list } 0 \ (\text{pix } x))) \ t$ 
        by (metis 4 Diff-iff add.right-neutral assms(1,2) atLeastLessThan-iff b
          le0 nat2list-ub table-inv-def
          vs-def)

```

```

have  $\forall p. (p \in ?l) = (p \in ?r)$ 
proof(clarsimp)
qed(metis (lifting) case-prodE case-prodI2 e pix-set snd-eqD)
thus ?thesis
by (metis (mono-tags, lifting) Collect-cong Sup-set-def mem-Collect-eq)
qed

with 4 and d
show min-distance-ext-inv  $\delta v (d, rs) \wedge (\#) v \text{ 'set } rs = \text{opt-Hamiltonian-paths}$ 
 $\delta v (\text{set } vs) t$ 
by (simp add: SUP-cong Union-eq nat2list-Nil opt-Hamiltonian-paths-fix
plus-1-eq-Suc vs-def)
qed
qed
qed
qed

```

```

corollary table-inv-upd-ext' :
assumes table-inv  $\delta N t T (ix-1)$ 
and  $ix < 2^N$ 
and  $ix > 0$ 
shows table-inv  $\delta N t (\text{upd-tab } \delta N ix T) ix$ 
by (metis One-nat-def assms le-add-diff-inverse2 less-eq-Suc-le table-inv-upd-ext)

```

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

```

function fill-tab :: (vertex  $\Rightarrow$  vertex  $\Rightarrow$  distance)  $\Rightarrow$  nat  $\Rightarrow$  table  $\Rightarrow$  nat  $\Rightarrow$  table
where
fill-tab  $\delta N T ix = (\text{if } ix < 2^N - 1 \text{ then } \text{fill-tab } \delta N (\text{upd-tab } \delta N ix T) (ix+1)$ 
else } T)
by pat-completeness auto
termination
by(relation measure ( $\lambda(a, b, c, d). 2^b - d$ ), fastforce+)

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 table-inv  $\delta N t (\text{fill-tab } \delta N T ix) (2^N - 2)$ 
using assms
proof(induct ix arbitrary: T rule: wf-induct[where r=measure( $\lambda x. 2^N - x$ )])
show wf (measure ((-) ( $2^N$ ))) by simp
next
case  $h:(2 x)$ 
have table-inv  $\delta N t (\text{fill-tab } \delta N (\text{upd-tab } \delta N x T) (x+1)) (2^N - 2)$ 
if  $a: x < 2^N - 1$ 

```

```

proof(rule h(1)[rule-format, simplified])
  show  $2^N - (x+1) < 2^N - x$ 
    using a by linarith
  next
    show  $0 < N$  by(rule h(2))
  next
    show  $0 < x+1$  by simp
  next
    fix ix' assume ix' < x+1
    thus table-inv  $\delta$  N t (upd-tab  $\delta$  N x T) ix'
      by (metis One-nat-def Suc-eq-plus1 Suc-pred h.prem1(2,3) less-imp-diff-less
not-less-iff-gr-or-eq
      not-less-less-Suc-eq table-inv-upd table-inv-upd-ext' that)
    qed
  with h(3,4) show ?case by(simp add: fill-tab.simps)
qed

```

2.5 Summing up

definition

```

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

```

proof(cases N = 0)

```

case True
with assms and opt-Hamiltonian-paths0 show ?thesis
  by(fastforce simp: tsp-def)

```

next

```

define vs where vs = nat2list 0 ( $2^N - 1$ )
define pix where pix = ( $\lambda w. list2nat 0 N (rem1S w vs)$ )
define T where T = fill-tab  $\delta$  N (init-tab  $\delta$  N N) 1
have vs1:vs = rev[0.. $N$ ]
  using nat2list-pred-pow2' vs-def by blast
have pix1: $\forall x \in \text{set } vs. \text{pix } x < 2^N - 1 \wedge x \notin \text{set}(\text{nat2list } 0 (\text{pix } x))$ 

```

```

    by (simp add: list2nat-rem1S-less list2nat-rem1S-not-in pix-def vs-def)
  with pix-def and vs-def have pix-set:  $\forall x. \text{set } vs - \{x\} = \text{set}(\text{nat2list } 0 \text{ (pix } x))$ 
  by (metis Diff-iff add.right-neutral diff-less list2nat-inj nat2list-sorted nat2list-ub
    nat-zero-less-power-iff rem1S sorted-wrt-rem1S zero-less-numeral zero-less-one)
  have pix2:  $\forall x \in \text{set } vs. \text{pix } x \leq 2^N - 2$ 
  using length-pos-if-in-set pix1 by fastforce
  from pix1 vs1 pix-set have pix3:  $\forall x \in \text{set } vs. \{0..<N\} - \text{set}(\text{nat2list } 0 \text{ (pix } x))$ 
  =  $\{x\}$ 
  by fastforce

  case a:False
  with T-def have inv:table-inv  $\delta \ N \ N \ T \ (2^N - 2)$ 
  by (simp add: fill-tab table-inv-init)

  note inv[simplified table-inv-def]
  with vs1 pix1 pix2 pix-set have
  c:  $\forall (h, d', l) \in \text{set}(\text{map } (\lambda v. (v, T(v, \text{pix } v))) \text{ vs}). \text{min-distance-ext-inv } \delta \ h \ (d', l)$ 
  by clarsimp

  from a and assms obtain ps' where
  d:min-distance-ext  $\delta \ N \ (\text{map } (\lambda v. (v, T(v, \text{pix } v))) \text{ vs}) = (d, ps') \wedge ps = \text{map}$ 
  ((#) N) ps'
  by (clarsimp simp: tsp-def Let-def split-def T-def pix-def vs-def)

  note min-distance-ext[OF d[THEN conjunct1]]
  with a and c and d and vs1 have e:min-distance-ext-inv  $\delta \ N \ (d, ps') \wedge$ 
  set ps = Min-distance-ext  $\delta \ N \ \{p. \exists (h, d', l) \in \text{set}(\text{map } (\lambda v. (v, T(v, \text{pix } v)))$ 
  vs).  $p \in (\#) \ h \text{ ' set } l\}$ 
  by simp

  with d have 1:  $\forall p \in \text{set } ps. \text{path-distance } \delta \ p = d$ 
  by (smt (verit) imageE image-set min-distance-ext-invD)

  from vs1 pix-set pix2 pix3 set-rev[of vs] inv[simplified table-inv-def]
  have f:  $\forall p. (\exists v \in \{0..<N\}. \text{case } T(v, \text{pix } v) \text{ of } (d', l) \Rightarrow p \in (\#) \ v \text{ ' set } l) =$ 
   $(\exists v \in \{0..<N\}. p \in \text{opt-Hamiltonian-paths } \delta \ v \ (\{0..<N\} - \{v\}) \ N)$ 
  by (smt (verit) atLeastLessThan-upt case-prodE case-prodI2 insertCI rev-rev-ident
    snd-eqD)

  have {p.  $\exists x \in \text{set } vs. \text{case } T(x, \text{pix } x) \text{ of } (d', l) \Rightarrow p \in (\#) \ x \text{ ' set } l$ } =
   $(\bigcup v \in \{0..<N\}. \text{opt-Hamiltonian-paths } \delta \ v \ (\{0..<N\} - \{v\}) \ N)$ 
  proof(rule set-eqI)
  qed(clarsimp simp: f vs1)

  with a and e and 1 show ?thesis
  proof(subst opt-Hamiltonian-paths-fix)
  qed(simp+)

qed

```

2.6 An example

An asymmetric edge distance function:

definition $\delta (u :: vertex) (v :: vertex) =$
 (if $u = 0$ then (if $v = 1$ then 9
 else (if $v = 2$ then 10
 else (if $v = 3$ then 5 else 0)))
 else (if $u = 1$ then (if $v = 0$ then 13
 else (if $v = 2$ then 12
 else (if $v = 3$ then 6 else 0)))
 else (if $u = 2$ then (if $v = 0$ then 8
 else (if $v = 1$ 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 $tsp \ \delta \ 3$

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.