

Knuth–Morris–Pratt String Search

Lawrence C. Paulson

May 26, 2024

Abstract

The naive algorithm to search for a pattern p within a string a compares corresponding characters from left to right, and in case of a mismatch, shifts one position along a and starts again. The worst-case time is $O(|p||a|)$.

Knuth–Morris–Pratt [1] exploits the knowledge gained from the partial match, never re-comparing characters that matched and thereby achieving linear time. At the first mismatched character, it shifts p as far to the right as safely possible. To do so, it consults a precomputed table, based on the pattern p . The KMP algorithm is proved correct.

Contents

1	Knuth-Morris-Pratt fast string search algorithm	3
1.1	General definitions	3
1.2	The Build-table routine	5
1.2.1	The invariant holds after an iteration	5
1.2.2	The build-table loop and its correctness	8
1.2.3	The build-table loop and its correctness	9
1.2.4	Linearity of <i>buildtabW</i>	10
1.3	The actual string search algorithm	11
1.4	Examples	13
1.5	Alternative approach, expressing the algorithms as while loops	14
1.5.1	Linearity of <i>buildtabW</i>	15
1.5.2	The actual string search algorithm	17

Acknowledgements This development closely follows a formal verification of the Knuth–Morris–Pratt algorithm by Jean-Christophe Filliâtre using Why3. Christian Zimmerer reworked the algorithms and termination proofs to use while loops. Tobias Nipkow made helpful suggestions.

1 Knuth-Morris-Pratt fast string search algorithm

Development based on Filliâtre's verification using Why3

Many thanks to Christian Zimmerer for versions of the algorithms as while loops

theory *KnuthMorrisPratt* **imports** *Collections.Diff-Array HOL-Library.While-Combinator*

begin

1.1 General definitions

abbreviation *array* \equiv *new-array*

abbreviation *length-array* $:: 'a$ *array* \Rightarrow *nat* ($\|-\|$)

where *length-array* \equiv *array-length*

notation *array-get* (**infixl** !! 100)

notation *array-set* ($[- ::= -]$ [1000,0,0] 900)

definition *matches* $:: 'a$ *array* \Rightarrow *nat* $\Rightarrow 'a$ *array* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool*

where *matches* *a i b j n* = $(i+n \leq \|a\| \wedge j+n \leq \|b\|$
 $\wedge (\forall k < n. a!!(i+k) = b!!(j+k)))$

lemma *matches-empty* [*simp*]: *matches a i b j 0* $\longleftrightarrow i \leq \|a\| \wedge j \leq \|b\|$

by (*simp add: matches-def*)

lemma *matches-right-extension*:

\llbracket *matches a i b j n*;
Suc (i+n) \leq \|a\|;
Suc (j+n) \leq \|b\|;
a!!(i+n) = b!!(j+n) $\rrbracket \Longrightarrow$
matches a i b j (Suc n)

by (*auto simp: matches-def less-Suc-eq*)

lemma *matches-contradiction-at-first*:

$\llbracket 0 < n; a!!i \neq b!!j \rrbracket \Longrightarrow \neg$ *matches a i b j n*

by (*auto simp: matches-def*)

lemma *matches-contradiction-at-i*:

$\llbracket a!!(i+k) \neq b!!(j+k); k < n \rrbracket \Longrightarrow \neg$ *matches a i b j n*

by (*auto simp: matches-def*)

lemma *matches-right-weakening*:

\llbracket *matches a i b j n*; *n' \leq n* $\rrbracket \Longrightarrow$ *matches a i b j n'*

by (*auto simp: matches-def*)

lemma *matches-left-weakening-add*:

assumes *matches a i b j n k \leq n*

shows *matches a (i+k) b (j+k) (n-k)*

using *assms* **by** (*auto simp: matches-def less-diff-conv algebra-simps*)

lemma *matches-left-weakening*:
assumes *matches a (i - (n - n')) b (j - (n - n')) n*
and $n' \leq n$
and $n - n' \leq i$
and $n - n' \leq j$
shows *matches a i b j n'*
by (*metis assms diff-diff-cancel diff-le-self le-add-diff-inverse2 matches-left-weakening-add*)

lemma *matches-sym*: *matches a i b j n \implies matches b j a i n*
by (*simp add: matches-def*)

lemma *matches-trans*:
 $\llbracket \text{matches } a \ i \ b \ j \ n; \text{ matches } b \ j \ c \ k \ n \rrbracket \implies \text{matches } a \ i \ c \ k \ n$
by (*simp add: matches-def*)

Denotes the maximal $n < j$ such that the first n elements of p match the last n elements of $p[0..j - (1::'a)]$ The first n characters of the pattern have a copy starting at $j - n$.

definition *is-next* :: $'a \ \text{array} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{is-next } p \ j \ n =$
 $(n < j \wedge \text{matches } p \ (j-n) \ p \ 0 \ n \wedge (\forall m. n < m \wedge m < j \longrightarrow \neg \text{matches } p \ (j-m) \ p \ 0 \ m))$

lemma *next-iteration*:
assumes *matches a (i-j) p 0 j is-next p j n j \leq i*
shows *matches a (i-n) p 0 n*
proof –
have *matches a (i-n) p (j-n) n*
using *assms by (auto simp: algebra-simps is-next-def intro: matches-left-weakening*
[**where** $n=j$])
moreover **have** *matches p (j-n) p 0 n*
using *is-next-def assms by blast*
ultimately show *?thesis*
using *matches-trans by blast*
qed

lemma *next-is-maximal*:
assumes *matches a (i-j) p 0 j is-next p j n*
and $j \leq i \ n < m \ m < j$
shows $\neg \text{matches } a \ (i-m) \ p \ 0 \ m$
proof –
have *matches a (i-m) p (j-m) m*
by (*rule matches-left-weakening [where n=j] (use assms in auto)*)
with *assms show ?thesis*
by (*meson is-next-def matches-sym matches-trans*)
qed

Filliâtre's version of the lemma above

corollary *next-is-maximal'*:

assumes *match*: *matches a (i-j) p 0 j is-next p j n*

and *more*: $j \leq i \ i-j < k \ k < i-n$

shows $\neg \text{matches } a \ k \ p \ 0 \ \|p\|$

proof –

have $\neg \text{matches } a \ k \ p \ 0 \ (i-k)$

using *next-is-maximal [OF match] more*

by (*metis add.commute diff-diff-cancel diff-le-self le-trans less-diff-conv less-or-eq-imp-le*)

moreover **have** $i-k < \|p\|$

using *assms* **by** (*auto simp: matches-def*)

ultimately **show** *?thesis*

using *matches-right-weakening nless-le* **by** *blast*

qed

lemma *next-1-0 [simp]*: $\text{is-next } p \ 1 \ 0 \longleftrightarrow 1 \leq \|p\|$

by (*auto simp add: is-next-def matches-def*)

1.2 The Build-table routine

definition *builddtab-step* :: $'a \ \text{array} \Rightarrow \text{nat array} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat array} \times \text{nat} \times \text{nat}$ **where**

builddtab-step p nxt i j =

(if p!!i = p!!j then (nxt[Suc i::=Suc j], Suc i, Suc j)

else if j=0 then (nxt[Suc i::=0], Suc i, j)

else (nxt, i, nxt!!j))

The conjunction of the invariants given in the Why3 development

definition *builddtab-invariant* :: $'a \ \text{array} \Rightarrow \text{nat array} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

builddtab-invariant p nxt i j =

$(\|nxt\| = \|p\| \wedge i \leq \|p\|$

$\wedge j < i \wedge \text{matches } p \ (i-j) \ p \ 0 \ j$

$\wedge (\forall k. \ 0 < k \wedge k \leq i \longrightarrow \text{is-next } p \ k \ (nxt!!k))$

$\wedge (\forall k. \ \text{Suc } j < k \wedge k < \text{Suc } i \longrightarrow \neg \text{matches } p \ (\text{Suc } i - k) \ p \ 0 \ k))$

The invariant trivially holds upon initialisation

lemma *builddtab-invariant-init*: $\|p\| \geq 2 \implies \text{builddtab-invariant } p \ (\text{array } 0 \ \|p\|) \ 1 \ 0$

by (*auto simp: builddtab-invariant-def is-next-def*)

1.2.1 The invariant holds after an iteration

each conjunct is proved separately

lemma *length-invariant*:

shows *let (nxt',i',j') = builddtab-step p nxt i j in \|nxt'\| = \|nxt\|*

by (*simp add: builddtab-step-def*)

lemma *i-invariant*:

assumes *Suc i < m*

shows *let (nxt',i',j') = builddtab-step p nxt i j in i' ≤ m*

```

using assms by (simp add: buildtab-step-def)

lemma ji-invariant:
  assumes buildtab-invariant p nxt i j
  shows let (nxt',i',j') = buildtab-step p nxt i j in j' < i'
proof -
  have j:  $0 < j \implies \text{nxt } !! j < j$ 
    using assms by (simp add: buildtab-invariant-def is-next-def)
  show ?thesis
    using assms by (auto simp add: buildtab-invariant-def buildtab-step-def intro:
order.strict-trans j)
qed

lemma matches-invariant:
  assumes buildtab-invariant p nxt i j and Suc i <  $\|p\|$ 
  shows let (nxt',i',j') = buildtab-step p nxt i j in matches p (i' - j') p 0 j'
  using assms by (auto simp: buildtab-invariant-def buildtab-step-def matches-right-extension
intro: next-iteration)

lemma is-next-invariant:
  assumes buildtab-invariant p nxt i j and Suc i <  $\|p\|$ 
  shows let (nxt',i',j') = buildtab-step p nxt i j in  $\forall k. 0 < k \longrightarrow k \leq i' \longrightarrow \text{is-next}$ 
p k (nxt'!!k)
proof (cases p!!i = p!!j)
  case True
  with assms have matches p (i-j) p 0 (Suc j)
    by (simp add: buildtab-invariant-def matches-right-extension)
  then have is-next p (Suc i) (Suc j)
    using assms by (auto simp: is-next-def buildtab-invariant-def)
  with True assms show ?thesis
    by (simp add: buildtab-invariant-def buildtab-step-def array-get-array-set-other
le-Suc-eq)
  next
  case neq: False
  show ?thesis
  proof (cases j=0)
    case True
    then have  $\neg \text{matches } p (i-j) p 0 (Suc j)$ 
      using matches-contradiction-at-first neq by fastforce
    with True assms have is-next p (Suc i) 0
      unfolding is-next-def buildtab-invariant-def
    by (metis Suc-leI diff-Suc-Suc diff-zero matches-empty nat-less-le zero-less-Suc)
    with assms neq show ?thesis
      by (simp add: buildtab-invariant-def buildtab-step-def array-get-array-set-other
le-Suc-eq)
    next
    case False
    with assms neq show ?thesis
      by (simp add: buildtab-invariant-def buildtab-step-def)
  end
end

```

qed
qed

lemma non-matches-aux:

assumes $Suc (Suc j) < k$ *matches* $p (Suc (Suc i) - k) p 0 k$
shows *matches* $p (Suc i - (k - Suc 0)) p 0 (k - Suc 0)$
using *matches-right-weakening assms* **by** *fastforce*

lemma non-matches-invariant:

assumes *bt: buildtab-invariant* p *next* $i j$ **and** $\|p\| \geq 2$ $Suc i < \|p\|$
shows *let* $(next', i', j') = \text{buildtab-step } p \text{ next } i j$ *in* $\forall k. Suc j' < k \longrightarrow k < Suc i'$
 $\longrightarrow \neg \text{matches } p (Suc i' - k) p 0 k$

proof (*cases* $p!!i = p!!j$)

case *True*

with *non-matches-aux* *bt* **show** *?thesis*

by (*fastforce simp add: Suc-less-eq2 buildtab-step-def buildtab-invariant-def*)

next

case *neq: False*

have $j < i$

using *bt* **by** (*auto simp: buildtab-invariant-def*)

then have *no-match-Sj*: $\neg \text{matches } p (Suc i - Suc j) p 0 (Suc j)$

using *neq* **by** (*force simp: matches-def*)

show *?thesis*

proof (*cases* $j=0$)

case *True*

have $\neg \text{matches } p (Suc (Suc i) - k) p 0 k$

if $Suc 0 < k$ **and** $k < Suc (Suc i)$ **for** k

proof (*cases* $k = Suc (Suc 0)$)

case *True*

with *assms neq that* **show** *?thesis*

by (*auto simp add: matches-contradiction-at-first <j=0>*)

next

case *False*

then have $Suc 0 < k - Suc 0$

using *that* **by** *linarith*

with *bt that* **have** $\neg \text{matches } p (Suc i - (k - Suc 0)) p 0 (k - Suc 0)$

using *True* **by** (*force simp add: buildtab-invariant-def*)

then show *?thesis*

by (*metis False Suc-lessI non-matches-aux that(1)*)

qed

with *True* **show** *?thesis*

by (*auto simp: buildtab-invariant-def buildtab-step-def*)

next

case *False*

then have $0 < j$

by *auto*

have *False* **if** *lessK*: $Suc (next!!j) < k$ **and** $k < Suc i$ **and** *contra*: *matches* $p (Suc i - k) p 0 k$ **for** k

proof (*cases* $Suc j < k$)

```

    case True
    then show ?thesis
      using bt that by (auto simp: buildtab-invariant-def)
next
case False
then have  $k \leq j$ 
  using less-Suc-eq-le no-match-Sj contra by fastforce
obtain  $k'$  where  $k' : k = \text{Suc } k' \ k' < i$ 
  using  $\langle k < \text{Suc } i \rangle$  lessK not0-implies-Suc by fastforce
have is-next  $p \ j \ (\text{nat}!!j)$ 
  using bt that  $\langle j > 0 \rangle$  by (auto simp: buildtab-invariant-def)
with no-match-Sj  $k'$  have  $\neg \text{matches } p \ (j - k') \ p \ 0 \ k'$ 
  by (metis Suc-less-eq  $\langle k \leq j \rangle$  is-next-def lessK less-Suc-eq-le)
moreover
have matches  $p \ 0 \ p \ (i - j) \ j$ 
  using bt buildtab-invariant-def by (metis matches-sym)
then have matches  $p \ (j - k') \ p \ (i - k') \ k'$ 
  using  $\langle j < i \rangle$  False  $k'$  matches-left-weakening
by (smt (verit, best) Nat.diff-diff-eq Suc-leI Suc-le-lessD  $\langle k \leq j \rangle$  diff-diff-cancel
diff-is-0-eq lessI nat-less-le)
moreover have matches  $p \ (i - k') \ p \ 0 \ k'$ 
  using contra  $k'$  matches-right-weakening by fastforce
ultimately show False
  using matches-trans by blast
qed
with assms neq False show ?thesis
  by (auto simp: buildtab-invariant-def buildtab-step-def)
qed
qed

```

```

lemma buildtab-invariant:
  assumes ini: buildtab-invariant  $p \ \text{next } i \ j$ 
  and Suc  $i < \|p\| \ (\text{next } i', j') = \text{buildtab-step } p \ \text{next } i \ j$ 
  shows buildtab-invariant  $p \ \text{next } i' \ j'$ 
  unfolding buildtab-invariant-def
  using assms i-invariant [of concl:  $p \ \text{next } i \ j$ ] length-invariant [of  $p \ \text{next } i \ j$ ]
  ji-invariant [OF ini] matches-invariant [OF ini] non-matches-invariant [OF ini]
  is-next-invariant [OF ini]
  by (simp add: buildtab-invariant-def split: prod.split-asm)

```

1.2.2 The build-table loop and its correctness

Declaring a partial recursive function with the `tailrec` option relaxes the need for a termination proof, because a tail-recursive recursion equation can never cause inconsistency.

1.2.3 The build-table loop and its correctness

partial-function (*tailrec*) *buildtab* :: 'a array \Rightarrow nat array \Rightarrow nat \Rightarrow nat \Rightarrow nat array **where**

buildtab *p* *next* *i* *j* =
 (if *Suc* *i* < $\|p\|$
 then let (*next'*, *i'*, *j'*) = *buildtab-step* *p* *next* *i* *j* in *buildtab* *p* *next'* *i'* *j'*
 else *next*)

declare *buildtab.simps*[code]

Nevertheless, termination must eventually be shown: to use induction to reason about executions. We do so by defining a well founded relation. Termination proofs are by well-founded induction.

definition *rel-buildtab* *m* = *inv-image* (*lex-prod* (*measure* ($\lambda i. m-i$)) (*measure id*)) *snd*

lemma *wf-rel-buildtab*: *wf* (*rel-buildtab* *m*)

unfolding *rel-buildtab-def*
by (*auto intro: wf-same-fst*)

lemma *buildtab-correct*:

assumes *k*: $0 < k \wedge k < \|p\|$ **and** *ini*: *buildtab-invariant* *p* *next* *i* *j*

shows *is-next* *p* *k* (*buildtab* *p* *next* *i* *j* !! *k*)

using *ini*

proof (*induction* (*next*, *i*, *j*) *arbitrary: next* *i* *j* *rule: wf-induct-rule*[*OF wf-rel-buildtab* [*of* $\|p\|$]])

case (*1 next* *i* *j*)

show ?*case*

proof (*cases* *Suc* *i* < $\|p\|$)

case *True*

then obtain *next'* *i'* *j'*

where *eq*: (*next'*, *i'*, *j'*) = *buildtab-step* *p* *next* *i* *j* **and** *invar'*: *buildtab-invariant* *p* *next'* *i'* *j'*

using *1.prem*s *buildtab-invariant* **by** (*metis surj-pair*)

then have $j > 0 \implies \text{next}' !! j < j$

using *1.prem*s

by (*auto simp: buildtab-invariant-def is-next-def buildtab-step-def split: if-split-asm*)

then have *decreasing*: (*next'*, *i'*, *j'*), *next*, *i*, *j* \in *rel-buildtab* $\|p\|$

using *eq True* **by** (*auto simp: rel-buildtab-def buildtab-step-def split: if-split-asm*)

show ?*thesis*

using *1.hyps* [*OF decreasing invar'*] *1.prem*s *eq True*

by(*auto simp add: buildtab.simps*[*of* *p next*] *split: prod.splits*)

next

case *False*

with *1 k* **show** ?*thesis*

by (*auto simp: buildtab-invariant-def buildtab.simps*)

qed

qed

Before building the table, check for the degenerate case

definition *table* :: 'a array \Rightarrow nat array **where**
table *p* = (if $\|p\| > 1$ then *buildtab* *p* (array 0 $\|p\|$) 1 0
else array 0 $\|p\|$)
declare *table-def*[code]

lemma *is-next-table*:
assumes $0 < j \wedge j < \|p\|$
shows *is-next* *p* *j* (*table* *p* !!*j*)
using *buildtab-correct*[of - *p*] *buildtab-invariant-init*[of *p*] *assms* **by** (*simp* *add*:
table-def)

1.2.4 Linearity of *buildtab* *W*

partial-function (*tailrec*) *T-buildtab* :: 'a array \Rightarrow nat array \Rightarrow nat \Rightarrow nat \Rightarrow nat
 \Rightarrow nat **where**

T-buildtab *p* *next* *i* *j* *t* =
(if *Suc* *i* < $\|p\|$
then let (*next'*, *i'*, *j'*) = *buildtab-step* *p* *next* *i* *j* in *T-buildtab* *p* *next'* *i'* *j'* (*Suc* *t*)
else *t*)

lemma *T-buildtab-correct*:
assumes *ini*: *buildtab-invariant* *p* *next* *i* *j*
shows *T-buildtab* *p* *next* *i* *j* *t* $\leq 2 * \|p\| - 2 * i + j + t$
using *ini*
proof (*induction* (*next*, *i*, *j*) *arbitrary*: *next* *i* *j* *t* *rule*: *wf-induct-rule*[*OF* *wf-rel-buildtab*
[*of* $\|p\|$]])

case 1
have *: *Suc* (*T-buildtab* *p* *next'* *i'* *j'* *t*) $\leq 2 * \|p\| - 2 * i + j + t$
if *eq*: *buildtab-step* *p* *next* *i* *j* = (*next'*, *i'*, *j'*) **and** *Suc* *i* < $\|p\|$
for *next'* *i'* *j'* *t*

proof –
have *invar'*: *buildtab-invariant* *p* *next'* *i'* *j'*
using 1.prem *buildtab-invariant* that **by** *fastforce*
then *have* *nextj*: *j* > 0 \implies *next'* !!*j* < *j*
using *eq* 1.prem
by (*auto simp*: *buildtab-invariant-def* *is-next-def* *buildtab-step-def* *split*: *if-split-asm*)
then *have* *decreasing*: ((*next'*, *i'*, *j'*), *next*, *i*, *j*) \in *rel-buildtab* $\|p\|$
using that **by** (*auto simp*: *rel-buildtab-def* *same-fst-def* *buildtab-step-def* *split*:
if-split-asm)

then *have* *T-buildtab* *p* *next'* *i'* *j'* *t* $\leq 2 * \|p\| - 2 * i + j + t$
using 1.hyps *invar'* **by** *blast*

then *show* ?*thesis*
using 1.prem that *nextj*
by (*force simp*: *T-buildtab.simps* [of *p* *next'* *i'* *j'*] *buildtab-step-def* *split*:
if-split-asm)

qed
show ?*case*
using * [where *t* = *Suc* *t*] **by** (*auto simp* *add*: *T-buildtab.simps* *split*: *prod.split*)
qed

lemma *T-buildtab-linear*:

assumes $2 \leq \|p\|$

shows $T\text{-buildtab } p \text{ (array } 0 \|p\|) 1 0 0 \leq 2 * (\|p\| - 1)$

using *assms T-buildtab-correct [OF buildtab-invariant-init, of p 0]* **by** *auto*

1.3 The actual string search algorithm

definition

KMP-step $p \text{ next } a \ i \ j =$

(if $a!!i = p!!j$ then $(\text{Suc } i, \text{Suc } j)$

else if $j=0$ then $(\text{Suc } i, 0)$ else $(i, \text{next}!!j)$)

The conjunction of the invariants given in the Why3 development

definition *KMP-invariant* $:: 'a \text{ array} \Rightarrow 'a \text{ array} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

KMP-invariant $p \ a \ i \ j =$

$(j \leq \|p\| \wedge j \leq i \wedge i \leq \|a\| \wedge \text{matches } a \ (i-j) \ p \ 0 \ j$

$\wedge (\forall k < i-j. \neg \text{matches } a \ k \ p \ 0 \ \|p\|))$

The invariant trivially holds upon initialisation

lemma *KMP-invariant-init*: *KMP-invariant* $p \ a \ 0 \ 0$

by (*auto simp: KMP-invariant-def*)

The invariant holds after an iteration

lemma *KMP-invariant*:

assumes *ini*: *KMP-invariant* $p \ a \ i \ j$

and $j < \|p\|$ **and** $i < \|a\|$

shows let $(i', j') = \text{KMP-step } p \ (\text{table } p) \ a \ i \ j$ in *KMP-invariant* $p \ a \ i' \ j'$

proof (*cases* $a!!i = p!!j$)

case *True*

then show *?thesis*

using *assms* **by** (*simp add: KMP-invariant-def KMP-step-def matches-right-extension*)

next

case *neq*: *False*

show *?thesis*

proof (*cases* $j=0$)

case *True*

with *neq* *assms* **show** *?thesis*

by (*simp add: matches-contradiction-at-first KMP-invariant-def KMP-step-def*

less-Suc-eq)

next

case *False*

then have *is-next*: *is-next* $p \ j \ (\text{table } p \ !!j)$

using *assms* *is-next-table* j **by** *blast*

then have *table* $p \ !!j \leq j$

by (*simp add: is-next-def*)

moreover have *matches* $a \ (i - \text{table } p \ !!j) \ p \ 0 \ (\text{table } p \ !!j)$

by (*meson is-next KMP-invariant-def ini next-iteration*)

moreover

```

have False if  $k < i - \text{table } p \text{ !! } j$  and  $ma: \text{matches } a \text{ } k \text{ } p \text{ } 0 \text{ } \|p\|$  for  $k$ 
proof –
  have  $k \neq i - j$ 
  by (metis KMP-invariant-def add-0 ini j le-add-diff-inverse2 ma matches-contradiction-at-i
neq)
  then show False
  by (meson KMP-invariant-def ini is-nxt k linorder-cases ma next-is-maximal')
qed
ultimately show ?thesis
using neq assms False by (auto simp: KMP-invariant-def KMP-step-def)
qed
qed

```

The first three arguments are precomputed so that they are not part of the inner loop.

```

partial-function (tailrec) search ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat array} \Rightarrow 'a \text{ array} \Rightarrow 'a \text{ array}$ 
 $\Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} * \text{nat}$  where
  search  $m \ n \ \text{nxt } p \ a \ i \ j =$ 
    ( $\text{if } j < m \wedge i < n \text{ then let } (i',j') = \text{KMP-step } p \ \text{nxt } a \ i \ j \text{ in } \text{search } m \ n \ \text{nxt } p$ 
 $a \ i' \ j'$ 
    else  $(i,j)$ )
declare search.simps[code]

```

definition *rel-KMP* $n = \text{lex-prod } (\text{measure } (\lambda i. n - i)) (\text{measure } \text{id})$

lemma *wf-rel-KMP*: *wf* (*rel-KMP* n)
unfolding *rel-KMP-def* **by** (*auto intro: wf-same-fst*)

Also expresses the absence of a match, when $r = \|a\|$

```

definition first-occur ::  $'a \text{ array} \Rightarrow 'a \text{ array} \Rightarrow \text{nat} \Rightarrow \text{bool}$ 
where first-occur  $p \ a \ r = ((r < \|a\| \longrightarrow \text{matches } a \ r \ p \ 0 \ \|p\|) \wedge (\forall k < r. \neg$ 
 $\text{matches } a \ k \ p \ 0 \ \|p\|))$ 

```

lemma *KMP-correct*:

```

assumes ini: KMP-invariant  $p \ a \ i \ j$ 
defines [simp]:  $\text{nxt} \equiv \text{table } p$ 
shows  $\text{let } (i',j') = \text{search } \|p\| \ \|a\| \ \text{nxt } p \ a \ i \ j \text{ in } \text{first-occur } p \ a \ (\text{if } j' = \|p\| \text{ then}$ 
 $i' - \|p\| \text{ else } i')$ 
using ini
proof (induction  $(i,j)$  arbitrary: i j rule: wf-induct-rule[OF wf-rel-KMP [of  $\|a\|$ ]])
case ( $1 \ i \ j$ )
then have ij:  $j \leq \|p\| \ j \leq i \ i \leq \|a\|$ 
and match:  $\text{matches } a \ (i - j) \ p \ 0 \ j$ 
and nomatch:  $(\forall k < i - j. \neg \text{matches } a \ k \ p \ 0 \ \|p\|)$ 
by (auto simp: KMP-invariant-def)
show ?case
proof (cases  $j < \|p\| \wedge i < \|a\|$ )
case True
have first-occur  $p \ a \ (\text{if } j'' = \|p\| \text{ then } i'' - \|p\| \text{ else } i'')$ 

```

```

    if eq: KMP-step p (table p) a i j = (i', j') and eq': search ||p|| ||a|| nxt p a i'
j' = (i'', j'')
    for i' j' i'' j''
    proof -
      have decreasing: ((i', j'), i, j) ∈ rel-KMP ||a||
        using that is-next-table [of j] True
        by (auto simp: rel-KMP-def KMP-step-def is-next-def split: if-split-asm)
      show ?thesis
        using 1.hyps [OF decreasing] 1.prem1 KMP-invariant that True by fastforce
    qed
    with True show ?thesis
      by (smt (verit, best) case-prodI2 nxt-def prod.case-distrib search.simps)
  next
  case False
  have False if matches a k p 0 ||p|| j < ||p|| i = ||a|| for k
  proof -
    have ||p||+k ≤ i
      using that by (simp add: matches-def)
    with that nomatch show False by auto
  qed
  with False ij show ?thesis
    apply (simp add: first-occur-def split: prod.split)
    by (metis le-less-Suc-eq match nomatch not-less-eq prod.inject search.simps)
  qed
qed

```

definition *KMP-search* :: 'a array ⇒ 'a array ⇒ nat × nat **where**
KMP-search p a = search ||p|| ||a|| (table p) p a 0 0
declare *KMP-search-def*[code]

lemma *KMP-search*:
(i, j) = KMP-search p a ⇒ first-occur p a (if j = ||p|| then i - ||p|| else i)
unfolding *KMP-search-def*
using *KMP-correct*[OF *KMP-invariant-init*[of p a]] **by** auto

1.4 Examples

Building the table, examples from the KMP paper and from Cormen et al.

definition *Knuth-pattern* = array-of-list [1,2,3,1,2,3,1,3,1,2::nat]

value *list-of-array* (table *Knuth-pattern*)

definition *CLR-pattern* = array-of-list [1,2,1,2,1,2,1,2,3,1::nat]

value *list-of-array* (table *CLR-pattern*)

Worst-case string searches

definition *bad-list* :: nat ⇒ nat list
where *bad-list* n = replicate n 0 @ [Suc 0]

definition *bad-pattern* = array-of-list (bad-list 1000)

definition *bad-string* = array-of-list (bad-list 2000000)

definition *worse-string* = array-of-list (replicate 2000000 (0::nat))

definition *lousy-string* = array-of-list (concat (replicate 2002 (bad-list 999)))

value *list-of-array* (table bad-pattern)

A successful search

value *KMP-search bad-pattern bad-string*

The search above from the specification alone, i.e. brute-force

lemma *matches bad-string (2000001–1001) bad-pattern 0 1001*

by *eval*

Unsuccessful searches

value *KMP-search bad-pattern worse-string*

The search above from the specification alone, i.e. brute-force

lemma $\forall k < 2000000. \neg \text{matches } \textit{worse-string } k \textit{ bad-pattern } 0 \textit{ 1001}$

by *eval*

value *KMP-search lousy-string bad-string*

lemma $\forall k < \|\textit{lousy-string}\|. \neg \text{matches } \textit{lousy-string } k \textit{ bad-pattern } 0 \textit{ 1001}$

by *eval*

1.5 Alternative approach, expressing the algorithms as while loops

definition *buildtabW*:: 'a array \Rightarrow nat array \Rightarrow nat \Rightarrow nat \Rightarrow nat array option

where

buildtabW *p* *next* *i* *j* \equiv

map-option fst (*while-option* ($\lambda(-, i', -). \text{Suc } i' < \|p\|$)
($\lambda(\text{next}', i', j'). \text{buildtab-step } p \text{ next}' i' j'$)
(*next*, *i*, *j*)))

lemma *buildtabW-halts*:

assumes *buildtab-invariant* *p* *next* *i* *j*

shows $\exists y. \text{buildtabW } p \text{ next } i \text{ j} = \text{Some } y$

proof –

have $\exists y. (\lambda p \text{ next } i \text{ j}. \text{while-option } (\lambda(-, i', -). \text{Suc } i' < \|p\|$
($\lambda(\text{next}', i', j'). \text{buildtab-step } p \text{ next}' i' j'$)
(*next*, *i*, *j*))) *p* *next* *i* *j* = *Some* *y*

proof (*rule measure-while-option-Some*[of $\lambda(\text{next}, i, j). \text{buildtab-invariant } p \text{ next } i$
j - -

$(\lambda p (nxt, i, j). 2 * \|p\| - 2 * i + j) p$,
clarify, rule conjI, goal-cases
case (2 *nxt i j*)
then show ?*case by* (auto simp: *buildtab-step-def buildtab-invariant-def matches-def is-next-def*)
qed (*fastforce simp: assms buildtab-invariant*)+
then show ?*thesis unfolding buildtabW-def by blast*
qed

lemma *buildtabW-correct*:

assumes *k*: $0 < k \wedge k < \|p\|$ **and** *ini*: *buildtab-invariant p nxt i j*
shows *is-next p k* (the (*buildtabW p nxt i j*) !! *k*)

proof –

obtain *nxt' i' j'* **where** †:
while-option ($\lambda(-, i', -). Suc\ i' < \|p\|$) ($\lambda(nxt', i', j'). buildtab-step\ p\ nxt'\ i'\ j'$)
(*nxt, i, j*) = *Some* (*nxt', i', j'*)
using *buildtabW-halts[OF ini]* **unfolding** *buildtabW-def* **by** *fast*
from *while-option-rule[OF - †, of* $\lambda(nxt, i, j). buildtab-invariant\ p\ nxt\ i\ j$
have *buildtab-invariant p nxt' i' j'* **using** *buildtab-invariant ini* **by** *fastforce*
with *while-option-stop[OF †] †* **show** ?*thesis*
using *assms k* **by** (auto simp: *is-next-def matches-def buildtab-invariant-def buildtabW-def*)
qed

1.5.1 Linearity of *buildtabW*

definition *T-buildtabW* :: 'a array \Rightarrow nat array \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat option
where

T-buildtabW p nxt i j t \equiv *map-option* ($\lambda(-, -, -, r). r$)
(*while-option* ($\lambda(-, i, -, -). Suc\ i < \|p\|$)
($\lambda(nxt, i, j, t). let\ (nxt', i', j') = buildtab-step\ p\ nxt\ i\ j\ in\ (nxt', i', j', Suc\ t)$)
(*nxt, i, j, t*))

lemma *T-buildtabW-halts*:

assumes *buildtab-invariant p nxt i j*
shows $\exists y. T-buildtabW\ p\ nxt\ i\ j\ t = Some\ y$

proof –

have $\exists y. (while-option\ (\lambda(-, i, -, -). Suc\ i < \|p\|)$
($\lambda(nxt, i, j, t). let\ (nxt', i', j') = buildtab-step\ p\ nxt\ i\ j\ in\ (nxt', i', j', Suc\ t)$)
(*nxt, i, j, t*) = *Some y*

proof (*intro measure-while-option-Some*[of $\lambda(nxt, i, j, t). buildtab-invariant\ p\ nxt\ i\ j - -$]

($\lambda p (nxt, i, j, t). 2 * \|p\| - 2 * i + j$) *p*, *clarify, rule conjI, goal-cases*)

case (2 *nxt i j t*)

then show ?*case by* (auto simp: *buildtab-step-def buildtab-invariant-def is-next-def*)

qed (*fastforce simp: assms buildtab-invariant split: prod.splits*)+

then show ?*thesis unfolding T-buildtabW-def by blast*

qed

lemma *T-buildtabW-correct*:

assumes *ini*: *buildtab-invariant p next i j*

shows the $(T\text{-buildtab}W\ p\ next\ i\ j\ t) \leq 2 * \|p\| - 2 * i + j + t$

proof –

let $?b = (\lambda(next', i', j', t'). Suc\ i' < \|p\|)$

let $?c = (\lambda(next, i, j, t). let\ (next', i', j') = buildtab\ step\ p\ next\ i\ j\ in\ (next', i', j', Suc\ t))$

let $?s = (next, i, j, t)$

let $?P1 = \lambda(next', i', j', t'). buildtab\ invariant\ p\ next'\ i'\ j'$

$\wedge (if\ Suc\ i' < \|p\| then\ Suc\ t'\ else\ t') \leq 2 * \|p\| - (2 * i' - j') + t'$

let $?P2 = \lambda(next', i', j', t'). buildtab\ invariant\ p\ next'\ i'\ j'$

$\wedge 2 * \|p\| - 2 * i' + j' + t' \leq 2 * \|p\| - 2 * i + j + t$

obtain $next'\ i'\ j'\ t'$ **where** \dagger : $(while\ option\ ?b\ ?c\ ?s) = Some\ (next', i', j', t')$

using *T-buildtabW-halts*[*OF ini*] **unfolding** *T-buildtabW-def* **by** *fast*

have 1 : $(\wedge s. ?P1\ s \implies ?b\ s \implies ?P1\ (?c\ s))$ **proof** (*clarify, intro conjI, goal-cases*)

case $(2\ next_1\ i_1\ j_1\ t_1\ next_2\ i_2\ j_2\ t_2)$

then show $?case$

by (*auto simp: buildtab-step-def split: if-split-asm*)

qed (*insert buildtab-invariant, fastforce split: prod.splits*)

have $P1$: $?P1\ ?s$ **using** *ini* **by** *auto*

from *while-option-rule*[*OF 1 † P1*]

have *invar1*: *buildtab-invariant p next' i' j' and*

invar2: $t' \leq 2 * \|p\| - (2 * i' - j') + t'$ **by** *blast (simp add: while-option-stop*[*OF*

\dagger])

have $?P2\ (next', i', j', t')$ **proof** (*rule while-option-rule*[*OF - †*], *clarify, intro conjI, goal-cases*)

case $(1\ next_1\ i_1\ j_1\ t_1\ next_2\ i_2\ j_2\ t_2)$

with *buildtab-invariant*[*OF 1(3)*] **show** *invar*: $?case$ **by** (*auto split: prod.splits*)

next

case $(2\ next_1\ i_1\ j_1\ t_1\ next_2\ i_2\ j_2\ t_2)$

with $2(4)$ **show** $?case$

by (*auto 0 2 simp: buildtab-step-def buildtab-invariant-def is-next-def split: if-split-asm*)

qed (*use ini in simp*)

with *invar1 invar2 †* **have** $t' \leq 2 * \|p\| - 2 * i + j + t$ **by** *simp*

with \dagger **show** $?thesis$ **by** (*simp add: T-buildtabW-def*)

qed

lemma *T-buildtabW-linear*:

assumes $2 \leq \|p\|$

shows the $(T\text{-buildtab}W\ p\ (array\ 0\ \|p\|)\ 1\ 0\ 0) \leq 2 * (\|p\| - 1)$

using *assms T-buildtabW-correct* [*OF buildtab-invariant-init, of p 0*] **by** *linarith*

1.5.2 The actual string search algorithm

definition $searchW :: nat \Rightarrow nat \Rightarrow nat \text{ array} \Rightarrow 'a \text{ array} \Rightarrow 'a \text{ array} \Rightarrow nat \Rightarrow nat \Rightarrow (nat * nat) \text{ option}$ **where**
 $searchW\ m\ n\ nat\ p\ a\ i\ j = \text{while-option } (\lambda(i, j). j < m \wedge i < n) (\lambda(i, j). \text{KMP-step } p\ \text{next } a\ i\ j) (i, j)$

lemma $searchW\text{-halts}$:

assumes $KMP\text{-invariant } p\ a\ i\ j$
shows $\exists y. searchW\ \|p\| \|a\| (table\ p)\ p\ a\ i\ j = \text{Some } y$
unfolding $searchW\text{-def}$

proof ((*intro measure-while-option-Some*[of $\lambda(i, j). KMP\text{-invariant } p\ a\ i\ j - - \lambda(i, j). 2 * \|a\| - 2 * i + j$], *rule conjI*; *clarify*), *goal-cases*)
case $(2\ i\ j)$

moreover obtain $i'\ j'$ **where** $\dagger: (i', j') = KMP\text{-step } p\ (table\ p)\ a\ i\ j$ **by** (*metis surj-pair*)

moreover have $KMP\text{-invariant } p\ a\ i'\ j'$ **using** \dagger $KMP\text{-invariant}[OF\ 2(1)\ 2(2)\ 2(3)]$ **by** *auto*

ultimately have $2 * \|a\| - 2 * i' + j' < 2 * \|a\| - 2 * i + j$
using $is\text{-next-table}[of\ j\ p]$

by (*auto simp: KMP-invariant-def KMP-step-def matches-def is-next-def split: if-split-asm*)

then show $?case$ **using** \dagger **by** (*auto split: prod.splits*)

qed (*use KMP-invariant assms in fastforce*)+

lemma $KMP\text{-correct}W$:

assumes $ini: KMP\text{-invariant } p\ a\ i\ j$

defines $[simp]: \text{next} \equiv \text{table } p$

shows *let* $(i', j') = \text{the } (searchW\ \|p\| \|a\| \text{next } p\ a\ i\ j)$ *in* $\text{first-occur } p\ a$ *(if* $j' = \|p\|$ *then* $i' - \|p\|$ *else* i' *)*

proof –

obtain $i'\ j'$ **where** $\dagger: \text{while-option } (\lambda(i, j). j < \|p\| \wedge i < \|a\|) (\lambda(i, j). KMP\text{-step } p\ \text{next } a\ i\ j) (i, j) = \text{Some } (i', j')$

using $searchW\text{-halts}[OF\ ini]$ **by** (*auto simp: searchW-def*)

have $KMP\text{-invariant } p\ a\ i'\ j'$

using $\text{while-option-rule}[OF\ -\ \dagger, \text{of } \lambda(i', j'). KMP\text{-invariant } p\ a\ i'\ j']$ *ini KMP-invariant by fastforce*

with \dagger $\text{while-option-stop}[OF\ \dagger]$ **show** $?thesis$

by (*auto simp: searchW-def KMP-invariant-def first-occur-def matches-def*)

qed

definition $KMP\text{-search}W :: 'a \text{ array} \Rightarrow 'a \text{ array} \Rightarrow nat \times nat$ **where**

$KMP\text{-search}W\ p\ a = \text{the } (searchW\ \|p\| \|a\| (table\ p)\ p\ a\ 0\ 0)$

declare $KMP\text{-search}W\text{-def}[code]$

lemma $KMP\text{-search}W$:

$(i, j) = KMP\text{-search}W\ p\ a \implies \text{first-occur } p\ a$ *(if* $j = \|p\|$ *then* $i - \|p\|$ *else* i *)*

unfolding $KMP\text{-search}W\text{-def}$

using $KMP\text{-correct}W[OF\ KMP\text{-invariant-init}[of\ p\ a]]$ **by** *auto*

end

References

- [1] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.