Knuth–Morris–Pratt String Search

Lawrence C. Paulson

March 17, 2025

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

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* ≡ *new-array*
abbreviation *length-array* :: *'a array* ⇒ *nat* (‹‖-‖›)
  where *length-array* ≡ *array-length*
notation *array-get* (**infixl** ‹!!› *100*)
notation *array-set* (‹-[- ::= -]› [*1000*,*0*,*0*] *900*)

definition *matches* :: *'a array* ⇒ *nat* ⇒ *'a array* ⇒ *nat* ⇒ *nat* ⇒ *bool*
 where *matches a i b j n* = (*i*+*n* ≤ ‖*a*‖ ∧ *j*+*n* ≤ ‖*b*‖
    ∧ (∀ *k*<*n*. *a*!!(*i*+*k*) = *b*!!(*j*+*k*)))

lemma *matches-empty* [*simp*]: *matches a i b j 0* ⟷ *i* ≤ ‖*a*‖ ∧ *j* ≤ ‖*b*‖
 by (*simp add*: *matches-def*)

lemma *matches-right-extension*:
 ⟦*matches a i b j n*;
  Suc (*i*+*n*) ≤ ‖*a*‖;
  Suc (*j*+*n*) ≤ ‖*b*‖;
  a!!(*i*+*n*) = *b*!!(*j*+*n*)⟧ ⟹
  matches a i b j (*Suc n*)
 by (*auto simp*: *matches-def less-Suc-eq*)

lemma *matches-contradiction-at-first*:
 ⟦*0* < *n*; *a*!!*i* ≠ *b*!!*j*⟧ ⟹ ¬ *matches a i b j n*
 by (*auto simp*: *matches-def*)

lemma *matches-contradiction-at-i*:
 ⟦*a*!!(*i*+*k*) ≠ *b*!!(*j*+*k*); *k* < *n*⟧ ⟹ ¬ *matches a i b j n*
 by (*auto simp*: *matches-def*)

lemma *matches-right-weakening*:
 ⟦*matches a i b j n*; *n'* ≤ *n*⟧ ⟹ *matches a i b j n'*
 by (*auto simp*: *matches-def*)

lemma *matches-left-weakening-add*:
 assumes *matches a i b j n k*≤*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*: $matches\ a\ i\ b\ j\ n \Longrightarrow matches\ b\ j\ a\ i\ n$
  by (*simp add*: *matches-def*)

lemma *matches-trans*:
  $[\![matches\ a\ i\ b\ j\ n;\ matches\ b\ j\ c\ k\ n]\!] \Longrightarrow matches\ a\ i\ c\ k\ n$
  by (*simp add*: *matches-def*)

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

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

lemma *next-iteration*:
  assumes $matches\ a\ (i-j)\ p\ 0\ j$ $is\text{-}next\ p\ j\ n$ $j \leq i$
  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 $matches\ a\ (i-j)\ p\ 0\ j$ $is\text{-}next\ p\ j\ n$
    and $j \leq i$ $n < m$ $m < j$
  shows $\neg\ matches\ a\ (i-m)\ p\ 0\ m$
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*: $j \le i$ $i{-}j < k$ $k < i{-}n$
shows $\neg$ *matches a k p 0* $\|p\|$
proof $-$
have $\neg$ *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*]: *is-next p 1 0* $\longleftrightarrow$ $1 \le \|p\|$
by (*auto simp*: *is-next-def matches-def*)

1.2 The Build-table routine

definition *buildtab-step* :: *′a array* $\Rightarrow$ *nat array* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat array* $\times$ *nat* $\times$ *nat* **where**
buildtab-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 *buildtab-invariant* :: *′a array* $\Rightarrow$ *nat array* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
buildtab-invariant p nxt i j =
$(\|nxt\| = \|p\| \wedge i \le \|p\|$
$\wedge$ $j{<}i$ $\wedge$ *matches p* $(i{-}j)$ *p 0 j*
$\wedge$ $(\forall k.\ 0 < k \wedge k \le i \longrightarrow$ *is-next p k* (*nxt*!!*k*))
$\wedge$ $(\forall k.$ *Suc j* $< k \wedge k <$ *Suc i* $\longrightarrow \neg$ *matches p* (*Suc i* $-$ *k*) *p 0 k*))

The invariant trivially holds upon initialisation

lemma *buildtab-invariant-init*: $\|p\| \ge 2 \Longrightarrow$ *buildtab-invariant p* (*array 0* $\|p\|$) *1 0*
by (*auto simp*: *buildtab-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') = *buildtab-step p nxt i j in* $\|nxt'\| = \|nxt\|$
by (*simp add*: *buildtab-step-def*)

lemma *i-invariant*:
assumes *Suc i* $< m$
shows *let* (nxt',i',j') = *buildtab-step p nxt i j in* $i' \le m$

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* ⟹ *nxt* !! *j* < *j*
using *assms* **by** (*simp add*: *buildtab-invariant-def is-next-def*)
show *?thesis*
using *assms* **by** (*auto simp*: *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* ∀ *k*. *0* < *k* ⟶ *k* ≤ i' ⟶ *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 ¬ *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*)

  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* − *1*)) *p 0* (*k* − *1*)
  using *matches-right-weakening assms* **by** *fastforce*

lemma *non-matches-invariant*:
  assumes *bt*: *buildtab-invariant p nxt i j* **and** $\|p\| \geq 2$ *Suc i* < $\|p\|$
  shows *let* (*nxt′*,*i′*,*j′*) = *buildtab-step p nxt i j in* ∀ *k*. *Suc j′* < *k* ⟶ *k* < *Suc i′* ⟶ ¬ *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*: *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*: ¬ *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 ¬ *matches p* (*Suc* (*Suc i*) − *k*) *p 0 k*
      if *1* < *k* **and** *k* < *Suc* (*Suc i*) **for** *k*
    proof (*cases k* = *2*)
      case *True*
      with *assms neq that* **show** *?thesis*
        by (*auto simp*: *matches-contradiction-at-first* ‹*j=0*›)
    next
      case *False*
      then have *1* < *k* − *1*
        using *that* **by** *linarith*
      with *bt that* **have** ¬ *matches p* (*Suc i* − (*k* − *1*)) *p 0* (*k* − *1*)
        using *True* **by** (*force simp*: *buildtab-invariant-def*)
      with *False that* **show** *?thesis*
        using *diff-le-self matches-right-weakening* **by** *force*
    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* (*nxt*!!*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 ≤ j
        using less-Suc-eq-le no-match-Sj contra by fastforce
      obtain k′ where k′: k = Suc k′ k′ < i
        using ‹k < Suc i› lessK not0-implies-Suc by fastforce
      have is-next p j (nxt!!j)
        using bt that ‹j>0› by (auto simp: buildtab-invariant-def)
      with no-match-Sj k′ have ¬ matches p (j − k′) p 0 k′
        by (metis Suc-less-eq ‹k ≤ j› 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 ‹j<i› False k′ matches-left-weakening
      by (smt (verit, best) Nat.diff-diff-eq Suc-leI Suc-le-lessD ‹k ≤ j› 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 nxt i j
  and Suc i < ‖p‖ (nxt′,i′,j′) = buildtab-step p nxt i j
  shows buildtab-invariant p nxt′ i′ j′
  unfolding buildtab-invariant-def
  using assms i-invariant [of concl: p nxt i j] length-invariant [of p nxt 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* ⇒ *nat array* ⇒ *nat* ⇒ *nat* ⇒ *nat array* **where**
buildtab p nxt i j =
(*if Suc i* < ‖*p*‖
then let (*nxt'*,*i'*,*j'*) = *buildtab-step p nxt i j in buildtab p nxt' i' j'*
else nxt)
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* (λ*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* ∧ *k* < ‖*p*‖ **and** *ini*: *buildtab-invariant p nxt i j*
shows *is-next p k* (*buildtab p nxt i j* !! *k*)
using *ini*
proof (*induction* (*nxt*,*i*,*j*) *arbitrary*: *nxt i j rule*: *wf-induct-rule*[*OF wf-rel-buildtab* [*of* ‖*p*‖]])
case (*1 nxt i j*)
show *?case*
proof (*cases Suc i* < ‖*p*‖)
case *True*
then obtain *nxt' i' j'*
where *eq*: (*nxt'*, *i'*, *j'*) = *buildtab-step p nxt i j* **and** *invar'*: *buildtab-invariant p nxt' i' j'*
using *1.prems buildtab-invariant* **by** (*metis surj-pair*)
then have *j*>*0* ⟹ *nxt'*!!*j* < *j*
using *1.prems*
by (*auto simp*: *buildtab-invariant-def is-next-def buildtab-step-def split*: *if-split-asm*)
then have *decreasing*: ((*nxt'*, *i'*, *j'*), *nxt*, *i*, *j*) ∈ *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.prems eq True*
by(*auto simp*: *buildtab.simps*[*of p nxt*] *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* ⇒ *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* ∧ *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 *buildtabW*

partial-function (*tailrec*) *T-buildtab* :: *'a array* ⇒ *nat array* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
T-buildtab p nxt i j t =
(*if Suc i* < ‖*p*‖
then let (*nxt'*,*i'*,*j'*) = *buildtab-step p nxt i j in T-buildtab p nxt' i' j'* (*Suc t*)
else t)

lemma *T-buildtab-correct*:
assumes *ini*: *buildtab-invariant p nxt i j*
shows *T-buildtab p nxt i j t* ≤ *2*∗‖*p*‖ − *2*∗*i* + *j* + *t*
using *ini*
proof (*induction* (*nxt*,*i*,*j*) *arbitrary*: *nxt i j t rule*: *wf-induct-rule*[*OF wf-rel-buildtab* [*of* ‖*p*‖]])
case *1*
have ∗: *Suc* (*T-buildtab p nxt' i' j' t*) ≤ *2*∗‖*p*‖ − *2*∗*i* + *j* + *t*
if *eq*: *buildtab-step p nxt i j* = (*nxt'*, *i'*, *j'*) **and** *Suc i* < ‖*p*‖
for *nxt' i' j' t*
proof −
have *invar'*: *buildtab-invariant p nxt' i' j'*
using *1.prems buildtab-invariant that* **by** *fastforce*
then have *nextj*: *j*>*0* ⟹ *nxt'*!!*j* < *j*
using *eq 1.prems*
by (*auto simp*: *buildtab-invariant-def is-next-def buildtab-step-def split*: *if-split-asm*)
then have *decreasing*: ((*nxt'*, *i'*, *j'*), *nxt*, *i*, *j*) ∈ *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 nxt' i' j' t* ≤ *2* ∗ ‖*p*‖ − *2* ∗ *i'* + *j'* + *t*
using *1.hyps invar'* **by** *blast*
then show *?thesis*
using *1.prems that nextj*
by (*force simp*: *T-buildtab.simps* [*of p nxt' i' j'*] *buildtab-step-def split*: *if-split-asm*)
qed
show *?case*
using ∗ [**where** *t* = *Suc t*] **by** (*auto simp*: *T-buildtab.simps split*: *prod.split*)
qed

lemma *T-buildtab-linear*:
  assumes $2 \leq \|p\|$
  shows *T-buildtab p* (*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 nxt a i j =
    (*if* $a!!i = p!!j$ *then* (*Suc i*, *Suc j*)
     else if $j{=}0$ *then* (*Suc i*, 0) *else* (i, $nxt!!j$))

The conjunction of the invariants given in the Why3 development

definition *KMP-invariant* :: *'a array* ⇒ *'a array* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
 KMP-invariant p a i j =
    ($j \leq \|p\| \wedge j{\leq}i \wedge i \leq \|a\| \wedge$ *matches a* $(i{-}j)$ *p* 0 *j*
  $\wedge$ ($\forall k < i{-}j.\ \neg$ *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*: $j < \|p\|$ **and** *i*: $i < \|a\|$
  shows *let* (i',j') = *KMP-step p* (*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-nxt*: *is-next p j* (*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 − *table p* !! j) *p* 0 (*table p* !! j)
      by (*meson is-nxt KMP-invariant-def ini next-iteration*)
    moreover

have *False* **if** k: $k < i - table\ p\ !!\ j$ **and** *ma*: *matches a k p 0* $\|p\|$ **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* :: $nat \Rightarrow nat \Rightarrow nat\ array \Rightarrow 'a\ array \Rightarrow 'a\ array \Rightarrow nat \Rightarrow nat \Rightarrow nat * nat$ **where**
search m n nxt p a i j =
(*if* $j < m \wedge i < n$ *then let* (i',j') = *KMP-step p nxt a i j in search m n nxt p a* i' j'
else (i,j))
declare *search.simps*[*code*]

definition *rel-KMP n* = *lex-prod* (*measure* ($\lambda i.\ n{-}i$)) (*measure 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* :: $'a\ array \Rightarrow 'a\ array \Rightarrow nat \Rightarrow bool$
where *first-occur p a r* = $((r < \|a\| \longrightarrow matches\ a\ r\ p\ 0\ \|p\|) \wedge (\forall k{<}r.\ \neg\ matches\ a\ k\ p\ 0\ \|p\|))$

lemma *KMP-correct*:
assumes *ini*: *KMP-invariant p a i j*
defines [*simp*]: $nxt \equiv table\ p$
shows *let* (i',j') = *search* $\|p\|$ $\|a\|$ *nxt p a i j in first-occur p a* (*if* $j' = \|p\|$ *then* $i' - \|p\|$ *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*: *matches a* $(i - j)$ *p 0 j*
and *nomatch*: $(\forall k{<}i - j.\ \neg\ matches\ a\ k\ p\ 0\ \|p\|)$
by (*auto simp*: *KMP-invariant-def*)
show *?case*
proof (*cases* $j < \|p\| \wedge i < \|a\|$)
case *True*
have *first-occur p a* (*if* $j'' = \|p\|$ *then* $i'' - \|p\|$ *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.prems 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* @ [*1*]

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 ∀ *k*<*2000000*. ¬ *matches worse-string k bad-pattern 0 1001*
by *eval*

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

lemma ∀ *k* < ‖*lousy-string*‖. ¬ *matches lousy-string k bad-pattern 0 1001*
by *eval*

1.5 Alternative approach, expressing the algorithms as while loops

definition *buildtabW*:: ′*a array* ⇒ *nat array* ⇒ *nat* ⇒ *nat* ⇒ *nat array option*
where
buildtabW p nxt i j ≡
map-option fst (*while-option* (λ(-, *i*′, -). *Suc i*′ < ‖*p*‖)
(λ(*nxt*′, *i*′, *j*′). *buildtab-step p nxt*′ *i*′ *j*′)
(*nxt*, *i*, *j*))

lemma *buildtabW-halts*:
assumes *buildtab-invariant p nxt i j*
shows ∃ *y*. *buildtabW p nxt i j* = *Some y*
proof −
have ∃ *y*. (λ*p nxt i j*. *while-option* (λ(-, *i*′, -). *Suc i*′ < ‖*p*‖)
(λ(*nxt*′, *i*′, *j*′). *buildtab-step p nxt*′ *i*′ *j*′)
(*nxt*, *i*, *j*)) *p nxt i j* = *Some y*
proof (*rule measure-while-option-Some*[*of* λ(*nxt*, *i*, *j*). *buildtab-invariant p nxt i j* - -

(λp (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 (λ(-, i', -). *Suc* $i' < \|p\|$) (λ(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* λ(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* ⇒ *nat array* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat option*
where
T-buildtabW p nxt i j t ≡ *map-option* (λ(-, -, -, r). r)
(*while-option* (λ(-, i, -, -). *Suc* $i < \|p\|$)
(λ(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* (λ(-, i, -, -). *Suc* $i < \|p\|$)
(λ(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* λ(nxt, i, j, t). *buildtab-invariant p nxt i j* - -
(λp (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 nxt i j*
  shows $the\ (T\text{-}buildtabW\ p\ nxt\ i\ j\ t) \leq 2{*}\|p\| - 2{*}i + j + t$
proof −
  let $?b = (\lambda(nxt', i', j', t').\ Suc\ i' < \|p\|)$
  let $?c = (\lambda(nxt, i, j, t).\ let\ (nxt', i', j') = buildtab\text{-}step\ p\ nxt\ i\ j\ in\ (nxt', i', j', Suc\ t))$
  let $?s = (nxt, i, j, t)$
  let $?P1 = \lambda(nxt', i', j', t').\ buildtab\text{-}invariant\ p\ nxt'\ i'\ j'$
    $\wedge\ (if\ Suc\ i' < \|p\|\ then\ Suc\ t'\ else\ t') \leq 2 * \|p\| - (2 * i' - j') + t'$
  let $?P2 = \lambda(nxt', i', j', t').\ buildtab\text{-}invariant\ p\ nxt'\ i'\ j'$
    $\wedge\ 2 * \|p\| - 2 * i' + j' + t' \leq 2 * \|p\| - 2 * i + j + t$

  obtain $nxt'\ i'\ j'\ t'$ **where** †: $(while\text{-}option\ ?b\ ?c\ ?s) = Some\ (nxt', i', j', t')$
    using *T-buildtabW-halts*[*OF ini*] **unfolding** *T-buildtabW-def* **by** *fast*
   have *1*: $(\bigwedge s.\ ?P1\ s \Longrightarrow ?b\ s \Longrightarrow ?P1\ (?c\ s))$ **proof** (*clarify*, *intro conjI*, *goal-cases*)
    case $(2\ nxt_1\ i_1\ j_1\ t_1\ nxt_2\ i_2\ j_2\ t_2)$
    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*: $buildtab\text{-}invariant\ p\ nxt'\ i'\ j'$ **and**
      invar2: $t' \leq 2 * \|p\| - (2 * i' - j') + t'$ **by** *blast* (*simp add*: *while-option-stop*[*OF* †])
   have $?P2\ (nxt', i', j', t')$ **proof** (*rule while-option-rule*[*OF* - †], *clarify*, *intro conjI*, *goal-cases*)
      case $(1\ nxt_1\ i_1\ j_1\ t_1\ nxt_2\ i_2\ j_2\ t_2)$
     with *buildtab-invariant*[*OF 1*(*3*)] **show** *invar*: *?case* **by** (*auto split*: *prod.splits*)
    next
      case $(2\ nxt_1\ i_1\ j_1\ t_1\ nxt_2\ i_2\ j_2\ t_2)$
      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 † **show** *?thesis* **by** (*simp add*: *T-buildtabW-def*)
qed

lemma *T-buildtabW-linear*:
  assumes $2 \leq \|p\|$
  shows $the\ (T\text{-}buildtabW\ p\ (array\ 0\ \|p\|)\ 1\ 0\ 0) \leq 2{*}(\|p\| - 1)$
  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* ⇒ *nat* ⇒ *nat array* ⇒ *'a array* ⇒ *'a array* ⇒ *nat* ⇒ *nat* ⇒ (*nat* ∗ *nat*) *option* **where**
searchW m n nxt p a i j = *while-option* (λ(*i*, *j*). *j* < *m* ∧ *i* < *n*) (λ(*i*,*j*). *KMP-step p nxt a i j*) (*i*,*j*)

lemma *searchW-halts*:
assumes *KMP-invariant p a i j*
shows ∃ *y*. *searchW* ‖*p*‖ ‖*a*‖ (*table p*) *p a i j* = *Some y*
unfolding *searchW-def*
proof ((*intro measure-while-option-Some*[*of* λ(*i*,*j*). *KMP-invariant p a i j* - - λ(*i*, *j*). *2* ∗ ‖*a*‖ − *2* ∗ *i* + *j*], *rule conjI*; *clarify*), *goal-cases*)
case (*2 i j*)
moreover obtain *i' j'* **where** †: (*i'*, *j'*) = *KMP-step p* (*table p*) *a i j* **by** (*metis surj-pair*)
moreover have *KMP-invariant p a i' j'* **using** † *KMP-invariant*[*OF 2*(*1*) *2*(*2*) *2*(*3*)] **by** *auto*
ultimately have *2* ∗ ‖*a*‖ − *2* ∗ *i'* + *j'* < *2* ∗ ‖*a*‖ − *2* ∗ *i* + *j*
using *is-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** † **by** (*auto split*: *prod.splits*)
qed (*use KMP-invariant assms* **in** *fastforce*)+

lemma *KMP-correctW*:
assumes *ini*: *KMP-invariant p a i j*
defines [*simp*]: *nxt* ≡ *table p*
shows *let* (*i'*,*j'*) = *the* (*searchW* ‖*p*‖ ‖*a*‖ *nxt p a i j*) *in first-occur p a* (*if j'* = ‖*p*‖ *then i'* − ‖*p*‖ *else i'*)
proof −
obtain *i' j'* **where** †: *while-option* (λ(*i*, *j*). *j* < ‖*p*‖ ∧ *i* < ‖*a*‖) (λ(*i*,*j*). *KMP-step p nxt a i j*) (*i*,*j*) = *Some* (*i'*, *j'*)
using *searchW-halts*[*OF ini*] **by** (*auto simp*: *searchW-def*)
have *KMP-invariant p a i' j'*
using *while-option-rule*[*OF* - †, *of* λ(*i'*, *j'*). *KMP-invariant p a i' j'*] *ini KMP-invariant* **by** *fastforce*
with † *while-option-stop*[*OF* †] **show** *?thesis*
by (*auto simp*: *searchW-def KMP-invariant-def first-occur-def matches-def*)
qed

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

lemma *KMP-searchW*:
(*i*,*j*) = *KMP-searchW p a* ⟹ *first-occur p a* (*if j* = ‖*p*‖ *then i* − ‖*p*‖ *else i*)
unfolding *KMP-searchW-def*
using *KMP-correctW*[*OF KMP-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.