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 $matches\ a\ i\ b\ j\ n = (i{+}n \le \|a\| \land j{+}n \le \|b\| \land (\forall k{<}n.\ a!!(i{+}k) = b!!(j{+}k)))$

lemma *matches-empty* [*simp*]: $matches\ a\ i\ b\ j\ 0 \longleftrightarrow i \le \|a\| \land j \le \|b\|$
 ⟨*proof*⟩

lemma *matches-right-extension*:
 $[\![matches\ a\ i\ b\ j\ n;$
  $Suc\ (i{+}n) \le \|a\|;$
  $Suc\ (j{+}n) \le \|b\|;$
  $a!!(i{+}n) = b!!(j{+}n)]\!] \Longrightarrow$
  $matches\ a\ i\ b\ j\ (Suc\ n)$
 ⟨*proof*⟩

lemma *matches-contradiction-at-first*:
 $[\![0 < n;\ a!!i \ne b!!j]\!] \Longrightarrow \neg\ matches\ a\ i\ b\ j\ n$
 ⟨*proof*⟩

lemma *matches-contradiction-at-i*:
 $[\![a!!(i{+}k) \ne b!!(j{+}k);\ k < n]\!] \Longrightarrow \neg\ matches\ a\ i\ b\ j\ n$
 ⟨*proof*⟩

lemma *matches-right-weakening*:
 $[\![matches\ a\ i\ b\ j\ n;\ n' \le n]\!] \Longrightarrow matches\ a\ i\ b\ j\ n'$
 ⟨*proof*⟩

lemma *matches-left-weakening-add*:
 assumes $matches\ a\ i\ b\ j\ n\ k{\le}n$
 shows $matches\ a\ (i{+}k)\ b\ (j{+}k)\ (n{-}k)$
 ⟨*proof*⟩

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 $matches\ a\ i\ b\ j\ n'$
⟨*proof*⟩

lemma *matches-sym*: $matches\ a\ i\ b\ j\ n \Longrightarrow matches\ b\ j\ a\ i\ n$
⟨*proof*⟩

lemma *matches-trans*:
$[\![matches\ a\ i\ b\ j\ n;\ matches\ b\ j\ c\ k\ n]\!] \Longrightarrow matches\ a\ i\ c\ k\ n$
⟨*proof*⟩

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**
$is\text{-}next\ p\ j\ n =$
$(n < j \land matches\ p\ (j-n)\ p\ 0\ n \land (\forall m.\ n < m \land m < j \longrightarrow \neg\ matches\ p\ (j-m)\ p\ 0\ m))$

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*⟩

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*⟩

Filliâtre's version of the lemma above

corollary *next-is-maximal'*:
assumes *match*: $matches\ a\ (i-j)\ p\ 0\ j\ is\text{-}next\ p\ j\ n$
and *more*: $j \leq i\ i-j < k\ k < i-n$
shows $\neg\ matches\ a\ k\ p\ 0\ \|p\|$
⟨*proof*⟩

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

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* ⇒ *nat array* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
buildtab-invariant p nxt i j =
  (‖*nxt*‖ = ‖*p*‖ ∧ *i* ≤ ‖*p*‖
  ∧ *j*<*i* ∧ *matches p* (*i*−*j*) *p 0 j*
  ∧ (∀ *k*. *0* < *k* ∧ *k* ≤ *i* ⟶ *is-next p k* (*nxt*!!*k*))
  ∧ (∀ *k*. *Suc j* < *k* ∧ *k* < *Suc i* ⟶ ¬ *matches p* (*Suc i* − *k*) *p 0 k*))

The invariant trivially holds upon initialisation

lemma *buildtab-invariant-init*: ‖*p*‖ ≥ *2* ⟹ *buildtab-invariant p* (*array 0* ‖*p*‖) *1 0*
⟨*proof*⟩

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*‖
⟨*proof*⟩

lemma *i-invariant*:
assumes *Suc i* < *m*
shows *let* (*nxt'*,*i'*,*j'*) = *buildtab-step p nxt i j in i'* ≤ *m*
⟨*proof*⟩

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*⟩

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'*
⟨*proof*⟩

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*⟩

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*)

$\langle proof \rangle$

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* $\forall k$. *Suc j'* $< k \longrightarrow k <$ *Suc i'* $\longrightarrow \neg$ *matches p* (*Suc i'* $- k$) *p 0 k*
$\langle proof \rangle$

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'*
$\langle proof \rangle$

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 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* ($\lambda i.\ m-i$)) (*measure id*)) *snd*

lemma *wf-rel-buildtab*: *wf* (*rel-buildtab m*)
$\langle proof \rangle$

lemma *buildtab-correct*:
assumes *k*: $0<k \wedge k <$ $\|p\|$ **and** *ini*: *buildtab-invariant p nxt i j*
shows *is-next p k* (*buildtab p nxt i j* !! *k*)
$\langle proof \rangle$

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*)
⟨*proof*⟩

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* $\leq 2*\|p\| - 2*i + j + t$
⟨*proof*⟩

lemma *T-buildtab-linear*:
assumes $2 \leq \|p\|$
shows *T-buildtab p* (*array 0* $\|p\|$) *1 0 0* $\leq 2*(\|p\| - 1)$
⟨*proof*⟩

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*
∧ (∀ *k* < *i*−*j*. ¬ *matches a k p 0* $\|p\|$))

The invariant trivially holds upon initialisation

lemma *KMP-invariant-init*: *KMP-invariant p a 0 0*
⟨*proof*⟩

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*⟩

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

partial-function (*tailrec*) *search* :: *nat* ⇒ *nat* ⇒ *nat array* ⇒ *'a array* ⇒ *'a array* ⇒ *nat* ⇒ *nat* ⇒ *nat* ∗ *nat* **where**
search m n nxt p a i j =
(*if j* < *m* ∧ *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* (λ*i*. *n*−*i*)) (*measure id*)

lemma *wf-rel-KMP*: *wf* (*rel-KMP n*)
⟨*proof*⟩

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

definition *first-occur* :: *'a array* ⇒ *'a array* ⇒ *nat* ⇒ *bool*
where *first-occur p a r* = ((*r* < ‖*a*‖ ⟶ *matches a r p 0* ‖*p*‖) ∧ (∀ *k*<*r*. ¬ *matches a k p 0* ‖*p*‖))

lemma *KMP-correct*:
assumes *ini*: *KMP-invariant p a i j*
defines [*simp*]: *nxt* ≡ *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'*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma ∀ *k* < ‖*lousy-string*‖. ¬ *matches lousy-string k bad-pattern 0 1001*
⟨*proof*⟩

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*⟩

lemma *buildtabW-correct*:
assumes *k*: *0*<*k* ∧ *k* < ‖*p*‖ **and** *ini*: *buildtab-invariant p nxt i j*
shows *is-next p k* (*the* (*buildtabW p nxt i j*) !! *k*)
⟨*proof*⟩

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 ∃ *y*. *T-buildtabW p nxt i j t* = *Some y*
⟨*proof*⟩

lemma *T-buildtabW-correct*:
assumes *ini*: *buildtab-invariant p nxt i j*
shows *the* (*T-buildtabW p nxt i j t*) ≤ *2*∗‖*p*‖ − *2*∗*i* + *j* + *t*
⟨*proof*⟩

lemma *T-buildtabW-linear*:
assumes *2* ≤ ‖*p*‖
shows *the* (*T-buildtabW p* (*array 0* ‖*p*‖) *1 0 0*) ≤ *2*∗(‖*p*‖ − *1*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)

⟨*proof*⟩

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.