Knuth-Morris-Pratt Algorithm

Jayadev Misra

June 5, 2017

The Knuth-Morris-Pratt string matching algorithm (KMP) locates all occurrences of a pattern string in a text string in linear time (in the combined lengths of the two strings). It is a refined version of a naive algorithm.

1 Informal Description

Let the pattern be "JayadevMisra". Suppose, we have matched the portion "JayadevM" against some part of the text string, but the next symbol in the text differs from 'i', the next symbol in the pattern. The naive algorithm would shift one position beyond 'J' in the text, and start the match all over starting with the first symbol of the pattern. The KMP algorithm is based on the observation that no symbol in the text that we have already matched with "JayadevM" can possibly be the start of a full match: we have just discovered that the there is no match starting at 'J', and there is no match starting at any other symbol because none of them is a 'J'. So, we may skip this entire string in the text and shift to the next symbol beyond "JayadevM" to begin a match.

In general, we will not be lucky enough to skip the entire piece of text that we had already matched, as we could in the case of "JayadevM". For instance, suppose the pattern is "axbcyaxbts", and we have already matched "axbcyaxb"; see Table 1. Suppose the next symbol in the text does not match 't', the next symbol in the pattern. A possible match could begin at the second occurrence of 'a', because the text beginning at that point is "axb", a prefix of the pattern. So, we shift to that position in the text, skip over "axb" and continue matching the remaining portion of the pattern, "cyaxbts"; see Table 1.

2 Algorithm Outline

Notation Let the text be t and the pattern be p. The symbols in a string are indexed starting at 0. Write $t[i]$ for the ith symbol of t, and $t[i..j]$ for the substring of t starting at i and ending just before j, $i \leq j$. Therefore, the length of $t[i..j]$ is $j - i$; it is empty if $i = j$. Similar conventions apply to p. The length of string r is $|r|$.

At any point during the algorithm we have matched a portion of the pattern against the text; we maintain the following invariant where l and r are indices in t defining the two ends of the matched portion.

KMP-INV:
$l \leq r \ \wedge \ t[l..r] = p[0..r-l]$, and
all occurrences of p starting prior to l in the text have been located.

The invariant is established initially by setting

$l, r := 0, 0$

In subsequent steps we compare the next symbols from the text and the pattern. If there is no next symbol in the pattern, we have found a match, and we discuss what to do next below. For the moment, assume that p has a next symbol, $p[r-l]$.

Below, $b \ \rightarrow \ C$, where b is a predicate and C a sequence of statements, is known as a *guarded command*; b is the guard and C the command. Command C is executed only if b holds. Below, exactly one guard is true in any iteration.

$t[r] = p[r-l] \qquad \rightarrow \ r := r+1$
{ more text has been matched }
$t[r] \neq p[r-l] \ \wedge \ r = l \quad \rightarrow \ l := l+1; \ r := r+1$
{ we have an empty string matched so far;
the first pattern symbol differs from the next text symbol }
$t[r] \neq p[r-l] \ \wedge \ r > l \quad \rightarrow \ l := l'$
{ a nonempty prefix of p has matched but the next symbols don't }

Note that in the third case, r is not changed; so, none of the symbols in $t[l'..r]$ will be scanned again. The question (in the third case) is, what is l'? Abbreviate $t[l..r]$ by v and $t[l'..r]$ by u. Now v is a prefix of p and also u is a prefix of p. Since $l' > l$, u is a shorter prefix than v. Therefore, u is a proper prefix of v. Further, since their right ends match, $t[l'..r]$ is a proper suffix of $t[l..r]$, i.e., u is a proper suffix of v. *We may set l' to any value such that u is a proper prefix and suffix of v.*

Example An example is in schematic form in Table 1. Here, we have already matched the prefix "axbcyaxb", which is v. There is a mismatch in the next symbol. We shift the pattern so that the prefix "axb", which is u, is aligned with a portion of the text that matches it. □

There could be many strings that are both prefix and suffix of v; which one should we choose as u? The only safe choice is the *longest* such string so that we do not miss a possible match. Call the longest string that is both a prefix and suffix of v the *core* of v. Then $l' := l + (\text{length of } v) - (\text{length of core of } v)$. Since v is a prefix of p, we precompute the cores of all prefixes of the pattern, so that we may compute l' whenever there is a failure in the match. After the pattern has been completely matched, we record this fact and let $l' = l + (\text{length of } p) - (\text{length of core of } p)$.

index	l					l'			r						
text	a	x	b	c	y	a	x	b	z	-	-	-	-	-	-
pattern	a	x	b	c	y	a	x	b	t	s					
newmatch						a	x	b	c	y	a	x	b	t	s

Table 1: Matching in the KMP algorithm

KMP runs in linear time. This is because $l + r$ increases in each step (in the last case, $l' > l$). Both l and r are bounded by the length of the text string; so the number of steps is bounded by a linear function of the length of text. The core computation, given in Section 4, is linear in the size of the pattern. So, the whole algorithm is linear.

In the rest of the note we develop the underlying theory (Section 3) and a procedure to compute the cores of all prefixes of any string (Section 4).

3 Underlying theory

3.1 Bifix

For strings u and v, write $u \preceq v$ to mean that u is both a prefix and a suffix of v and call u a *bifix* of v. Observe that u is a bifix of v iff u is a prefix of v and the reverse of u is a prefix of the reverse of v. As is usual, we write $u \prec v$ to mean that $u \preceq v$ and $u \neq v$.

The following properties of $\preceq$ follow; you are expected to develop the proofs. Henceforth, u and v denote arbitrary strings and ϵ is the empty string.

Exercise 1

1. $\epsilon \preceq u$.
2. $\preceq$ is a partial order. Use the fact that prefix relation is a partial order.
3. There is a total order among all bifixes of v, for any v, i.e.,

$$(u \preceq v \ \wedge \ w \preceq v) \ \Rightarrow \ (u \preceq w \ \vee \ w \preceq u) \qquad \square$$

3.2 Core

For any nonempty v, *core* of v, written as $c(v)$, is its longest bifix. The core is defined for every non-empty string because there is at least one string, namely ϵ, that is a bifix of every nonempty string.

a	ab	abb	abba	abbab	abbabb	abbabba	abbabab
ϵ	ϵ	ϵ	a	ab	abb	abba	ab

Table 2: Example of cores

Example Cores of several strings are given in Table 2.

The traditional way to formally define core of v, $c(v)$, is as follows: (1) $c(v) \prec v$, and (2) for any w where $w \prec v$, $w \preceq c(v)$. We give a different, though equivalent, definition that is more convenient for formal manipulations (some readers may discern a Galois connection). For any u and v, $v \neq \epsilon$,

(core definition): $u \preceq c(v) \;\equiv\; u \prec v$

It follows, by replacing u with $c(v)$, that $c(v) \prec v$. So, $|c(v)| < |v|$. Every non-empty string v has a core because $\epsilon \prec v$. The core is unique. To see this, let r and s be cores of v; we show that $r = s$. From the definition using r and s for $c(v)$, conclude $u \preceq r \;\equiv\; u \prec v$ and $u \preceq s \;\equiv\; u \prec v$. That is, $u \preceq r \;\equiv\; u \preceq s$, for all u. Setting u to r, we get $r \preceq r \;\equiv\; r \preceq s$, i.e., $r \preceq s$. Similarly, deduce $s \preceq r$. So, $r = s$ from the antisymmetry of $\preceq$. □

Write $c^i(v)$ for i-fold application of c to v, i.e., $c^i(v) = \overbrace{c(c(..(c}^{i \text{ times}}(v)..)))$ and $c^0(v) = v$. Since $|c(v)| < |v|$, $c^i(v)$ is defined only for some i, not necessarily all i, in the range $0 \leq i \leq |v|$. Note that, $c^{i+1}(v) \prec c^i(v) \ldots c^1(v) \prec c^0(v) = v$.

Exercise 2

1. Let u be a longer string than v. Is $c(u)$ necessarily longer than $c(v)$?
2. Show that the core function is monotonic, that is,

   $$u \preceq v \;\Rightarrow\; c(u) \preceq c(v)$$

3. What is $c^i(ab)^n$ for $i \leq n$? □

3.3 A characterization of bifix using the core function

The following proposition says that every bifix of v can be obtained by applying function c a sufficient number of times to v.

P1: For any u and v, $u \preceq v \ \equiv \ \langle \exists i : 0 \leq i : u = c^i(v) \rangle$.
Proof: The proof is by induction on the length of v.

- $|v| = 0$:

$$\begin{array}{ll} & u \preceq v \\ \equiv & \{|v| = 0, \text{ i.e., } v = \epsilon\} \\ & u = \epsilon \ \wedge \ v = \epsilon \\ \equiv & \{\text{definition of } c^0\text{: } v = \epsilon \ \Rightarrow \ c^i(v) \text{ is defined for } i = 0 \text{ only}\} \\ & \langle \exists i : 0 \leq i : u = c^i(v) \rangle \end{array}$$

- $|v| > 0$:

$$\begin{array}{ll} & u \preceq v \\ \equiv & \{\text{definition of } \preceq\} \\ & u = v \ \vee \ u \prec v \\ \equiv & \{\text{definition of core}\} \\ & u = v \ \vee \ u \preceq c(v) \\ \equiv & \{|c(v)| < |v|\text{; apply induction hypothesis on second term}\} \\ & u = v \ \vee \ \langle \exists i : 0 \leq i : u = c^i(c(v)) \rangle \\ \equiv & \{\text{rewrite}\} \\ & u = c^0(v) \ \vee \ \langle \exists i : 0 < i : u = c^i(v) \rangle \\ \equiv & \{\text{rewrite}\} \\ & \langle \exists i : 0 \leq i : u = c^i(v) \rangle \end{array}$$ □

Corollary For any u and v, $v \neq \epsilon$,

$$u \prec v \ \equiv \ \langle \exists i : 0 < i : u = c^i(v) \rangle$$

3.4 Incremental computation of core

The following proposition characterizes the core of us, where s is a symbol, in terms of the bifixes of u.

P2: ws is the core of us iff w is the longest bifix of u whose successor symbol is s. If no bifix of u has s as its successor symbol, $c(us) = \epsilon$.
Proof: We show below that vs is a bifix of us iff v is a bifix of u whose successor symbol is s. This proves (P2).

$$\begin{array}{ll} & vs \prec us \\ \equiv & \{vs \text{ is a proper prefix and suffix of } us\} \\ & v \text{ is a proper prefix and suffix of } u\text{, successor symbol of } v \text{ is } s \\ \equiv & \{\text{definition of } \prec\} \\ & v \prec u\text{, successor symbol of } v \text{ is } s \end{array}$$

4 Computing cores

The KMP algorithm needs the core of string u when the pattern match fails after having matched u. String u is a prefix of the pattern string p. Therefore, we pre-compute the cores of *all* non-empty prefixes of p.

4.1 Implementation

First, let us decide how to store the cores. Any core of a prefix of p is a prefix of p. So, instead of storing the actual cores we may simply store their lengths. Let d be an array where $d[k]$, for $k > 0$, is the length of the core of $p[0..k]$, i.e., $d[k] = |c(p[0..k])|$. ($d[0]$ has no value because ϵ does not have a core.) Therefore, $d[k] < k$ for all $k > 0$, which can also be proved as an invariant of the program given below.

The computation is based on Proposition P2 of Section 3.4. After computing the cores up to some prefix u to compute the core of us, where s is the successor symbol of u, consider all bifixes of u in the order of decreasing lengths until one is found whose successor symbol is s. The successively smaller bifixes of u are $c(u), c^2(u), \ldots$, from Proposition P1 of Section 3.4. If no such bifix is found, $c(us) = \epsilon$.

The following program includes variables i and j in addition to the pattern p and array d, where $u = p[0..j]$, $s = p[j]$, the cores of all prefixes of up to and including u are known (and stored in d), and i is the length of the bifix of u that is under consideration for the computation of $c(us)$, according to Proposition P2. The successor symbol of this bifix is $p[i]$.

$j := 1;\ d[1] := 0;\ i := 0;$
while $j < |p|$ **do**
  S_1:: $p[i] = p[j] \quad \rightarrow \quad i, j := i + 1, j + 1; d[j] := i$
  S_2:: $p[i] \neq p[j] \wedge i \neq 0 \ \rightarrow \quad i := d[i]$
  S_3:: $p[i] \neq p[j] \wedge i = 0 \ \rightarrow \quad j := j + 1; d[j] := 0$
endwhile

4.2 Analysis of the running time of core computation

We show that the program for core computation runs in linear time in the length of pattern p. Specifically, execution of each guarded command increases $2j - i$. Since $j \leq |p|$ and $i \geq 0$ (prove these as invariants), $2j - i$ never exceeds $2|p|$. Initially, $2j - i = 2$. Therefore, the number of executions of all guarded commands is $O(|p|)$.

We show

$$\{2j - i = n\} \quad \text{right side of } \mathrm{S}_k \quad \{2j - i > n\},\ 1 \leq k \leq 3$$

Each proof starts with this goal and successively reduces the number of statements in the assertion until a only a logical proposition, *true*, is to be established.

• Proof for S_1:

$\{2j - i = n\}\ i, j := i + 1, j + 1; d[j] := i\ \{2j - i > n\}$
, goal
$\{2j - i = n\}\ i, j := i + 1, j + 1\ \{2j - i > n\}$
, axiom of assignment
$2j - i = n\ \Rightarrow\ 2(j + 1) - (i + 1) > n$
, axiom of assignment
true , arithmetic □

• Proof for S_2:

$\{2j - i = n\}\ i := d[i]\ \{2j - i > n\}$
, goal
$2j - i = n\ \Rightarrow\ 2j - d[i] > n$, axiom of assignment
$2j - i < 2j - d[i]$, arithmetic
$d[i] < i$, arithmetic
true , program invariant □

• Proof for S_3:

$\{2j - i = n\}\ j := j + 1; d[j] := 0\ \{2j - i > n\}$
, goal
$\{2j - i = n\}\ j := j + 1\ \{2j - i > n\}$
, axiom of assignment
$\{2j - i = n\}\ \{2(j + 1) - i > n\}$
, axiom of assignment
$2j - i = n\ \Rightarrow\ 2(j + 1) > n$, using $i = 0$ from the guard of S_3
true , arithmetic □

Exercise 3

1. Show that you can match pattern p against text t by computing the cores of all prefixes of pt (pt is the concatenation of p and t).

2. Define u to be the k-core of string v, where $k \geq 0$ and $v \neq \epsilon$, if $u \prec v$, u's length is at most k and u is the longest string with this property. Show that the k-core is well-defined. Devise an algorithm to compute the k-core of a string for a given k. □

Acknowledgement I am grateful to Steve Li for pointing out an error in the text and that the definition of core is motivated by Galois connection.