ICAI 2023
12th International Conference on Applied Informatics
Eger, Hungary, 2023

# Properties of strings used in the analysis of the Knuth-Morris-Pratt algorithm*

**Tibor Ásványi**[a]

[a]Eötvös Loránd University, Faculty of Informatics
asvanyi@inf.elte.hu

## Abstract

This paper is about the string-matching problem. First, we summarise its naive solution. Next, we analyse an outstanding method, the Knuth-Morris-Pratt (KMP) algorithm.

Given alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ $(d \in \mathbb{N}_+)$, text $T : \Sigma[n]$ and pattern $P : \Sigma[m]$, we search for all the occurrences of $P[0\,..\,m)$ in $T[0\,..\,n)$, provided that $0 < m \leq n$. (We index from 0, $P[0\,..\,m) = P[0\,..\,m-1]$ and similarly for $T$.)

**Definition 1.** $s \in 0\,..\,n-m$ is a *possible shift* of $P$ on $T$. It is a *valid shift*, **if** $T[s\,..\,s+m) = P[0\,..\,m)$. Otherwise, it is an *invalid shift*.

**Problem 2** (String-matching)**.** *Compute the set of valid shifts of $P$ on $T$.*

The naive string-matching (Brute-Force) algorithm checks each possible shift in order and collects the valid shifts into the set $S$ with maximal (i.e. worst-case) time complexity $MT(n,m) \in \Theta((n-m+1) * m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. [2]

More advanced methods like the different versions of Boyer-Moore [1, 3], the Rabin-Karp and the KMP [2] algorithms use information gained about the pattern and the text. They do not check each possible shift of $P$ on $T$ but often make a jump in $T$.

KMP is outstanding because it runs in $\Theta(n)$ time on all the possible inputs, and it never backtracks on $T$, which means that it is easy to implement on sequential files. Traditionally KMP is introduced as an extremely efficient simulation of *string matching with finite automata* [2]. Here we avoid these automata and start with the analysis of $P$ and $T$, i.e. the strings.

*This research was supported by the author's Institute

**Definition 3.** If $x$ and $y$ are strings, $x+y$ is their concatenation; $x \sqsupseteq y$ ($x$ is *suffix* of $y$) means that $\exists z$ string, so that $z+x=y$; $x \sqsupset y$ ($x$ is *proper suffix* of $y$) means that $x \sqsupseteq y \wedge x \neq y$. *Prefix* ($\sqsubseteq$) and *proper prefix* ($\sqsubset$) are defined similarly. We also use the following abridgement. $P_{:j} = P[0\,..\,j) \quad (j \in 0\,..\,m)$.

In the rest of this paper, we suppose that $P$ and $T$ and their lengths, $m$ and $n$ are fixed where $0 < m \leq n$. We are going to analyze the main procedure of the KMP algorithm. Working with the author's invariant, we prove only its partial correctness in this Abstract. Its termination and efficiency will be considered later. It works as follows, with

invariant $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \wedge S =$ the set of valid shifts in $[0\,..\,i{-}j)$.

(1) $i := j := 0; S := \{\}$.
(2) Here, the invariant is true.
If $i < n$, go to step (3).
Otherwise, $S =$ the set of valid shifts in $[0\,..\,n-j)$. And $j < m$, consequently $n-j > n-m$. As a result, $S =$ the set of valid shifts in $0\,..\,n-m$, because there is no possible shift greater than $n-m$. Therefore the string-matching problem is solved and we STOP.
(3) If $P[j] \neq T[i]$, then $i-j$ is not a valid shift. Go to step (7)

**Lemma 4.** $P_{:j} \sqsupseteq T_{:i} \wedge P[j] = T[i] \iff P_{:j+1} \sqsupseteq T_{:i+1}$.

(4) If $P[j] = T[i]$, $i := i+1; j := j+1$.
Now, because of because of Lemma 4 (see **Example 5**),
$P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S =$ the set of valid shifts in $[0\,..\,i{-}j)$.
(5) If $j \neq m$, then $j < m$. Thus the invariant is true. Go to step (2).
(6) If $j = m$, then $P = P_{:m} \sqsupseteq T_{:i}$, and we have found a valid shift, i.e. $i{-}m$.
As a result, we perform $S := S \cup \{i{-}m\}$. Now
$P_{:j} \sqsupseteq T_{:i} \wedge m = j \leq i \leq n \wedge S =$ the set of valid shifts in $0\,..\,i{-}j$.
Go to step (8).
(7) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \wedge S =$ the set of valid shifts in $0\,..\,i{-}j$
$\wedge\, P[j] \neq T[i] \wedge i{-}j$ is not a valid shift.
If $j > 0$, go to step (8). Otherwise,
$0 = j \leq i < n \wedge S =$ the set of valid shifts in $0\,..\,i{-}j \wedge P[0] \neq T[i] \wedge i = i{-}j$ is not a valid shift. We perform $i := i+1$, and still
$0 = j < i \leq n \wedge S =$ the set of valid shifts in $[0\,..\,i{-}j)$.
This implies the invariant. Thus we go to step (2).
(8) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S =$ the set of valid shifts in $0\,..\,i{-}j$.
We make a *minimal further shift* of $P$ on $T$ so that the $P_{:k}$ $(0 \leq k < j)$ which is still against $T[i{-}k\,..\,i)$ matches it, i.e. $P_{:k} \sqsupset T_{:i}$ (see **Example 5**). This *minimal further shift* is $\leq j$, because $P_{:0} \sqsupset T_{:i}$. And in this way, we do not jump over any possibly valid shift. Then we can perform $j := k$ and the invariant above will be true after this shift. Thus we go to step (2).

**Example 5.** Let $P = BABABB$ and $j = 5$. The first three lines of the next table demonstrate **Lemma 4**, i.e. $P_{:j} \sqsupseteq T_{:i} \wedge P[j] = T[i] \iff P_{:j+1} \sqsupseteq T_{:i+1}$. The underscored letters of the pattern were matched against the appropriate letter of the text successfully ($P_{:5} \sqsupseteq T_{:i}$) and the last $B = P[j]$ of the pattern is crossed because it does not match $T[i]$. The fourth line illustrates the *minimal further shift* in step (8) of KMP. After this shift, $P_{:3} \sqsupset T_{:i}$ and we compare only $P[3\,..\,5]$ to $T[i\,..\,i{+}2]$. The bigger possible shifts would jump over valid shift $i{-}3$.

| ... | $T[i{-}5]$ | $T[i{-}4]$ | $T[i{-}3]$ | $T[i{-}2]$ | $T[i{-}1]$ | $T[i]$ | $T[i{+}1]$ | $T[i{+}2]$ |
|---|---|---|---|---|---|---|---|---|
| ... | $B$ | $A$ | $B$ | $A$ | $B$ | $A$ | $B$ | $B$ |
| $P =$ | $\underline{B}$ | $\underline{A}$ | $\underline{B}$ | $\underline{A}$ | $\underline{B}$ | $\cancel{B}$ | | |
| | | | $B$ | $A$ | $B$ | $\underline{A}$ | $\underline{B}$ | $\underline{B}$ |

The question remains, how to efficiently determine $k$ in step (8) of the algorithm above? Unquestionably a greater further shift corresponds to a smaller $k$ and a smaller further shift corresponds to a greater $k$. And $k$ corresponds to the *minimal further shift* of $P$ on $T$ so that $P_{:k} \sqsupset T_{:i}$. Thus $k$ is the greatest $h$ so that $P_{:h} \sqsupset T_{:i}$ and $0 \le h < j$. On the other hand, $P_{:h} \sqsupset T_{:i}$ is equivalent to $P_{:h} \sqsupset P_{:j}$ because $P_{:j} \sqsupseteq T_{:i}$ and $0 \le h < j$.

As a result, $k$ depends only on $P_{:j}$. Because $P$ is fixed, $k$ depends only on $j$. In other words, we need the longest $P_{:h}$ so that $P_{:h} \sqsubset P_{:j} \wedge P_{:h} \sqsupset P_{:j}$. The following definition gives its length with the prefix function $\pi$.

**Definition 6.** $H_j = \{\, h \in 0\,..\,j{-}1 \mid P_{:h} \sqsupset P_{:j} \,\} \quad (j \in 1\,..\,m)$.
$\pi(j) = \max H_j \quad (j \in 1\,..\,m)$

Undoubtedly the values of this prefix function $\pi$ can be collected into the $\pi/1 : \mathbb{N}[m]$ array as the initialization of the algorithm above. (We index the array $\pi$ from 1 to $m$ for convenience.[1]) Then the *minimal further shift* in step (8) of the algorithm above can be realized with the assignment $j := \pi[j]$.

It is known that the initialization of the $\pi$ array can be done with a simple *while program* which runs in $\Theta(m)$ time, although the proof of its correctness is nontrivial. We prove it in the paper with a new approach, although our main concepts and results will be familiar from [2].

# References


[1] R. S. Boyer, J. S. Moore: *A Fast String Searching Algorithm*, Communications of ACM 20.10 (1977), pp. 762–772, doi: https://doi.org/10.1145/359842.359859.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *String Matching*, in: Introduction to Algorithms (Fourth Edition), Cambridge, Massachusetts: The MIT Press, 2022, pp. 957–985.

[3] M. Crochemore, T. Lecroq: *Tight bounds on the complexity of the Apostolico-Giancarlo algorithm*, Information Processing Letters 63.4 (1997), pp. 195–203, doi: https://doi.org/10.1016/S0020-0190(97)00107-5.


[1] This notation may seem strange. But the author believes that indexing does not belong to the array object. It belongs to the access of the array.