Acta Cybernetica **27** (2025) 5–14.

Invariants and String Properties in the Analysis of the Knuth-Morris-Pratt Algorithm*

Tibor Ásványi[a]

Abstract

This paper is about the string-matching problem. We find the occurrences of a pattern inside a text. First, we formally define the problem and summarise its naive solution. Next, we analyse an efficient method, the Knuth-Morris-Pratt (KMP) algorithm.

We prove the correctness of the KMP algorithm. We also analyse its efficiency. Our reasoning is based on the properties of the pattern and the text. It is also based on the invariant properties of KMP. In this way, we could develop an extremely compact and elegant proof. And the method of proving program correctness with the invariant properties of the program is already familiar to our students at our university.

Keywords: string-matching, valid shift, prefix function, suffix, invariant

1 Notations and basic notions

$\mathbb{N} = \{\, i \in \mathbb{Z} \mid i \geq 0 \,\}$ $\qquad$ $i\,..\,k = [i\,..\,k] = \{\, j \in \mathbb{N} \mid i \leq j \leq k \,\}$
$[i\,..\,k) = \{\, j \in \mathbb{N} \mid i \leq j < k \,\}$ $\qquad$ $(i\,..\,k) = \{\, j \in \mathbb{N} \mid i < j < k \,\}$
$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ is the *alphabet* where $d \in \mathbb{N} \wedge d > 0$

$T : \Sigma[n]$ is the *text*, an array of letters. $T = T[0\,..\,n) = T[0\,..\,n{-}1]$.
Provided that $0 \leq i \leq l \leq n$, $T[i\,..\,l)$ is a *string;* $T[i\,..\,i) = \varepsilon$ is the *empty string.*
$T[j\,..\,k)$ is a *substring* of $T[i\,..\,l)$ if $i \leq j \leq k \leq l$.

$P : \Sigma[m]$ is the *pattern* $(0 < m \leq n)$. We search for the occurrences of P in T, i.e. we look for the substrings of T equal to P.
$P_{:j} = P[0\,..\,j) = P[0\,..\,j{-}1]$ and similarly for T. $\quad$ Thus, $P_{:0} = \varepsilon$.

$P[j\,..\,k) \sqsupseteq P[i\,..\,k)$ (string $P[j\,..\,k)$ is a *suffix* of string $P[i\,..\,k)$) if $i \leq j \leq k$.
Consequently, $P_{:0} \sqsupseteq P_{:j}$ because $P_{:0} = \varepsilon = P[j\,..\,j) \sqsupseteq P[0\,..\,j)$.

*Thanks to our faculty leaders for financial support, my colleagues for the encouragement and my students for the questions.

[a]Eötvös Loránd University, Faculty of Informatics, Pázmány Péter sétány 1/C, Budapest, 1117, Hungary, E-mail: asvanyi@inf.elte.hu, ORCID: 0000-0002-1715-9195

DOI: 10.14232/actacyb.308844

$P[j\,..\,k) \sqsupset P[i\,..\,k)$ (string $P[j\,..\,k)$ is a *proper suffix* of string $P[i\,..\,k)$) if $i < j \leq k$.
Thus, $P_{:0} \sqsupset P_{:j}$ if $j > 0$. (Proof: $P_{:0} = P[j\,..\,j) \sqsupset P[0\,..\,j)$ if $j > 0$.)

$P[i\,..\,j)$ is a *proper prefix* of $P[i\,..\,k)$) if $i \leq j < k$.
String x is a *proper prefix-suffix* (*PPS*) of y if x is a proper prefix and suffix of y.
Consequently, $P_{:i}$ is a *PPS* of $P_{:j}$ if $P_{:i} \sqsupset P_{:j}$ (because $P_{:i} \sqsupset P_{:j}$ implies $i < j$).

The following three lemmas on strings will be useful. (Most of them are illustrated in [2]. We present their proof here.) The first one tells us that a proper suffix of a string's suffix is a proper suffix of this string.

Lemma 1.1. *(Transitivity of the suffix relation)*
$x \sqsupset y \wedge y \sqsupseteq z \Rightarrow x \sqsupset z$.

Proof. We can suppose that $z = R[i\,..\,l)$ where $R[i\,..\,l)$ is a string. Thus, $y \sqsupseteq z$ means that $y = R[j\,..\,l)$ where $i \leq j \leq l$; and $x \sqsupset y$ means that $x = R[k\,..\,l)$ where $j < k \leq l$. Consequently, $i < k \leq l$. Therefore, $x = R[k\,..\,l) \sqsupset R[i\,..\,l) = z$. □

The second lemma contains three statements. Given two suffixes of a string, (1) one is a suffix of the other if it is not longer than the other, (2) one is a proper suffix of the other if it is shorter than the other, (3) they are equal if their lengths are equal.

Lemma 1.2. *(Overlapping-suffix lemma)*
Suppose that x, y and z are strings such that $x \sqsupseteq z$ and $y \sqsupseteq z$.
$|x| \leq |y| \Rightarrow x \sqsupseteq y$. $|x| < |y| \Rightarrow x \sqsupset y$. $|x| = |y| \Rightarrow x = y$.

Proof. We can suppose that $z = R[i\,..\,l)$. Thus, $x \sqsupseteq y$ means that $x = R[j\,..\,l)$ where $i \leq j \leq l$; and $y \sqsupseteq z$ means that $y = R[k\,..\,l)$ where $i \leq k \leq l$. Consequently, (1) $|x| \leq |y|$ means that $l - j \leq l - k$; therefore, $k \leq j \leq l$, i.e. $x = R[j\,..\,l) \sqsupseteq R[k\,..\,l) = y$. (2) $|x| < |y|$ means that $l - j < l - k$; therefore, $k < j \leq l$, i.e. $x = R[j\,..\,l) \sqsupset R[k\,..\,l) = y$. (3) $|x| = |y|$ means that $l - j = l - k$; therefore, $k = j \leq l$, i.e. $x = R[j\,..\,l) = R[k\,..\,l) = y$. □

The third lemma tells us that given two nonempty strings (the strings on the right side of the logical equivalences), one is a (proper) suffix of the other if and only if their last letters are the same and the first without its last letter is a (proper) suffix of the second without its last letter.

Lemma 1.3. *(Suffix-extension lemma)*
$P_{:j} \sqsupseteq T_{:i} \wedge P[j] = T[i] \iff P_{:j+1} \sqsupseteq T_{:i+1}$.
$P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j] \iff P_{:i+1} \sqsupset P_{:j+1}$.

Proof.
(1) $P_{:j+1} = P[0\,..\,j] \wedge T_{:i+1} = T[0\,..\,i]$. Thus,
$P_{:j+1} \sqsupseteq T_{:i+1} \iff P[0\,..\,j] = T[i-j\,..\,i] \iff$
$P[0\,..\,j) = T[i-j\,..\,i) \wedge P[j] = T[i] \iff P_{:j} \sqsupseteq T_{:i} \wedge P[j] = T[i]$.

(2) $P_{:i+1} = P[0\,..\,i] \wedge P_{:j+1} = P[0\,..\,j]$. Thus,
$P_{:i+1} \sqsupset P_{:j+1} \iff i < j \wedge P[0\,..\,i] = P[j-i\,..\,j] \iff$
$i < j \wedge P[0\,..\,i) = P[j-i\,..\,j) \wedge P[i] = P[j] \iff P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j]$. □

2 Introduction

In this paper, we suppose that $P : \Sigma[m]$, $T : \Sigma[n]$ and their lengths, i.e. m and n are fixed where $0 < m \leq n$. We search for those shifts s of P on T where $T[s\,..\,s{+}m) = P[0\,..\,m)$. Clearly, s must be in $0\,..\,n{-}m$.

Definition 2.1. *$s \in 0\,..\,n{-}m$ 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 2.1 (String-matching)**.** *Compute the set V of the valid shifts of P on T, i.e.*

$$V = \{\, s \in 0\,..\,n{-}m \mid T[s\,..\,s{+}m) = P[0\,..\,m) \,\}$$

The naive string-matching (Brute-Force) algorithm checks each possible shift in order and collects the valid shifts with maximal (i.e. worst-case) time complexity $\Theta((n{-}m{+}1){*}m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. [2] (See the details of this Θ-notation in the first chapter of [4].)

More advanced methods – like the different versions of the Boyer-Moore [1, 3], Rabin-Karp, and 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.

We prefer KMP because it runs in $\Theta(n)$ time on all the possible inputs, and it never backtracks on T, making it easy to implement on sequential files. KMP is traditionally introduced as a highly efficient simulation of *string matching with finite automata* [2]. Here, we avoid these automata and start with analysing P and T, i.e. the strings. To introduce KMP, let us see Example 2.1.

Example 2.1. In this example, we suppose there is a longer text T, but we consider only $T[i{-}5\,..\,i{+}2] = BABABABB$ here. The pattern is $P = P_{:6} = BABABB$. The actual shift is $i - 5$. The successfully matched characters are underlined. The unsuccessfully matched character is crossed out.

$\ldots$	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
$\ldots$	B	A	B	A	B	A	B	B
$P =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\cancel{B}$		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$

In the third line of the table, we successfully matched $P_{:5}$ to $T[i{-}5\,..\,i)$ but $P[5] \neq T[i]$. Consequently, $i{-}5$ is not a valid shift.

Thus we make a *minimal additional shift* of P on T so that the $P_{:k}$ $(0 \leq k < 5)$ which is still against $T[i{-}k\,..\,i)$ matches it, i.e. $P_{:k} \sqsupset T_{:i}$. (See the last line of the table.) *This shift* is ≤ 5, because $P_{:0} \sqsupset T_{:i}$. And with *this shift*, we do not jump over any possibly valid shift. Actually $k = 3$. Then we successfully match $P[3]$ to $T[i]$, $P[4]$ to $T[i{+}1]$ and $P[5]$ to $T[i{+}2]$. Thus, $i{-}3$ is a valid shift. The bigger possible shifts would jump over the valid shift $i{-}3$.

Understanding the previous example, the question remains: How do we efficiently determine the value of k above? In the previous example, $j = 5$, but the following argument can be applied to any $j \in 1\,..\,m$ where $i - j$ is the actual shift, $P_{:j} \sqsupseteq T_{:i}$ and $(P[j] \neq T[i] \vee j = m)$.

Unquestionably, a greater additional shift corresponds to a smaller k, and a smaller additional shift corresponds to a greater k. And k corresponds to the *minimal additional 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 $0 \leq h < j$. Moreover, $P_{:h} \sqsupset T_{:i}$ is equivalent to $P_{:h} \sqsupset P_{:j}$ because $P_{:j} \sqsupseteq T_{:i}$ and $0 \leq h < j$. Consequently, k is the greatest h so that $P_{:h} \sqsupset P_{:j}$.

As a result, k depends only on $P_{:j}$. After all, we need the longest *PPS* of $P_{:j}$. Its length is defined by the prefix function π. Because P is fixed, this length depends only on j.

Definition 2.2. $\pi(j) = \max\{\, h \in 0\,..\,j-1 \mid P_{:h} \sqsupset P_{:j} \,\} \qquad (j \in 1\,..\,m)$

An efficient calculation of this function is given in Section 4, but before it, in Section 3, we analyse the main procedure of the KMP algorithm.

3 The KMP algorithm

In this section, first, we transform the intuitive approach of the KMP algorithm of the Introduction into **Algorithm 1**. Next, we analyse it.

Algorithm 1 the Knuth-Morris-Pratt algorithm

```
procedure KMP(T : Σ[n] ; P : Σ[m] ; S : ℕ{})
 1: π : ℕ[1..m] ; INIT(π, P) ; S := {} ; i := j := 0
 2: while i < n do
 3:    if P[j] = T[i] then
 4:       i++ ; j++
 5:       if j = m then
 6:          S := S ∪ {i − m} ; j := π[j]
 7:       end if
 8:    else if j = 0 then
 9:       i++
10:    else
11:       j := π[j]
12:    end if
13: end while
```

We will prove in Section 4 that INIT(π, P) collects the values of the π prefix function into the $\pi[1\,..\,m]$ prefix array in $\Theta(m)$ time. And the postcondition of the INIT(π, P) call is $\forall j \in 1\,..\,m : \pi[j] = \pi(j)$. (See **Algorithm 2**.) Our analysis of the KMP algorithm is based on the invariant (Inv) of Theorem 3.1 where $i - j$ is the actual shift. (See Section 2.4 of [4] on an exact program correctness proof with loop invariant.)

3.1 The partial correctness of the procedure KMP(T, P, S)

The following lemma will be appropriate where $i-j$ is the actual shift. (It tells us the following. When we search for a valid shift of P on T, $i-j$ is P's actual shift and $\varepsilon \neq P[0\,..\,j) = T[i-j\,..\,i)$, then the shift values between $i-j$ and $i-\pi(j)$ are invalid: We find no solution there; and this does not depend on the validity of the actual shift, i.e. $i-j$.)

Lemma 3.1. $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i} \Rightarrow$ *there is no valid shift in* $(i-j\,..\,i-\pi(j))$.

Proof. Assume indirectly that $k \in (\pi(j)\,..\,j)$ and $i-k$ is a valid shift. This means $T[i-k\,..\,i-k+m) = P[0\,..\,m)$. Clearly, $k < j \leq m$, therefore $k < m$. Thus $T[i-k\,..\,i) = P[0\,..\,k)$, i.e. $P_{:k} \sqsupseteq T_{:i}$. And $P_{:j} \sqsupseteq T_{:i} \wedge k < j$. As a result, $P_{:k} \sqsupset P_{:j}$ because of the Overlapping-suffix lemma (1.2). But $k > \pi(j)$. For this reason, $P_{:k} \not\sqsupset P_{:j}$ follows from Definition 2.2 of the π function. □

The following theorem is the key to the KMP algorithm. It formulates an invariant property of its main loop. Again, $i-j$ is P's actual shift and $P[0\,..\,j) = T[i-j\,..\,i)$, but $P[0\,..\,j)$ is not the the whole pattern. Thus, we do not know whether $i-j$ is a valid shift, but we know that the S set already contains all the valid shifts before the actual shift.

Theorem 3.1.
Statement (Inv) *is an invariant of the loop of the procedure KMP*(T, P, S).
(Inv) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \;\wedge S = V \cap [0\,..\,i-j)$.

Proof. Immediately before the first loop iteration, we perform the $S := \{\}$; $i := j := 0$ initialisations. Thus, (Inv) holds because $i = j = 0 \wedge P_{:0} \sqsupseteq T_{:0} \wedge 0 \leq 0 \leq 0 \leq n \wedge 0 < m \wedge S = \{\} = V \cap [0\,..\,0)$.

We prove that each iteration of the loop keeps (Inv). The postcondition of the init(π, P) call, i.e. $(\forall j \in 1\,..\,m : \pi[j] = \pi(j))$ is implicitly added to each statement.

Supposing that $i < n$, we enter the loop and
(Inv1) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \;\wedge S = V \cap [0\,..\,i-j)$ $\quad$ stands.

- If $P[j] = T[i]$, then
  $P_{:j+1} \sqsupseteq T_{:i+1}$ according to the Suffix-extension lemma (1.3). After increasing i and j we have
  (Inv2) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \;\wedge S = V \cap [0\,..\,i-j)$.

  1. If $j = m$, then $P_{:m} \sqsupseteq T_{:i}$, i.e. $P[0\,..\,m) = T[i-m\,..\,i)$. This means $i-m$ is a valid shift. Thus, we add it to S. Then, we have the following statement.
     (Inv3) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \;\wedge S = V \cap [0\,..\,i-j]$.
     Because $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i}$, considering Lemma 3.1, we receive that there is no valid shift in the interval $(i-j\,..\,i-\pi(j))$. Thus
     $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \;\wedge S = V \cap [0\,..\,i-\pi(j))$.
     Consider that $P_{:\pi(j)} \sqsupset P_{:j} \sqsupseteq T_{:i}$. Based on the transitivity of the suffix

relation (Lemma 1.1) and $\pi[j] = \pi(j)$:
$P_{:\pi[j]} \sqsupset T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}\pi[j])$.
Because $\pi[j] = \pi(j) \in [0\,..\,j)$, after the $j := \pi[j]$ assignment already $0 \leq j < i \wedge j < m$ stands. Consequently,
$P_{:j} \sqsupset T_{:i} \wedge 0 \leq j < i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$ holds. At the end of the first program branch, this directly implies (Inv).

2. Provided that $j \neq m$, (Inv2) implies $j < m$. Thus, (Inv) holds at the end of the second program branch.

- In case of $P[j] \neq T[i]$, the Suffix-extension lemma (1.3) implies $P_{:j+1} \not\sqsupseteq T_{:i+1}$, i.e. $P[0\,..\,j] \neq T[i-j\,..\,i]$. Based on (Inv1), $j < m$. Thus, $P[0\,..\,m) \neq T[i-j\,..\,i-j+m)$ if $i-j+m < n$. Consequently, $i{-}j$ is not a valid shift. Comparing this to (Inv1), i.e.
$P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$, we have
(Inv4) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j]$.

3. If $j = 0$, then considering (Inv4) we receive
$P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j]$.
After performing $i++$,
$P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$ stands.
Therefore, (Inv) holds at the end of the third program branch:
(Inv) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$.
4. Provided that $j \neq 0$, then taking (Inv4) into account, we obtain
$P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j]$
This has the direct consequence
(Inv3) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}j]$.
We have already seen in the examination of the first program branch that in the case of (Inv3), after performing assignment $j := \pi[j]$, the invariant (Inv) holds. Finally, (Inv) also stands at the end of the last program branch.

□

Theorem 3.2. *If the KMP algorithm terminates, it solves Problem 2.1 of string-matching, i.e. $S = V$ holds when it returns.*

Proof. Let us consider Theorem 3.1. The (Inv) invariant of KMP's loop with the loop's termination condition, i.e. $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \wedge S = V \cap [0\,..\,i{-}j)$ with $i \geq n$ implies that $i = n \wedge j < m \wedge S = V \cap [0\,..\,n{-}j)$ holds when the loop of KMP becomes completed. Furthermore, $j < m \Rightarrow [0\,..\,n{-}j) \supset [0\,..\,n{-}m) \supseteq \{\, s \in 0\,..\,n{-}m \mid T[s\,..\,s{+}m) = P[0\,..\,m) \,\} = V \Rightarrow [0\,..\,n{-}j) \supset V$. Thus $S = V \cap [0\,..\,n{-}j) = V$. Consequently, $S = V$ holds when the procedure KMP returns. □

3.2 The termination of the procedure KMP(T, P, S)

First, we prove that the loop iterates at least n times. Before the first iteration, $i = 0$. Each iteration increases i by 1 or 0. And the loop terminates with $i = n$

according to the $i < n$ condition and the $0 \leq i \leq n$ invariant. Thus, there are at least n iterations before the loop terminates.

Second, we prove that the loop iterates at most $2n$ times. Let the termination function be $2i - j$ where $0 \leq j \leq i \leq n$ [see the (Inv) invariant in Theorem 3.1]. Thus $2i-j \in 0\,..\,2n$. Before the loop, $2i-j = 0$, and each iteration increases $2i-j$. Consequently, there are at most $2n$ iterations before the loop terminates.

The loop of KMP runs in $\Theta(n)$ time because n is at least the number of the iterations of the KMP loop, which is at most $2n$.

Remember that we will prove in Section 4 that the INIT(π, P) call terminates in $\Theta(m)$ time.

As a result, the time complexity of the KMP(T, P, S) procedure is $\Theta(n) + \Theta(m) = \Theta(n)$ because $n \geq m \geq 0$.

4 Initializing the prefix array

In this section, we will compute the values of the $\pi : [1\,..\,m] \to [0\,..\,m)$ prefix function (see Definition 2.2) and store them in the $\pi[1\,..\,m]$ array. Remember that $\pi(j)$ is the length of the longest PPS of $P_{:j}$. Thus, $\pi(1) = 0$, and we can perform $\pi[1] := 0$. Subsequently, provided that we have filled $\pi[1\,..\,j]$ where $1 \leq j < m$, we want to calculate $\pi(j+1)$, store it in $\pi[j+1]$ and so on.

$\pi(j+1)$ is the length of the longest PPS of $P_{:j+1}$. If this PPS is nonempty, let us denote it with $P_{:k+1}$. Thus $0 \leq k < j$. According to the 1.3 Suffix-extension lemma, $P_{:i+1} \sqsupset P_{:j+1} \iff P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j]$. This means that $P_{:k}$ is the longest $P_{:i} \sqsupset P_{:j}$ where $P[i] = P[j]$. To determine $P_{:k}$ (and hence $P_{:k+1}$), we check the $P_{:i}$ PPSs of $P_{:j}$ in decreasing order according to i and find the first one where $P[i] = P[j]$. If we do not find such a $P_{:i}$, then $\pi(j+1) = 0$.

The question is, given a $P_{:i} \sqsupset P_{:j}$ where $i > 0$ and $P[i] \neq P[j]$, how to determine the next longest PPS of $P_{:j}$. Let it be $P_{:l}$. As a result, $P_{:i} \sqsupset P_{:j} \wedge P_{:l} \sqsupset P_{:j} \wedge l < i$. Thus, $P_{:l} \sqsupset P_{:i}$ (see Lemma 1.2) and it is the longest one. Consequently, $l = \pi(i)$. $\pi(i) = \pi[i]$ because $i < j$ and $\pi[1\,..\,j]$ is already calculated. As a result, we can apply $i := \pi[i]$ and have the following intuitive loop invariant:

$\pi[1\,..\,j]$ have been calculated where $1 \leq j \leq m$, and if $j < m$, then $P_{:i}$ is the longest PPS of $P_{:j}$ for which still there is a chance that $P[i] = P[j]$.

Based on this invariant, we can write the INIT(π, P) procedure, i.e. Algorithm 2.

The following lemmas will be appropriate to prove the correctness of Algorithm 2. The first one says that the π-values can only increase at most one by one (when they grow). When $\pi(j)$ has been calculated, it provides an upper limit for $\pi(j+1)$.

Lemma 4.1. $j \in [1\,..\,m) \Rightarrow \pi(j+1) \leq \pi(j) + 1$

Proof. If $\pi(j+1) = 0 \Rightarrow \pi(j+1) = 0 \leq 0 + 1 \leq \pi(j) + 1$ because $\pi(j) \geq 0$ by definition. If $\pi(j+1) > 0 \Rightarrow$ with Definition 2.2, $P_{:(\pi(j+1)-1)+1} = P_{:\pi(j+1)} \sqsupset P_{:j+1} \Rightarrow$ with Lemma 1.3, $P_{:\pi(j+1)-1} \sqsupset P_{:j} \Rightarrow$ again with Definition 2.2, $\pi(j+1) - 1 \leq \pi(j) \Rightarrow \pi(j+1) \leq \pi(j) + 1$. □

Algorithm 2 Knuth-Morris-Pratt initialization

procedure INIT($\pi : \mathbb{N}[1\,..\,m]$; $P : \Sigma[m]$)
1: $\pi[1] := i := 0$; $j := 1$
2: **while** $j < m$ **do**
3: **if** $P[i] = P[j]$ **then**
4: $i++$; $j++$; $\pi[j] := i$
5: **else if** $i = 0$ **then**
6: $j++$; $\pi[j] := 0$
7: **else**
8: $i := \pi[i]$
9: **end if**
10: **end while**

The next lemma also helps us when we are going to calculate $\pi(j+1)$, i.e. the length of the longest PPS of $P_{:j+1}$: In this situation, because of the following invariants of the INIT() procedure, the conditions of the lemma are almost satisfied. If $P[i] \neq P[j]$, then $\pi(j+1) \leq i$ will be found. In this case, if $i = 0$, then $\pi(j+1) = 0$ follows; and if $i > 0$, then the lemma reduces the upper bound of $\pi(j+1)$.

Lemma 4.2. $P_{:i} \sqsupset P_{:j} \wedge 0 < i \wedge j < m \wedge \pi(j+1) \leq i \Rightarrow \pi(j+1) \leq \pi(i) + 1$

Proof. If $\pi(j+1) = 0$ then $\pi(j+1) < 0 + 1 \leq \pi(i) + 1$ because $\pi(i) \geq 0$ by definition. Provided that $\pi(j+1) > 0$, $k := \pi(j+1) - 1$. Thus $i > k \geq 0$ and $k + 1 = \pi(j+1)$. By the definition of the π function, $P_{:k+1} \sqsupset P_{:j+1}$. Consequently $P_{:k} \sqsupset P_{:j}$. Considering $P_{:i} \sqsupset P_{:j}$ and $k < i$, we have $P_{:k} \sqsupset P_{:i}$, thus $k \leq \pi(i)$. Therefore $\pi(j+1) = k + 1 \leq \pi(i) + 1$. □

4.1 The partial correctness of the procedure INIT(π, P)

The following invariant of the INIT(π, P) procedure's loop is the formalised version of the intuitive loop invariant above. It is the key to the partial correctness of this subroutine, which means that $\forall k \in 1\,..\,m : \pi[k] = \pi(k)$ stands when it returns.

Theorem 4.1.
Statement (inv) *is an invariant of the loop of the procedure* INIT(π, P).
(inv) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \leq i < j \leq m \wedge (\,j < m \rightarrow \pi(j+1) \leq i + 1\,)$.

Proof. Because of the initialisations $\pi[1] := i := 0$; $j := 1$, immediately before the first iteration of the loop, (inv) corresponds to the following statement: $P_{:0} \sqsupset P_{:1} \wedge (\,\forall k \in 1\,..\,1 : \pi[k] = \pi(k) = \pi(1) = 0\,) \wedge 0 \leq 0 < 1 \leq m \wedge (\,1 < m \rightarrow \pi(2) \leq 1\,)$. To prove the elements of this formula, the $P_{:0}$ empty string is a proper suffix of any nonempty string; according to Definition 2.2, $\pi(1) = 0$; the size m of the pattern P is not zero; and finally, because of Lemma 4.1, $\pi(1+1) \leq \pi(1) + 1 = 1$, provided that $m > 1$.

Still, we have to prove that the iterations of the loop keep the (inv) invariant, i.e. provided that (inv) holds before an iteration of the loop, it will also stand at the end of any branch of the loop's body. When we enter into the body of the loop, (inv) and the loop's condition ($j < m$) implies

(inv1) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \leq i < j < m \wedge \pi(j+1) \leq i+1.$

1. If $P[i] = P[j]$, according to Lemma 1.3 we have $P_{:i+1} \sqsupset P_{:j+1}$ because $P_{:i} \sqsupset P_{:j}$ [see (inv1)]. Based on the definition of the π prefix function (2.2), $P_{:i+1} \sqsupset P_{:j+1}$ implies $\pi(j+1) \geq i+1$. But $\pi(j+1) \leq i+1$ is found in (inv1). Consequently, $\pi(j+1) = i+1$.
Performing the assignments $i++; j++; \pi[j] := i$,

   $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge 0 < i < j \leq m \wedge \pi(j) = i.$

   Provided that $j < m$, $\pi(j) = i$ and Lemma 4.1 implies $\pi(j+1) \leq \pi(j)+1 = i+1$. Therefore, at the end of the first branch of the loop's body, (inv) holds.

2-3. If $P[i] \neq P[j]$, then $P_{:i+1} \not\sqsupset P_{:j+1}$ because of Lemma 1.3. Thus, $\pi(j+1) \neq i+1$, according to Definition 2.2. In (inv1) we have $\pi(j+1) \leq i+1$. Consequently, $\pi(j+1) \leq i$.

   Comparing this to (inv1), we receive that (inv2) stands in line 5 before the if-statement:

   (inv2) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
   $0 \leq i < j < m \wedge \pi(j+1) \leq i.$

   2. Provided that $i = 0$, consider $\pi(j+1) \leq i$ from (inv2).
   We have $\pi(j+1) = 0$ because the π function is non-negative.

      Comparing this to (inv2), after the assignments $j++;\ \pi[j] := 0,$ we receive

      $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge 0 = i < j \leq m \wedge \pi(j) = i.$

      If $j < m$, Lemma 4.1 and $\pi(j) = i$ implies $\pi(j+1) \leq \pi(j)+1 = i+1$. Therefore, at the end of the second branch of the loop's body, (inv) also holds.

   3. Provided that $i \neq 0$, then (inv2) implies (inv3):

      (inv3) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
      $0 < i < j < m \wedge \pi(j+1) \leq i.$

      Thus, we can apply Lemma 4.2, and we have $\pi(j+1) \leq \pi(i)+1$.

      On the other hand, from (inv3) we receive $\pi[i] = \pi(i)$. Comparing this to Definition 2.2, we receive $P_{:\pi[i]} \sqsupset P_{:i}$. In (inv3), $P_{:i} \sqsupset P_{:j}$ is found. With Lemma 1.1, $P_{:\pi[i]} \sqsupset P_{:j}$ follows.

      Consider (inv3) and $\pi(j+1) \leq \pi(i)+1$. After the assignment $i := \pi[i]$, we receive

$P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \leq i < j < m \wedge \pi(j+1) \leq i+1.$

Therefore, at the end of the last branch of the loop's body, (inv) also holds. □

Corollary 4.1. *The loop invariant* (inv) *and the negation of the loop's condition, i.e.* $j \geq m$*, implies* $j = m$*. As a result, procedure* INIT$(\pi : \mathbb{N}[1\,..\,m]\ ;\ P : \Sigma[m])$ *has the post-condition* $\forall k \in 1\,..\,m : \pi[k] = \pi(k)$.

4.2 The termination of the procedure INIT(π, P)

First, we prove that the loop iterates at least $m-1$ times. Before the first iteration, $j = 1$. Each iteration increases j by 1 or 0. And the loop terminates with $j = m$ according to the $j < m$ condition and the $j \leq m$ invariant. Thus, there are at least $m-1$ iterations before the loop terminates.

Second, we prove that the loop iterates at most $2m-2$ times. Let the termination function be $2j - i$ where $0 \leq i < j \leq m$. Thus $2j - i \in 2\,..\,2m$. Before the loop, $2j - i = 2$, and each iteration increases $2j - i$. Consequently, the loop iterates not more than $2m-2$ times.

As a result, the time complexity of the INIT(π, P) procedure is $\Theta(m)$.

5 Summary

In this paper, we found relatively simple and short proof of the correctness and efficiency of the KMP algorithm. (Compare it to 32.3-4 in [2].) It is based on
(1) the properties of strings,
(2) the appropriate invariant properties of the loops of the algorithm and
(3) the suitable termination functions of these loops.

References

[1] Boyer, R. S. and Moore, J. S. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977. DOI: 10.1145/359842.359859.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *String Matching.* In *Introduction to Algorithms (4th Edition)*, pages 957–985. The MIT Press, 2022. ISBN: 978-0-262-04630-5.

[3] Crochemore, M. and Lecroq, T. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Information Processing Letters*, 63(4):195–203, 1997. DOI: 10.1016/S0020-0190(97)00107-5.

[4] Stinson, D. R. *Techniques for Designing and Analyzing Algorithms (1st Edition).* CRC Press (Taylor & Francis Group), Boca Raton, London, New York, 2022. DOI: 10.1201/9780429277412.