Heikki Hyyrö

On Boyer-Moore Preprocessing

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2004-1

TAMPERE 2004

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2004-1, NOVEMBER 2004

Heikki Hyyrö

On Boyer-Moore Preprocessing

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 951-44-6181-9
ISSN 1795-4274

On Boyer-Moore Preprocessing

Heikki Hyyrö
Department of Computer Sciences
University of Tampere, Finland
Heikki.Hyyro@cs.uta.fi

Abstract

Probably the two best-known exact string matching algorithms are the linear-time algorithm of Knuth, Morris and Pratt (KMP), and the fast on average algorithm of Boyer and Moore (BM). The efficiency of these algorithms is based on using a suitable failure function. When a mismatch occurs in the currently inspected text position, the purpose of a failure function is to tell how many positions the pattern can be shifted forwards in the text without skipping over any occurrences. The BM algorithm uses two failure functions: one is based on a *bad character* rule, and the other on a *good suffix* rule. The classic linear-time preprocessing algorithm for the good suffix rule has been viewed as somewhat obscure [8]. A formal proof of the correctness of that algorithm was given recently by Stomp [14]. That proof is based on linear time temporal logic, and is fairly technical and *a-posteriori* in nature. In this paper we present a *constructive* and somewhat simpler discussion about the correctness of the classic preprocessing algorithm for the good suffix rule. We also highlight the close relationship between this preprocessing algorithm and the exact string matching algorithm of Morris and Pratt (a pre-version of KMP). For these reasons we believe that the present paper gives a better understanding of the ideas behind the preprocessing algorithm than the proof by Stomp. This paper is based on [9], and thus the discussion is originally roughly as old as the proof by Stomp.

1 Introduction

The need to search for occurrences of some string within some other string arises in countless applications. Exact string matching is a fundamental task in computer science that has been studied extensively. Given a *pattern* string P and a typically much longer *text* string T, the task of exact string matching is to find all locations in T where P occurs. Let $|s|$ denote the length of a string s. Also let the notation x_i refer to the ith character of a string x, counting from the left, and let the notation $x_{h..i}$ denote the *substring* of x that is formed by the characters of x from its hth position to the ith position. Here we require that $h \leq i$. If $i > |x|$ or $i < 1$, we interpret the character x_i to be a non-existing

character that does not match with any character. A string y is a *prefix* of x if $y = x_{1..h}$ for some $h > 0$. In similar fashion, y is a *suffix* of x if $y = x_{h..|x|}$ for some $h \leq |x|$. It is common to denote the length of the pattern string P by m and the length of the text T by n. With this notation $P = P_{1..m}$ and $T = T_{1..n}$, and the task of exact string matching can be defined more formally as searching for such indices j for which $T_{j-m+1..j} = P$.

A naive "Brute-Force" approach for exact string matching is to check each possible text location separately for an occurrence of the pattern. This can be done for example by sliding a *window* of length m over the text. Let us say that the window is in position w when it overlaps the text substring $T_{w-m+1..w}$. The position w is checked for a match of P by a sequential comparison between the characters P_i and T_{w-m+i} in the order $i = 1 \ldots m$. The comparison is stopped as soon as $P_i \neq T_{w-m+i}$ or all m character-pairs have matched (in which case $T_{w-m+1..w} = P$). After the window position w has been checked, the window is shifted one step right to the position $w+1$, and a new comparison is started. As there are $n - m + 1$ possible window positions, and checking each location may involve up to m character comparisons, the worst-case run time of the naive method is $O(mn)$.

Morris and Pratt have presented a linear $O(n)$ algorithm for exact string matching [11]. Let us call this algorithm MP. It improves the above-described naive approach by using a suitable *failure function*, which utilizes the information gained from previous character comparisons. The failure function enables to move the window forward in a smart way after the window position w has been checked. Later Knuth, Morris and Pratt presented an $O(n)$ algorithm [10] that uses a slightly improved version of the failure function.

Boyer and Moore have presented an algorithm that is fast in practice, but $O(mn)$ in the worst case. Let us call it BM. Subsequently also many variants of BM have been proposed (e.g. [7, 2, 6]). The main innovation in BM is to check the window in reverse order. That is, when the window is at position w, the characters P_i and T_{w-m+i} are compared from right to left in the order $i = m \ldots 1$. This enables to use a failure function that can often skip over several text characters. BM actually uses two different failure functions, δ_1 and δ_2. The former is based on so-called *bad character* rule, and the latter on so-called *good suffix* rule.

The failure function δ_1 of BM is very simple to precompute. But the original preprocessing algorithm given in [4] for the δ_2 function has been viewed as somewhat mysterious and incomprehensible [8]. Stomp even states that the algorithm is known to be "notoriously difficult" [14]. An example of this is that the algorithms shown in [4, 10] were slightly erroneous, and a corrected version was given without any detailed explanations by Rytter [12]. A formal proof of the correctness of the preprocessing algorithm was given recently by Stomp [14]. He analysed the particular version shown in [3, 1], and also found and corrected a small error that concerns running out of bounds of an array. Stomp's proof is based on linear temporal logic and it is *a-posteriori* in nature: he first shows the algorithm, and then proceeds to prove that that given algorithm computes δ_2 correctly. The proof is also fairly technical, and does not shed too much light

on the intuitive foundations of the algorithm.

In this paper we present a constructive analysis about the original preprocessing algorithm for δ_2. In doing this our goal is to expose the ideas behind the algorithm, and subsequently make it easier to understand. Our discussion highlights the tight connection between the preprocessing algorithms for δ_2 and the failure function of the exact string matching algorithm of Morris and Pratt. The latter is much more simple to understand, and we show how the original preprocessing algorithm for δ_2 can be derived from it in a fairly straightforward manner. The present work is a modified conversion of a part in [9]. It is thus originally roughly as old as the proof by Stomp.

2 Morris-Pratt

The $O(n)$ exact string matching algorithm of Morris and Pratt [11], MP, is based on using information about the *borders* of a string. A string y is a border of the string x, if y is both a prefix and a suffix of x and $|y| < |x|$. The last condition means that x is not a border of itself even though it is both a prefix and a suffix of itself. Clearly each border of x can be uniquelly identified by the ending position of the corresponding prefix of x. That is, the border of x with an index i is the border $y = x_{1..i} = x_{|x|-|y|+1..|x|}$. An index 0 means that no nonempty border exists. It is clear that such border-indices form a finite and ordered set (we assume that the strings are finite). Let us define $B_s(i)$ as a set of the border-indices of the string $s_{1..i}$.

Definition 2.1 $B_s(i) = \{k \mid (k = 0) \vee ((0 < k < i) \wedge (s_{1..k} = s_{i-k+1..i}))\}$.

It is clear that the conditions $0 \in B_s(i)$ and $\min(k \mid k \in B_s(i)) = 0$ hold for all nonempty strings $s_{1..i}$. Under the convention that the first character of a string x is x_1, the index and the length of the border are the same. Thus for example in [5] the borders are identified by their lengths.

Let us also define a separate function $lb_s(i)$ that gives the index of the longest border of the string $s_{1..i}$. It is clear that this corresponds to the maximum border index of $s_{1..i}$.

Definition 2.2 $lb_s(i) = \max(k \mid k \in B_s(i))$.

It is worth noting that the function $lb_s(i)$ can be used as a *successor function* to enumerate the elements in the set $B_s(i)$ in descending order. Let $b_s(i, k)$ denote the kth-largest number in the set $B_s(i)$ and let $|B_s(i)|$ denote the cardinality of $B_s(i)$. For example $b_s(i, 1) = lb_s(i)$ and $b_s(i, |B(s)|) = 0$, and $lb_s(i)$ induces an ordered version of $B_s(i) = (b_s(i, k) \mid k \in [1..|B_s(i)|])$, where $b_s(i, k) < b_s(i, k-1)$ if $k > 0$. Also let $lb_s^k(i)$ denote applying the function $lb_s(i)$ k times. For example $lb_s^2(i) = lb_s(lb_s(i))$ and $lb_s^0(i) = i$.

Lemma 2.3 *If* $b_s(i, k) > 0$, *then* $b_s(i, k + 1) = lb_s(b_s(i, k))$.

Proof: Let us denote the value of $b_s(i,k)$ by p, the value of $lb_s(b_s(i,k)) = lb_s(p)$ by q, and the value $b_s(i,k+1)$ by r. Then $s_{1..p} = s_{i-p+1..i}$, and it is also known that $i > p$ and $p > q \geq 0$. If $q = 0$, then $q \in B_s(i)$. The other possibility is that $p > q > 0$, in which case $s_{1..q} = s_{p-q+1..p}$. But from the equality $s_{1..p} = s_{i-p+1..i}$ it also follows that $s_{1..q} = s_{i-q+1..i}$, and again $q \in B_s(i)$. Thus $q \in B_s(i)$, and since $p = b_s(i,k) > q$, we have that $q \leq r = b_s(i,k+1)$.

Let us now use contraposition by assuming that $q < r$. Then $q < r < p$ and $s_{1..r} = s_{i-r+1..i}$, and since $s_{1..p} = s_{i-p+1..i}$, we also have that $s_{1..r} = s_{p-r+1..p}$. This means that $r \in B_s(p)$. This contradicts the assumption $q < r$ because $q = lb_s(p)$ is the maximal element in $B_s(p)$.

From the previous we may conclude that $q = r$, that is, $lb_s(b_s(i,k)) = b_s(i,k+1)$. □

Fig. 1a shows an example of borders. Since $lb_s(i)$ is the maximal element in $B_s(i)$, the following is a direct consequence of Lemma 2.3.

Corollary 2.4 *If $1 \leq k \leq |B_s(i)|$, then $b_s(i,k)$, the kth-largest element in $B_s(i)$, is equal to $lb_s^k(i)$.*

Consider now a situation where a length-m window is at position w in the text (i.e., it overlaps the characters $T_{w-m+1..w}$), and the character-comparisons in the current position found that $P_{1..i-1} = T_{j-i+1..j-1}$ and $P_i \neq T_j$, where $j = w - m + i$. If an occurrence of the pattern begins at position $j - r$ in the text so that $1 \leq r < i - 1$ and $T_{j-r..j-r+m-1} = P_{1..m}$, then $T_{j-r..j-1} = P_{i-r..i-1} = P_{1..r}$. Thus any such r must belong to the set $B_P(i-1)$. Moreover, the next such possible position in the text corresponds to the *maximal* such r, that is, $lb_P(i-1)$. The algorithm of Morris and Pratt is based on this information. In the depicted situation, MP shifts the length-m window forward in the text according to the value $r = lb_P(i-1)$. If $r > 0$, the matching borders $P_{1..r}$ and $T_{j-r..j-1}$ are aligned with each other. This corresponds to shifting the window forward by $i - r - 1$ positions. And since the equality $P_{1..r}$ and $T_{j-r..j-1}$ is already known, the character-comparisons in the new window-position can proceed by comparing $P_{r+1..}$ against $T_{j..}$. Fig. 1b illustrates. If it happens that $r = lb_P(i-1) = 0$, then no pattern occurrence can begin inside the text-segment $T_{w-m+2..w-m+i} = T_{j-i+2..j-1}$. In this case the window is shifted to the position $w + i - 1 = j + m - 1$, so that T_j is the leftmost character it overlaps, and the character-comparisons start from the beginning of the pattern and the window.

Let us denote the failure function of MP as $f_P(i)$. When the character P_i is involved in a mismatch with T_j, the value $f_P(i)$ gives the index of the pattern character that will be compared next. From the preceding discussion it is seen that if $r = lb_P(i-1) > 0$, the next pattern character to be compared is P_{r+1}, and the next compared text character is the same as before, T_j. There are two exceptions.

If the mismatch happens already at the first pattern character P_1, the window should be moved one step forward and the character comparisons should start again from the character P_1. In this case also the next compared text

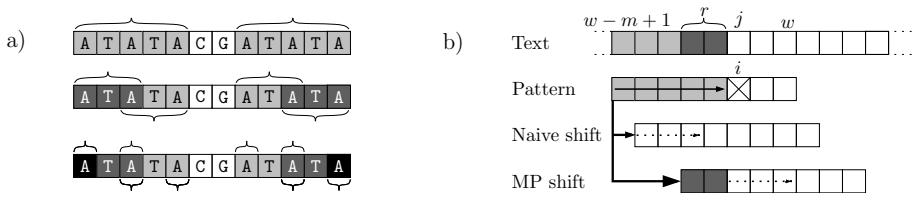

Figure 1: Figure a) shows the different borders within the string $s = s_{1..12}$ = "ATATACGATATA". The first row shows the longest borders of s, which are $s_{1..5} = s_{8..12}$ = "ATATA" (light gray shading) and correspond to the value $lb_s(12) = 5$. The second row shows the second-longest borders of s, which correspond to $lb_s^2(12) = 3$ and are $s_{1..3} = s_{10..12}$ = "ATA" (dark gray). In similar fashion, the third row shows the third-longest borders $lb_s^3(12) = 1$ and are $s_1 = s_12$ = "A" (black). To illustrate how the kth-longest border of $s_{1..12}$ is $lb_s^k(12)$, each border occurrence inside $s_{1..12}$ is also shown with curly braces. For example the third-longest border $lb_s^3(12)$ = "A" is also a border of $s_{1..lb_s(12)} = s_{1..5}$ = "ATATA" and $s_{1..lb_s^2(12)} = s_{1..3}$ = "ATA".
Figure b) compares the shifts made by the naive and the MP method.

character changes one position forward: it will now be T_{j+1}. This situation will be signaled with a special value $f_P(1) = 0$. In addition to this, the only other situation when the comparison point in the text is incremented is after succesfully matching a character-pair between the text and the pattern.

The second exception is how to shift the windows after a pattern occurrence has been found. It is clear that we can use the value $lb_P(m)$ in order to shift the window to align the next possible matching borders, if such exist. As we assume that the non-existing character P_{m+1} does not match with any character, we do not need to handle this case separately (although in practice it might be a good idea). After the occurrence has been found, we simply let the algorithm make one further "comparison" that mismatches P_{m+1} and the next text character (possibly also non-existing), and an "extra" failure function value $f_P(m+1)$ is then used in the usual way.

Definition 2.5 *The failure function f_P of the exact string matching algorithm of Morris and Pratt:*
$f_P(1) = 0$
$f_P(i) = lb_P(i-1) + 1$, *for* $2 \leq i \leq m+1$.

It is useful to note that a straighforward relationship between $f_P(i)$ and $lb_P(i-1)$ holds also when the functions are applied k times.

Lemma 2.6 *The following holds for $f_P(i)$:*
If $1 \leq k \leq |B_P(i)|$ *and* $2 \leq i \leq m+1$, *then* $f_P^k(i) = lb_P^k(i-1) + 1$.
If $k = |B_P(i)| + 1$ *and* $2 \leq i \leq m+1$, *then* $f_P^k(i) = f_P(1) = 0$.
If $i = 1$ *and* $k = 1$, *then* $f_P^k(i) = f_P(1) = 0$.

Proof: Assume first that $1 \leq k \leq |B_P(i)|$ and $2 \leq i \leq m+1$. Then, by definition of f_P, the proposition holds for $k = 1$. Now assume that it holds when $k = h$, where $h < |B_P(i)|$, and consider the value $k = h+1$. Then $f_P^{h+1}(i) = f_P(f_P^h(i)) = f_P(lb_P^h(i-1)+1) = lb_P(lb_P^h(i-1))+1 = lb_P^{h+1}(i-1)+1$. Here $lb_P^h(i-1) > 0$ since $i > 1$ and $h < |B_P(i)|$, and therefore the value $lb_P(lb_P^h(i-1)) = lb_P^{h+1}(i-1)$ exists.

Now consider the situation where $k = |B_P(i)| + 1$ and $2 \leq i \leq m+1$. We know from the previous case that $f_P^{k-1}(i) = lb_P^{k-1}(i-1)+1$. Now $lb_P^{k-1}(i-1) = \min(k \mid k \in B_P(i-1)) = 0$, because $lb_P^{k-1}(i-1)$ is the $|B_P(i)|$-largest number in $B_P(i-1)$ (i.e. the smallest). Thus $f_P^k(i) = f_P(f_P^{k-1}(i)) = f_P(lb_P^{k-1}(i-1)+1) = f_P(1) = 0$.

The last case of $i = 1$ and $k = 1$ is trivially true. We conclude by induction that the proposition holds. □

The pseudocode shown in Fig. 2 basically follows the following procedure when j denotes the index of the next compared text character and i the index of the next compared pattern character, and initially $j = 1$ and $i = 1$.

Procedure MP

1 If $P_i = T_j$ then goto step 3, else goto step 2.

2 Go through the values $f_P^k(i)$ in the order $k = 1..$ until either $P_{f_P^k(i)} = T_j$ (and thus $P_{1..f_P^k(i)} = T_{j-f_P^k(i)+1..j}$), or $f_P^k(i) = 0$ (no prefix of P ends at T_j).

3 If $i = m$, declare a found pattern occurrence $P_{1..m} = T_{j-m+1..j}$. Increment both i and j by one. Goto step 1 if $j - i \leq n - m$ (the length-m window is completely inside the text), and stop otherwise.

```
MPSearch(P, T)
1.    i ← 1, j ← 1
2.    While j − i ≤ n − m Do
3.        While i > 0 AND T_j ≠ P_i Do
4.            i ← f_P(i)
5.        If i = m Then
6.            Report occurrence at T_{j−m+1..j}
7.        i ← i + 1, j ← j + 1
```

Figure 2: The Morris-Pratt algorithm that corresponds to our exposition. This differs slightly from typically shown versions in how the shift after finding a pattern occurrence is handled. We do it after a mismatch of the non-existing character P_{m+1}, but typically it is done explicitly at the same time as a pattern occurrence is declared.

The run time of MP is $O(n)$ because the number of times that the inner loop is executed cannot be larger than the number of times that the outer loop is executed: initially $i = 1$. The inner loop is executed only as long as $i > 0$,

and each iteration decrements i by at least 1. On the other hand, the value of i is incremented only in the outer loop, and in there only by one during each iteration. Thus the total number of decrements cannot be larger than the total time of increments, which is $O(n)$.

The correctness of the MP algorithm can be derived in a straighforward manner from Lemma 2.8. But before that we introduce the following auxiliary Lemma.

Lemma 2.7 *If* $h-1 = \max(k \mid (k=0) \vee (x_{1..k} = y_{r-k..r-1}))$, *then:*
$h = \max(k \mid (k=0) \vee (x_{1..k} = y_{r-k+1..r}))$, *if* $x_h = y_r$, *and*
$h > \max(k \mid (k=0) \vee (x_{1..k} = y_{r-k+1..r}))$, *if* $x_h \neq y_r$.

Proof: Assume first that there is some $q > h$ for which $x_{1..q} = y_{r-q+1..r}$. Here we do not need to consider the option $q = 0$ in the max-clause since $q > h > 0$. Now $x_{1..q-1} = y_{r-q+1..r-1}$, and so $\max(k \mid (k = 0) \vee (x_{1..k} = y_{r-k..r-1})) \geq q - 1 \geq h > h - 1$. This is a contradiction. Therefore $h \geq \max(k \mid (k = 0) \vee (x_{1..k} = y_{r-k+1..r}))$.

If $x_h = y_r$, then $x_{1..h} = y_{r-h+1..r}$, and thus $h \leq \max(k \mid (k = 0) \vee (x_{1..k} = y_{r-k+1..r}))$. Now the preceding observation leads to the equality $h = \max(k \mid (k=0) \vee (x_{1..k} = y_{r-k+1..r}))$.

If $x_h \neq y_r$, then clearly $h \neq \max(k \mid (k = 0) \vee (x_{1..k} = y_{r-k+1..r}))$. It follows from the preceding observation that now $h > \max(k \mid (k=0) \vee (x_{1..k} = y_{r-k+1..r}))$. □

Lemma 2.8 *When the MP algorithm arrives at the text character* T_j *and the next comparison is to be made with the character* P_i, *then*
$i - 1 = \max(k \mid (k=0) \vee (P_{1..k} = T_{j-k..j-1}))$.

Proof: The proposition holds for the first text character T_1, as then $j = 1$ and $i = 1$, and clearly $i - 1 = 0 = \max(k \mid (k = 0) \vee (P_{1..k} = T_{1-k..0}))$. Now assume that $i - 1 = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k..j-1}))$ when MP arrives at the position j, where $j < n$, and consider what happens before MP moves to the next position $j + 1$. The proposition is obviously true if and only if $i = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k+1..j}))$ right before i and j are incremented. There are two cases to consider.

If $P_i = T_j$, then we have from Lemma 2.7 that $i = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k+1..j}))$.

If $P_i \neq T_j$, MP will go through the values $f_P^k(i)$ in the order $k = 1..$ until either $P_{f_P^k(i)} = T_j$ or $f_P^k(i) = 0$. Thus after this stage, and before incrementing i and j, the value i is determined by the condition $i = \max(k \mid (k = 0) \vee ((1 \leq h \leq |B_P(i-1)|) \wedge (k = f_P^h(i)) \wedge (P_k = T_j))) = \max(k \mid (k = 0) \vee (((k-1) \in B_P(i-1)) \wedge (P_k = T_j)))$. Here we used Corollary 2.4 and Lemma 2.6.

Let us first note that if $i = 1$, then MP will set $i = f_P(1) = 0$. In this case the Inductive Hypothesis states that $i - 1 = 0 = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k..j-1}))$. This makes it impossible for there to be any such $k > 1$

that $P_{1..k} = T_{j-k..j}$. And because in this case also $P_1 \neq T_j$, it follows that $\max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k+1..j})) = 0$. Thus the proposition holds.

Now we compare the values $p = \max(k \mid (k = 0) \vee (((k-1) \in B_P(i-1)) \wedge (P_k = T_j)))$ and $q = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k+1..j}))$ in the case where $i > 1$. From the Inductive Hypothesis we know that now $P_{1..i-1} = T_{j-i+1..j-1}$.

Let us first assume that $p > q$. Now $p > 0$, and therefore $p-1 \in B_P(i-1)$ and $P_p = T_j$. But this means either that $p = 1$ and $B_1 = T_j$, or that $p > 1$, $P_{1..p-1} = P_{i-p+1..i-1} = T_{j-p+1..j-1}$ and $P_p = T_j$. In both cases $P_{1..p} = T_{j-p+1..j}$, and so $p \leq \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k+1..j})) = q$. This is a contradiction, and so $p \leq y$.

Assume now that $q > p$. In this case $q > 0$ and $P_{1..q} = T_{j-q+1..j}$, and from Lemma 2.7 we know that $q \leq i - 1$. Now $q - 1 = 0$ or $P_{1..q-1} = T_{j-q+1..j-1} = P_{i-q+1..i-1}$. In both of these cases $q - 1 \in B_P(i-1)$. And because $P_q = T_j$, we have that $q \leq \max(k \mid (k = 0) \vee (((k-1) \in B_P(i-1)) \wedge (P_k = T_j))) = p$. This is a contradiction and thus $q \not> p$. Therefore it must hold that $p = q$.

From the preceding we have that MP sets $i = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k+1..j}))$ in all cases before incrementing i and j. Thus the condition $i' - 1 = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j'-k..j'-1}))$ holds for the next pattern and text indices $i' = i + 1$ and $j' = j + 1$, and we conclude by induction that the proposition is true.

□

So far we have not discussed how to precompute the values $f_P(i)$. It is an interesting fact that these values can be computed by using the MP algorithm itself. Consider the scenario where we use MP to search for P in a text where $T_{1..m-1} = P_{2..m}$. From Lemma 2.8 we know that when MP arrives at the character T_j, where $j < m$, then $i - 1 = \max(k \mid (k = 0) \vee (P_{1..k} = T_{j-k..j-1})) = \max(k \mid (k = 0) \vee (P_{1..k} = P_{j-k+1..j})) = \max(k \mid k \in B_P(j)) = lb_P(j)$. Thus, by Definition 2.5 of f_P, now $i = 1 + lb_P(j) = f_P(j + 1)$ and the values $f_P(j)$ could be computed in this way for $j = 2 \ldots m + 1$. The "missing" value $f_P(1) = 0$ is known beforehand, and poses no problem. The only delicate point here is whether the value $f_P(i)$ is always computed before it is needed in the algorithm. But since the value $f_P(i)$ can be recorded when arriving to the position $j = i$, and clearly $i \leq j$ at all times in the MP algorithm, this indeed is the case. Fig. 3 shows the pseudocode for computing the values $f_P(i)$ for $i = 1 \ldots m + 1$. It goes through the "text pattern" with an index $j^+ = j + 1$, which corresponds to the fact that $P_{j+1} = P_{j^+} = T_j$ when $T = P_{2..m}$. Further differences to the MP algorithm in Fig. 2 are that no occurrence-checking is done, and the search process is continued while $j \leq m$. Note that we set initially $i = 0$ and $j^+ = 1$, which corresponds to $j = 0$, in order to assign the special value $f_P(1) = 0$. The value $f_P(m+1)$ is recorded after the main loop: the iterations are stopped once $j^+ = m + 1$, and at that point the current value of i corresponds to $f_P(m + 1)$.

```
ComputeF(P)
1.    i ← 0, j+ ← 1
2.    While j+ ≤ m Do
3.        fP(j+) ← i
4.        While i > 0 AND Pj+ ≠ Pi Do
5.            i ← fP(i)
6.        i ← i + 1, j+ ← j+ + 1
7.    fP(m + 1) ← i
```

Figure 3: Computing the MP failure function values $f_P(i)$.

3 Computing the Boyer-Moore δ_2 function

When the pattern window is at position w in the text, the Boyer-Moore algorithm (BM) compares the characters P_i and T_j in reverse order. That is, i goes from m down towards 1, and j goes from w down towards $w - m + 1$. BM does not remember previous matches between the text and the pattern, and thus the comparisons always begin again from the last characters of the window and the pattern. When P_i and T_j mismatch during this process, the δ_2 function of BM shifts the pattern forward in the text in order to align the already matched suffix $P_{i+1..m} = T_{j+1..w}$ with some earlier part of the pattern $P_{i+1-k..m-k}$ that is equal to $P_{i+1..m}$. The value $\delta_2(i)$ gives the total shift of the text comparison index j. For example if the shift aligns $P_{i+1..m} = T_{j+1..w}$ with $P_{i+1-k..m-k}$, the corresponding value of $\delta_2(i)$ is $k + (m - i)$. Here k corresponds to the amount by which the pattern window is shifted, and $m - i$ to shifting the comparison point i within the window back to its last position. The version of δ_2 that we discuss here uses the so-called *strong good-suffix rule* [10]. If $P_{i+1-k..m-k} = P_{i+1..m} = T_{j+1..w}$, the strong good-suffix rule aligns $P_{i+1..m} = T_{j+1..w}$ with $P_{i+1-k..m-k}$ only if $P_{i-k} \neq P_i$. This is reasonable, since the characters $P_{i-k} = P_i$ and T_j would be known to mismatch at the new position of the pattern window. There are also special cases to consider. Fig. 4 shows the different possibilities that arise when determining the values of the δ_2 function. The formal definition that encloses these cases is as follows.

Definition 3.1 *The δ_2 function of the exact string matching algorithm of Boyer and Moore for $i = 1 \ldots m$:*

$$\delta_2(i) = m - i + \min\left(k \;\middle|\; \begin{array}{l} ((0 < k < i) \wedge ((i = m) \vee (P_{i+1..m} = P_{i+1-k..m-k})) \wedge (P_i \neq P_{i-k})) \vee \\ ((i \leq k < m) \wedge (P_{k+1..m} = P_{1..m-k})) \vee \\ (k = m) \end{array} \right)$$

The first line inside the min-clause of Definition 3.1 corresponds to the cases a) and c), the second line to the case b), and the third line to the case d) in Fig. 4.

Let us now turn into the eventual goal of this paper: constructing an algorithm that computes the values of the function δ_2. The end result is an amended

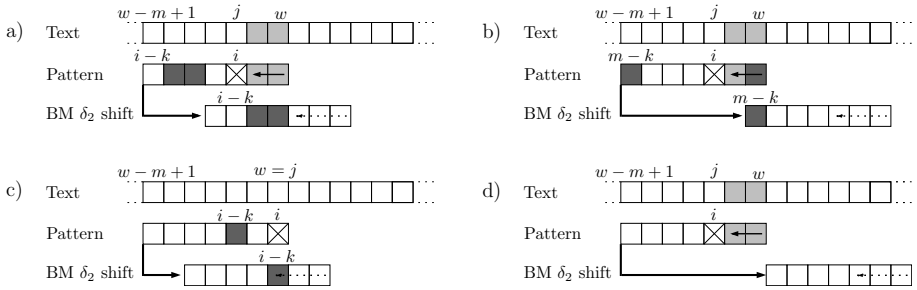

Figure 4: The four different shifting scenarios for δ_2 when $P_i \neq T_j$. In each case the window would be shifted right by k positions. Figure a) shows the case where the suffix $P_{i+1..m}$ matches with $T_{j+1..w}$, $P_{i+1..m} = P_{i+1-k..m-k}$ (shown in dark gray), and $P_i \neq P_{i-k}$. The window is shifted right in order to align the matching segments $T_{j+1..w}$ and $P_{i+1-k..m-k}$. Figure b) shows the case were $P_{1..m-k} = P_{k+1..m}$. Now the window is shifted right in order to align the matching segments $P_{1..m-k}$ and $T_{w-m+k+1..w}$. Figure c) shows the case where P_m and $T_j = T_w$ mismatch, and P_{i-k} is the first different character backwards from P_m ($P_{i-k} \neq P_m$). Now the window is shifted in order to align the possibly matching characters T_w and P_{i-k}. Figure d) shows the case where none of the preceding cases hold, and the window is shifted completely over the current window position.

version of the original preprocessing algorithm from [10]. The incompleteness of the original algorithm was shown and corrected in [12]. The correcting amendment in our construct is similar to the versions shown for example in [13, 3, 1]. In order to simplify this task, we divide the min-clause of Definition 3.1 into the following two components d_1 and d_2.

$$d_1(i) = \min(k \mid (k = m) \vee ((0 < k < i) \wedge ((i = m) \vee (P_{i+1..m} = P_{i+1-k..m-k})) \wedge (P_i \neq P_{i-k}))).$$

$$d_2(i) = \min(k \mid (i \leq k < m) \wedge (P_{k+1..m} = P_{1..m-k})).$$

Here d_1 corresponds to the first and third, and d_2 to the second line of the min-clause. We assume that when the value $d_2(i)$ is undefined, it cannot be chosen by a min-clause. In this case it is clear that $\delta_2(i) = m - i + \min(d_1(i), d_2(i)) = \min(m - i + d_1(i), m - i + d_2(i))$.

Let us denote by s^R the *reverse* string of s. This means that $s_i^R = s_{|s|-i+1}$ for $i = 1 \ldots |s|$. For example if $s =$ "abc", then $s^R =$ "cba", $s_{1..2}^R =$ "cb" and $(s_{1..2})^R =$ "ba". Note the last two examples about how the parentheses can be used in order to differentiate between "a substring of a reversed string" and "the reverse of a substring". By using this notation, we may transform the definitions of $d_1(i)$ and $d_2(i)$ into a more convenient form.

The equality $P_{i+1..m} = P_{i+1-k..m-k}$ can be written as $P_{1..m-i}^R = P_{k+1..m-i+k}^R$, and the non-equality $P_i \neq P_{i-k}$ as $P_{m-i+1}^R \neq P_{m-i+k+1}^R$. Since the condition

$i = m$ means that $m - i = 0$, and $0 \in B_{P^R}(m - i + k)$ when $1 \leq m - i + k \leq m$, the condition $(i = m) \vee (P^R_{1..m-i} = P^R_{k+1..m-i+k})$ is equivalent to the condition $(m - i) \in B_{P^R}(m - i + k)$. The requirement $0 < k < i \leq m$ guarantees that the condition $1 \leq m - i + k \leq m$ holds for this part of d_1. So now we have that $d_1(i) = \min(k \mid (k = m) \vee ((0 < k < i) \wedge ((m-i) \in B_{P^R}(m-i+k)) \wedge (P^R_{m-i+1} \neq P^R_{m-i+k+1})))$.

By doing a similar transformation as above, we have that $d_2(i) = \min(k \mid (i \leq k < m) \wedge (P_{k+1..m} = P_{1..m-k})) = \min(k \mid (i \leq k < m) \wedge ((m - k) \in B_{P^R}(m)))$.

3.1 Computing d_1

Let us now consider now how to compute the value $d_1(i) = \min(k \mid (k = m) \vee ((0 < k < i) \wedge ((m - i) \in B_{P^R}(m - i + k)) \wedge (P^R_{m-i+1} \neq P^R_{m-i+k+1})))$. Clearly one way is to first set $d_1(i) = m$ for $i = 1 \ldots m$, and then update the value $d_1(i)$ whenever such h is found that $h < d_1(i)$ and $(0 < h < i) \wedge ((m-i) \in B_{P^R}(m - i + h)) \wedge (P^R_{m-i+1} \neq P^R_{m-i+h+1})$. Here $1 \leq m - i + h \leq m - 1$ since $0 < h < i \leq m$. Suppose we use the following exhaustive procedure to initialize and then update the values.

Procedure d_1-exhaustive

1 Set $d_1(i) = m$ for $i = 1 \ldots m$, set $q = 1$, and goto step 2.

2 Inspect each element $r \in B_{P^R}(q)$ in descending order, and set $d_1(m - r) = \min(d_1(m - r), q - r)$ if $P^R_{r+1} \neq P^R_{q+1}$. Goto step 3.

3 Set $q = q + 1$. Goto step 2 if $q < m$, and stop otherwise.

Lemma 3.2 *Procedure d_1-exhaustive sets correctly the values $d_1(i)$ for $i = 1 \ldots m$.*

Proof: Consider any index i so that $1 \leq i \leq m$. Initially the procedure has set $d_1(i) = m$. The correct value of $d_1(i)$ is $h < m$ if and only if $(0 < h < i) \wedge ((m - i) \in B_{P^R}(m - i + h)) \wedge (P^R_{m-i+1} \neq P^R_{m-i+h+1})$.

Let us first assume that this is the case, that is, that the correct value is $d_1(i) = h < m$. Because in this case $1 \leq m - i + h \leq m - 1$ and the procedure inspects all elements of the sets $B_{P^R}(q)$ for $q = 1 \ldots m - 1$, the scheme will also inspect the set $B_{P^R}(m - i + h)$ and evaluate its element $m - i$. Let us use the notation $q' = m - i + h$ and $r' = m - i$ in order to relate the situation to the description of the inspection scheme. Now $r' \in B_{P^R}(q')$ and $P^R_{r'+1} \neq P^R_{q'+1}$, and the procedure thus sets $d_1(m - r') = d_1(i) = \min(d_1(m - r'), q' - r') = \min(d_1(i), h) \leq h$. Therefore we know at this point that eventually $d_1(i)$ will hold a value that is too large.

Consider now whether it is possible that $d_1(i)$ gets a too small value $h' < m$. Now denote $q = m - i + h'$ and $r = m - i$. The procedure may set $d_1(i) = d_1(m - r) = h' = q - r$ only if $r \in B_{P^R}(q)$ and $P^R_{r+1} \neq P^R_{q+1}$. But in this case h' is a legal value for $d_1(i)$. Therefore the procedure cannot set a too small value for $d_1(i)$.

From the preceding we conclude that Procedure d_1-exhaustive sets the correct value for each $d_1(i)$, $i = 1 \ldots m$. □

Procedure $d_1(i)$ can be improved by noticing the following property in its step 2.

Lemma 3.3 *If step 2 of Procedure d_1-exhaustive moves into step 3 after finding the first $r \in B_{P^R}(q)$ for which $P^R_{r+1} = P^R_{q+1}$, the values $d_1(i)$ are still set correctly for $i = 1 \ldots m$.*

Proof: The key here is that Procedure d_1-exhaustive inspects the sets $B_{P^R}(q)$ in the order $q = 1 \ldots m-1$, and the elements $r \in B_{P^R}(q)$ in the descending order $lb^k_{P^R}(q)$ for $k = 1 \ldots |B_{P^R}(q)|$.

First we note that $B_{P^R}(1) = \{0\}$, and thus the complete set $B_{P^R}(1)$ will always be inspected. Therefore the modification does not affect the correctness for this part. Now assume that the sets $B_{P^R}(q)$ have been correctly inspected for $q = 1 \ldots q' - 1$, and consider the set $B_{P^R}(q')$. If the modified scheme does not inspect the set $B_{P^R}(q')$ completely, then there must be some $r \in B_{P^R}(q')$ so that $q' > r > 0$ and $P^R_{r+1} = P^R_{q+1}$. The value $lb_{P^R}(r)$ belongs to the set $B_{P^R}(r)$, and from Lemma 2.3 we see that the values that are smaller than r are identical in the sets $B_{P^R}(r)$ and $B_{P^R}(q')$. Consider the case where there is some $r' < r$ so that $r' \in B_{P^R}(q')$ and $P^R_{r'+1} \neq P^R_{q+1}$. In this case the exhaustive procedure would set $d_1(m - r') = \min(d_1(m - r'), q' - r')$. But now also $r' \in B_{P^R}(r)$ and $P^R_{r'+1} \neq P^R r + 1$, and the inspection of the set $B_{P^R}(r)$ has been correct. Therefore the value of $d_1(m-r')$ already holds a value that is at most $r - r' < q' - r'$, and the update $d_1(m-r') = \min(d_1(m-r'), q'-r')$ has no effect. This means that no such value r' needs to be considered anymore.

□

From Lemma 2.6 we know that if $1 \leq k \leq |B_{P^R}(i)|$ and $2 \leq i \leq m+1$, then $f^k_{P^R}(i) = lb^k_{P^R}(i-1)+1$. This enables us to use the Morris-Pratt failure function in stepping through the border-sets in the inspection process: when inspecting the kth largest element $lb^k_{P^R}(q)$ in the set $B_{P^R}(q)$, where $1 \leq q \leq m-1$, we can use the value $f^k_{P^R}(q+1) - 1$. The following procedure uses this way, and it also incorporates the improvement stated by Lemma 3.3.

Procedure d_1-pruned

1 Set $d_1(i) = m$ for $i = 1 \ldots m$, set $q = 1$, and goto step 2.

2 Go through the elements $r \in B_{P^R}(q)$ in descending order by computing $r' = r + 1 = f^k_{P^R}(q+1)$ for $k = 1 \ldots |B_{P^R}(q)|$. If $P^R_{r'} = P^R_{q+1}$, interrupt the process and goto step 3, and otherwise set $d_1(m - r' + 1) = \min(d_1(m - r' + 1), q - r' + 1)$. Goto step 3.

3 Set $q = q + 1$. Goto step 2 if $q < m$, and stop otherwise.

Now we note that $i = f_{P^R}(j+1) = f_{P^R}(j^+)$ when the algorithm for computing the Morris-Pratt failure function for P^R arrives to the "text pattern" character

$T_j = P^R_{j+1} = P^R_{j^+}$. We further note that after that algorithm has first set $f_{P^R}(j^+) = i$, it begins to further compute the values $i = f^k_{P^R}(j^+)$ for $k = 1 \ldots$ as long as either $P^R_i \neq P^R_{j^+} = P^R_{j+1}$ or i becomes zero after the value $k = |B_{P^R}(q)|$. This is exactly what is done in step 2 of Procedure d_1-pruned. Also, when the algorithm goes through the values $j^+ = j + 1 = 1 \ldots m$, the sets $B_{P^R}(q)$ will be handled in the order $1 \ldots m-1$. It does not matter that j^+ begins from the value 1, which would correspond to $q = 0$, since at that point $i = 0$ and the inner loop is not yet activated.

We are eventually interested in the values $m - i + d_1(i)$. The procedure for computing d_1 can directly redord these values by simply adding $m - i$ to each value (both to the recorded values as well as the new value-candidates). Fig. 5 shows the pseudocode for computing the values $m-i+d_1(i)$. When $f^k_{P^R}(j^+) = i$ and $P^R_{j^+} \neq P^R_i$, the recorded value for the index $m - (i-1) = m - i + 1$ is $m - (m - i + 1) + (j^+ - 1) - (i - 1) = j^+ - 1$. The shown algorithm is a straightforward conversion from the algorithm in Fig. 3. The values are already recorded into the array δ_2.

```
ComputeD1(P^R)
1.  For i ∈ 1...m Do
2.      δ2(i) ← 2m − i
3.  i ← 0, j+ ← 1
4.  While j+ ≤ m Do
5.      f_PR(j+) ← i
6.      While i > 0 AND P^R_{j+} ≠ P^R_i Do
7.          δ2(m − i + 1) ← min(δ2(m − i + 1), j+ − 1)
8.          i ← f_PR(i)
9.      i ← i + 1, j+ ← j+ + 1
```

Figure 5: Computing the values $m - i + d_1(i)$.

3.2 Computing d_2

Let us now consider how to compute the value d_2 after the values d_1 have been processed. Since $d_2(i) = \min(k \mid (i \leq k < m) \wedge ((m-k) \in B_{P^R}(m)))$, d_2 can naturally be computed by inspecting all values in the set $B_{P^R}(m)$. We may transform the min-clause into the max-clause $d_2(i) = m - \max(k \mid (i \leq m-k < m) \wedge (k \in B_{P^R}(m)))$. Now it is quite obvious that $d_2(i) = m - lb^h_{P^R}(m)$ for $i = m - lb^{h-1}_{P^R}(m) + 1 \ldots m - lb^h_{P^R}(m)$, where $h = 1 \ldots |B_{P^R}(m)| - 1$. If we would choose a k larger than $lb^h_{P^R}(m)$ in the max-clause, it would have to be at least $lb^{h-1}_{P^R}(m)$. And then the values of i in the interval $m - lb^{h-1}_{P^R}(m) + 1 \ldots m - lb^h_{P^R}(m)$ would violate the rule $i \leq m - k$. On the other hand the value $lb^h_{P^R}(m)$ is valid for this interval. Therefore it is the largest such value, and thus the correct one. Note also that we are not allowed to choose the value $lb^{|B_{P^R}(m)|}_{P^R}(m) = 0$, hence the limit $h \leq |B_{P^R}(m)| - 1$.

By Lemma 2.6, we may use the rule $f^k_{P^R}(m+1) = lb^k_{P^R}(m) + 1$ in order to inspect the values in the set $lb^k_{P^R}(m)$ in descending order. The procedure is simple, and its pseudocode is shown in Fig. 6. Here we update the array δ_2 with the values $m - i + d_2(i)$, when necessary, so that it then holds the final values $\delta_2(i) = \min(m - i + d_1(i), m - i + d_2(i))$. This part of the process corresponds to the first published preprocessing algorithm for δ_2. It was not complete as it missed the values $d_2(i)$, but Rytter corrected this error in [12].

```
ComputeD2
1.   j+ ← m − i + 1
2.   i ← 1
3.   While j+ < m Do
4.       While i ≤ j+ Do
5.           δ2(i) ← min(δ2(i), m − i + j+)
6.           i ← i + 1
7.       j+ ← m − fP^R(m − j+ + 1) + 1
```

Figure 6: Computing the values $m - i + d_2(i)$. We assume that the algorithm ComputeD1 in Fig. 6 has been executed right before executing ComputeD2, so that the values $f_{P^R}(i)$ are recorded for $i = 1 \ldots m$, the initial value of i equals $f_{P^R}(m+1)$, and the array δ_2 already contains the values d_1. Throughout the computation the value j^+ will hold the upper limit $m - lb^h_{P^R}(m)$ of the currently handled interval $m - lb^{h-1}_{P^R}(m) + 1 \ldots m - lb^h_{P^R}(m)$. The lower limit of the first interval is $m - lb^0_{P^R}(m) + 1 = m - m + 1 = 1$, and so i begins from the position 1.

3.3 Addressing P instead of P^R

A difference between the algorithm we constructed and the usually shown algorithms for computing δ_2 is that we address the indices of P^R instead of P. We chose this way to make the exposition simpler. Let us now discuss what needs to be changed if we wish to address the characters of P instead. The first natural change is that each direct reference to a character P^R_i must be replaced by a reference to P^R_{m-i+1}. The second is the interchange of the "end-points". In the set $B_P(i)$, the border index 0 represents an empty border that ends before the character P_1. So when we look at the situation in the case of $B_{P^R}(i)$, this kind of index points to the non-existing character before s^R_1, which in terms of P corresponds to the non-existing character P_{m+1} after the character P_m. A third change is that the directions forward and backward are interchanged. Increment the position in P corresponds to decrementing it in P^R, and vice versa.

We now list the required changes into the algorithms in Figs. 5 and 6, and then show the complete algorithm that addresses P.

In the function ComputeD1 we have to change the following.

- Replace P^R by P.

- Since $P_i^R = P_{m-i+1}$, initialize i with $m-0+1 = m+1$ and j^+ with $m-1+1 = m$.

- The character P_m^R corresponds to the character P_1. Therefore change the condition of the outer loop into "**While** $j^+ \geq 1$ **Do**".

- Now the index 0, "the position before P_1^R", corresponds to $m-0+1 = m+1$. Therefore change the condition of the inner loop into "**While** $i < m+1$ AND $P_{j^+} \neq P_i$ **Do**".

- Since also the δ_2 function addresses P, replace i by $m-i+1$ and j^+ by $m-j^++1$ in computing the function. That is, set $\delta_2(i) \leftarrow \min(\delta_2(i), m-j^+)$ instead.

- Since going forward in P^R corresponds to going backwards in P, change the increments of i and j^+ into decrements.

Note that if the preceding changes are made, the values of f_{P^R} still point to the same characters as before. But now the characters are addressed with relation to P instead of P^R.

The function ComputeD2 requires less changing because it does not use P (or P^R), but only the already computed values $d_1(i)$ and $f_{P^R}(i)$. And since we have already originally addressed the values $d_1(i)$ in terms of P, the only change concerns how the function f_{P^R} is used: instead of $f_{P^R}(i)$, we should use the value $m-f_{P^R}(m-i+1)+1$. The reason for using $m-f_{P^R}(m-i+1)+1$ instead of simply $f_{P^R}(m-i+1)$ is that now we assume that the values of the function f_{P^R} address P instead of P^R. The changes to ComputeD2 are as follows.

- Initialize j^+ with $m-(m-i+1)+1 = i$ on the first line. Note that now i has the value $f_{P^R}(m-(m+1)+1) = f_{P^R}(0)$ after ComputeD1 is finished.

- Set $j^+ \leftarrow m-(m-f_{P^R}(m-(m-j^++1)+1)+1)+1 = f_{P^R}(m-(m-j^++1)+1) = f_{P^R}(j^+)$ on the last line.

Fig. 7 shows the complete algorithm for computing δ_2 when P is addressed. It is practically identical to the version shown in [13].

4 Conclusions

In this paper we analysed how to construct a correct version of the original algorithm for computing the δ_2 function of the very widely known Boyer-Moore exact string matching algorithm. As noted for example by Gusfield [8] and Stomp [14], the algorithm may seem quite mysterious and difficult. In fact, the present discussion is the first that we know of that truly delves into exploring the principles behind the algorithm. Our discussion also highlights the close relationship between the preprocessing algorithm for δ_2 and the Morris-Pratt exact string matching algorithm. Stomp [14] proved recently that the original (and later amended) algorithm is in principle correct, but that proof is *a-posteriori*

```
ComputeDelta2(P)
 1.   For i ∈ 1...m Do
 2.       δ2(i) ← 2m − i
 3.   i ← m + 1, j⁻ ← m
 4.   While j⁻ ≥ 1 Do
 5.       f̄P(j⁻) ← i
 6.       While i < m + 1 AND Pj⁻ ≠ Pi Do
 7.           δ2(i) ← min(δ2(i), m − j⁻)
 8.           i ← f̄P(i)
 9.       i ← i − 1, j⁻ ← j⁻ − 1
10.   j⁻ ← i
11.   i ← 1
12.   While j⁻ < m Do
13.       While i ≤ j⁻ Do
14.           δ2(i) ← min(δ2(i), m − i + j⁻)
15.           i ← i + 1
16.       j⁻ ← f̄P(j⁻)
```

Figure 7: Computing the values $\delta_2(i)$ when P is addressed. We have renamed the variable j^+ into j^-, since it now represents $j-1$, and we also denote by $\overline{f}_P$ the array f_{P^R} that addresses P instead of P^R.

and quite technical in nature, and does not give a very clear picture about *why* the algorithm is like it is. The present work is adapted from a part of [9], and therefore it is originally roughly as old as the work of Stomp. We hope that the presented constructive analysis is helpful for anyone who wishes to understand how the algorithm works, so that the thought "what a mysterious algorithm" would be changed into "what a clever algorithm".

References

[1] A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Volume A): Algorithms and Complexity*, pages 257–295. Elsevier, 1990.

[2] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. on Computing*, 15:98–105, 1986.

[3] S. Baase. *Computer Algorithms – Introduction to Desing and Analysis.* Addison-Wesley, 2nd edition, 1988.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762–772, 1977.

[5] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, Oxford, UK, 1994.

[6] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, Thierry Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[7] Z. Galil. On improving the worst case running time of the Boyer-Moore string searching algorithm. *Comm. ACM*, 22:505–508, 1979.

[8] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[9] H. Hyyrö. Merkkijonotäsmäyksestä. Master's thesis, Department of Computer and Information Sciences, University of Tampere, August 2000. Written in Finnish.

[10] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(1):323–350, 1977.

[11] J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

[12] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM J. on Computing*, 9:509–512, 1980.

[13] G. A. Stephen. *String Searching Algorithms.* World Scientific, 1994.

[14] F. Stomp. Correctness of substring-preprocessing in Boyer-Moore's pattern matching algorithm. *Theoretical Computer Science*, 290:59–78, 2003.