Deriving the Boyer-Moore-Horspool algorithm

Loek G.W.A. Cleophas

Software Engineering & Technology Group,
Dept. of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
loek@loekcleophas.com

***Abstract*—The keyword pattern matching problem has been frequently studied, and many different algorithms for solving it have been suggested. Watson and Zwaan in the early 1990s derived a set of well-known solutions from a common starting point, leading to a taxonomy of such algorithms. Their taxonomy did not include a variant of the Boyer-Moore algorithm developed by Horspool. In this paper, I present the Boyer-Moore-Horspool algorithm in the context of the taxonomy.**

I. Introduction

The *(exact) keyword pattern matching problem* can be described as "the problem of finding all occurences of keywords from a given set as substrings in a given string" [WZ96]. This problem has been frequently studied in the past, and many different algorithms have been suggested for solving it. Watson and Zwaan [WZ96], [Wat95, Chapter 4] derived a set of well-known solutions to the problem from a common starting point, factoring out their commonalities and presenting them in a common setting to better comprehend and compare them. This has lead to a taxonomy of such algorithms as the single-keyword Knuth-Morris-Pratt and Boyer-Moore algorithms, as well as the multiple-keyword Aho-Corasick and Commentz-Walter algorithms. Taxonomies like this are useful to bring order to a problem field and increase its accessibility, showing relations between the algorithms and simplifying the construction of coherent toolkits of such algorithms.

Watson and Zwaan's original taxonomy contained the single keyword Boyer-Moore algorithm [BM77] as well as a multiple keyword version of the algorithm. Three years after Boyer and Moore's paper, Horspool published a paper describing a simplification of Boyer and Moore's algorithm [Hor80] that nevertheless performs quite well in comparison on practical problem instances [NR02]. Despite its close similarity to the Boyer-Moore algorithm, Horspool's algorithm is not included in the original taxonomy or corresponding toolkit.

In 2003, under the supervision of Watson and Zwaan, I extended the original taxonomy and toolkit with a number of algorithms that had originally been omitted or that had not appeared at the time the taxonomy was constructed [Cle03], [CWZ04a]. An earlier version of the algorithm presentation occurs there and in [CWZ04b], [CWZ04a]. In this short paper, I show how Horspool's algorithm can easily be derived and presented as part of the taxonomy. In [Cle03] a different derivation and presentation of the *single-keyword* Horspool algorithm is given as well.

A. Dedication to Derrick Kourie

A lot of my work—including the algorithm exposition in this paper—falls into the Software by Construction approach, an approach very much advocated by Derrick. He has used it e.g. as the basis for his honours course Formal Aspects of Computing, for which he developed extensive lecture notes—a draft book really—together with Bruce Watson. Even though my presentation and derivation style is slightly different—I am using a Dijkstra/Feijen/Hoare style originating in Eindhoven, while he is using a Dijkstra/Morgan style—the essential approach is the same and the work fits together well.

Derrick was also a co-author on a number of papers on TABASCO (together with Watson, Andrew Boake, Sergei Obiedkov and myself). TABASCO is an approach for TAxonomy-BAsed Software COnstruction, initially developed in Eindhoven by Watson and others, but used at Pretoria by Watson, Kourie and their students as well. My MSc and PhD research was very much in this area. Derrick has always expressed a keen interest in it and asked interesting questions to ponder, during mutual visits and at conferences and workshops. For this—but perhaps just as much for his many questions and thoughts outside algorithms, computer science, or even science—I am sincerely grateful to him. I therefore am hopeful our interaction and cooperation will intensify in the future, but confident it will at least continue.

B. Preliminaries

A *string* $p = p_1...p_m$ of length m is a sequence of characters from an alphabet V. The empty string is denoted by ε. We will use p^R for the *reverse* of a string p, i.e. for $p_m...p_1$. Similarly, we use P^R for the set of strings obtained by reversing all strings from P. For $p \neq \varepsilon$ we use $p{\restriction}1$ ($p{\restriction}1$) for p's rightmost (leftmost) character, and $p{\downarrow}1$ ($p{\downarrow}1$) for p minus its rightmost (leftmost) character.

We will use **pref**(p) and **suff**(p) for the set of *prefixes* and *suffixes* of p respectively. A prefix (resp. suffix) is a *proper* prefix (resp. suffix) of a string p if it does not equal p. These notions are extended to a set of strings $P = \{p_1,p_2,...,p_r\}$ in the usual way. We will use $\leq_p$ to denote that a string is a prefix of another string.

A *deterministic finite automaton* is a 5-tuple $M = (Q, V, \delta, q_0, F)$ where Q is a finite set of states, V is an alphabet, $\delta \in Q \times V \rightarrow Q$ is a transition function, $q_0 \in Q$ is a start state, and $F \subseteq Q$ is a set of final states. We extend

transition function $\delta \in Q \times V \to Q$ to $\delta^* \in Q \times V^* \to Q$ defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$. When δ is not a total function, we call the finite automaton (FA) *weakly deterministic*.

Algorithms are described using a variation of Dijkstra's Guarded Command Language (GCL)—see [Cle03] for details. In particular, the *conditional and* is denoted by **cand**, **as** $b \to S$ **sa** is a shortcut for **if** $b \to S \,[\!]\, \neg b \to$ skip **fi**, and **for** $x : P \to S$ **rof** is used for executing S once for each value of x initially satisfying P (assuming there is a finite number of such values), where the order in which values of x are chosen is arbitrary.

C. The taxonomy

II. The problem and some naive solutions

As in previous publications on the keyword pattern matching taxonomy [WZ96], [Wat95], [Cle03], we start out with a naive solution to the problem and derive more detailed solutions from there.

Formally the keyword pattern matching problem, given an alphabet V (a non-empty finite set of symbols), an input string $S \in V^*$, and a finite non-empty pattern set $P = \{p_0, p_1, \ldots p_{|P|-1}\} \subseteq V^*$, is to establish[1]

$$R: \; O = \Big(\bigcup \; l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\} \Big) .$$

that is to let O be the set of triples (l, v, r) such that l, v and r form a splitting of input string S in three parts and the middle part is a keyword in P. For reasons of simplicity, we assume that $P \neq \varnothing$ and that $\varepsilon \notin P$ for the remainder of this paper.

Example II.1. *For keyword set $P = \{her, his, she\}$ and input string $S = hishershey$, set O will contain four triples, i.e.*

$$\begin{array}{lll} (\varepsilon, & his, & hershey), \\ (hi, & she, & rshey), \\ (his, & her, & shey), \\ (hisher, & she, & y). \end{array}$$

□

A trivial (but unrealistic) solution to the problem is

Algorithm II.2()

$O := \big(\bigcup \; l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\} \big)$
$\{ \; R \; \}$

Two basic directions in which to proceed while developing naive algorithms to solve this problem are, informally, to consider a substring of S as "suffix of a prefix of S" or as "prefix of a suffix of S". We consider the first possibility (the second leads to mirror images of algorithms obtained here). This is the way that algorithms such as the Boyer-Moore and Boyer-Moore-Horspool algorithms treat substrings of input string S.

Formally, we can consider "suffixes of prefixes of S" as follows:

$$\begin{array}{ll} & \big(\bigcup \; l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\} \big) \\ = & \quad \{ \text{ introduce } u : u = lv \, \} \\ & \big(\bigcup \; l, v, r, u : ur = S \wedge lv = u \wedge v \in P : \{(l, v, r)\} \big) \\ = & \quad \{ \text{ nesting } \} \\ & \big(\bigcup \; u, r : ur = S : X(u, r) \big) \end{array}$$

where $X(u, r) = \big(\bigcup \; l, v : lv = u \wedge v \in P : \{(l, v, r)\} \big)$. We obtain a simple nondeterministic algorithm by applying "examine prefixes of a given string in any order" (algorithm detail (P)) to input string S:

Algorithm II.3(P)

$O := \varnothing$;
for $(u, r) : ur = S \to$
    $O := O \cup \big(\bigcup \; l, v : lv = u \wedge v \in P : \{(l, v, r)\} \big)$
rof$\{ \; R \; \}$

This algorithm is (also) used as a starting point to derive e.g. the Aho-Corasick algorithm and variants in [WZ96], [Wat95].

Here, we consider how to update O in the repetition of Algorithm II.3. The update can be computed with another nondeterministic repetition. This inner repetition would consider suffixes of u. Thus by applying "examine suffixes of a given string in any order" (algorithm detail (S)) to string u we obtain:

Algorithm II.4(P, S)

$O := \varnothing$;
for $(u, r) : ur = S \to$
    for $(l, v) : lv = u \to$
        as $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
    rof
rof$\{ \; R \; \}$

Algorithm (P, S) consists of two nested nondeterministic repetitions. In each case, the repetition can be made deterministic by considering prefixes (or suffixes as the case is) in increasing (called detail (+)) or decreasing (detail (−)) order of length. This gives two binary choices. Since the Boyer-Moore and Boyer-Moore-Horspool algorithms examine string S from left to right, and the patterns in P from right to left we focus our attention on the following algorithm:

Algorithm II.5(P_+, S_+)

[1] Note that the problem definition is slightly different but equivalent to that used in [WZ96], [Wat95]. As a result, the algorithms given in this text will be slightly different in structure but equivalent in meaning to the algorithms of the same name in those texts.

$u, r := \varepsilon, S;$
$O := \varnothing;$
$\{$ **inv** $ur = S$

$$\wedge\, O = \left(\bigcup x, y, z : \begin{array}{ll} xyz = S & : \{(x, y, z)\} \\ \wedge\, xy \leq_p u & \\ \wedge\, y \in P & \end{array} \right) \}$$

do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;$
  $l, v := u, \varepsilon;$
  $\{$ **inv** $u = lv$ $\}$
  do $l \neq \varepsilon \rightarrow$
    $l, v := l\downarrow 1, (l\restriction 1)v;$
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
od$\{$ R $\}$

This algorithm has running time $\Theta(|S|^2)$, assuming that computing membership of P is a $\Theta(1)$ operation.

III. Suffix-based pattern matching

To improve the running time of Algorithm II.5, we consider the keyword suffixes, **suff**(P). We know that $w \in$ **suff**$(P) \equiv \big(\exists x : x \in V^* : xw \in P\big)$. It follows that if $w \notin$ **suff**(P) any extension of w on the left is not an element of **suff**(P) either. Consequently, the inner repetition in Algorithm II.5 can terminate as soon as $(l\restriction 1)v \notin$ **suff**(P) holds, since then all suffixes of u that are equal to or longer than $(l\restriction 1)v$ are not in **suff**(P) either and hence not in P.

Example III.1. *Consider the situation in Algorithm II.5 with $S = hishershey$ and $P = \{her, his, she\}$ as before, when execution is at the beginning of the inner loop and $(l, v, r) = (hish, \varepsilon, ershey)$. Note that $(l\restriction 1)v = h\varepsilon = h$ is not a suffix of any keyword, i.e. $(l\restriction 1)v \notin$ **suff**(P). Clearly, no extension of v to the left can lead to additional matches, so the inner loop can be terminated.* □

The inner repetition guard can therefore be strengthened to

$$l \neq \varepsilon \textbf{ cand } (l\restriction 1)v \in \textbf{suff}(P).$$

Observe that $v \in$ **suff**(P) is now an invariant of the inner repetition. This invariant is initially established by the assignment $v := \varepsilon$ since $P \neq \varnothing$ and thus $\varepsilon \in$ **suff**(P).

Algorithm detail III.2. *(*GS=S*). (*Guard Strengthening = Suffix*). Strengthen the guard of the inner repetition by adding conjunct $(l\restriction 1)v \in$ **suff**(P).* □

Direct evaluation of $(l\restriction 1)v \in$ **suff**(P) is expensive. Therefore, it is done using the transition function of a finite automaton based on **suff**$(P)^R$, with the following properties:

Property III.3 (Transition function of automaton recognizing $\mathbf{f}(P)^R$)**.** *The transition function $\delta_{R,\mathbf{f},P}$ of a (weakly deterministic) FA $M = (Q, V, \delta_{R,\mathbf{f},P}, q_0, F)$ recognizing $\mathbf{f}(P)^R$ has the property that*

$$\delta^*_{R,\mathbf{f},P}(q_0, w^R) \neq \perp \equiv w^R \in \mathbf{f}(P)^R$$

and we assume

$$\delta_{R,\mathbf{f},P}(q, \varepsilon) = q.$$

□

Note that Property III.3 requires **pref**$(\mathbf{f}(P)^R) \subseteq \mathbf{f}(P)^R$ i.e. **suff**$(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$.

Since we will always refer to the same set P in the remainder of this document, we will use $\delta_{R,\mathbf{f}}$ instead of $\delta_{R,\mathbf{f},P}$.

Note that function **suff** satisfies the requirements on function **f** stated by Property III.3. Transition function $\delta_{R,\textbf{suff}}$ can be computed beforehand [WZ96, section 4.1][2].

Example III.4. *For $P = \{his, her, she\}$, an automaton satisfying Property III.3 for* **f** = **suff** *(i.e. recognizing precisely* **suff**$(P)^R$*) is depicted in Figure 1.* □

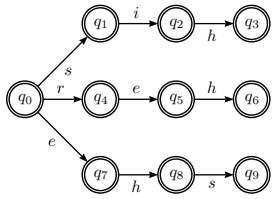

Fig. 1. Reverse suffix automaton recognizing **suff**$(P)^R$.

By making $q = \delta^*_{R,\textbf{suff}}(q_0, ((l\restriction 1)v)^R)$ an invariant of the inner repetition of the algorithm, we can use the following algorithm detail:

Algorithm detail III.5. *(*EGC=RSA*). (*Efficient Guard Computation = Reverse Suffix Automaton*). Given an FA based on* **suff**$(P)^R$ *and satisfying Property III.3, update a state variable q to uphold invariant $q = \delta^*_{R,\textbf{suff}}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in$* **suff**(P) *then becomes $q \neq \perp$.* □

Algorithm III.6(P$_+$, S$_+$, GS=S, EGC=RSA)

$u, r := \varepsilon, S;$
$O := \varnothing;$
$\{$ **inv** $ur = S$

$$\wedge\, O = \left(\bigcup x, y, z : \begin{array}{ll} xyz = S & : \{(x, y, z)\} \\ \wedge\, xy \leq_p u & \\ \wedge\, y \in P & \end{array} \right) \}$$

do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;$
  $l, v := u, \varepsilon;$
  $q := \delta_{R,\textbf{suff}}(q_0, l\restriction 1);$
  $\{$ **inv** $u = lv \wedge v \in$ **suff**(P)
    $\wedge\, q = \delta^*_{R,\textbf{suff}}(q_0, ((l\restriction 1)v)^R)$ $\}$

[2] In [WZ96], the transition function is called τ_P.

do $l \neq \varepsilon$ **cand** $q \neq \bot \rightarrow$
$l, v := l\lfloor 1, (l\lceil 1)v;$
$q := \delta_{R,\textbf{suff}}(q, l\lceil 1);$
as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
$\{\ l = \varepsilon$ **cor** $(l\lceil 1)v \notin$ **suff**$(P)\ \}$
od$\{\ R\ \}$

Assuming P is constant, $\big(\textbf{MAX}\, p : p \in P : |p|\big)$—the length of a longest keyword in P—is constant and this algorithm has $\Theta(|S|)$ running time.

IV. Suffix-based sublinear pattern matching

In Section 3 of [WZ96], a family of sublinear keyword pattern matching algorithms is derived starting from Algorithm III.6. The basic idea is to make shifts *of more than one symbol*. This is accomplished by replacing $u, r := u(r\lceil 1), r\lfloor 1$ by $u, r := u(r\lceil k), r\lfloor k$ for some k satisfying $1 \leq k \leq \big(\textbf{MIN}\, n : 1 \leq n \wedge \textbf{suff}(u(r\lceil n)) \cap P \neq \varnothing : n\big)$. The upper bound is the distance to the next match, the *maximal safe shift distance*. Any smaller number k satisfying the equation is safe as well, and we thus define a safe shift distance as:

Definition IV.1 (Safe shift distance)**.** *A shift distance k satisfying*

$$1 \leq k \leq \big(\textbf{\textit{MIN}}\, n : 1 \leq n \wedge \textbf{suff}(u(r\lceil n)) \cap P \neq \varnothing : n\big)$$

is called a safe shift distance. □

Example IV.2. *Consider the situation in Algorithm III.6 with $S = hishershey$ and $P = \{her, his, she\}$ as before, when execution is at the guard of the inner loop and $(l, v, r) = (h, \varepsilon, ishershey)$. As $(l\lceil 1)v \notin$ **suff**(P), $q = \bot$ will hold, so the inner loop will not be executed. After the assignments at the beginning of the outer loop have been executed again, we will have $(l, v, r) = (hi, \varepsilon, shershey)$ with a shift of* one *position, but a shift to $(l, v, r) = (his, \varepsilon, hershey)$ i.e. a shift of* two *positions would be optimal.* □

Algorithm detail IV.3. *(*SSD*). Replace assignment $u, r := u(r\lceil 1), r\lfloor 1$ in Algorithm III.6 (*P$_+$, S$_+$, GS=S, EGC=RSA*) by assignment $u, r := u(r\lceil k), r\lfloor k$ using a safe shift distance k.* □

In [WZ96], various approximations from below of the maximal safe shift distance are derived by weakening the predicate **suff**$(u(r\lceil n)) \cap P \neq \varnothing$. This results in safe shift distances that are easier to compute than the maximal safe shift distance. In these derivations, the $u = lv \wedge v \in$ **suff**(P) part of the invariant of the inner repetition in Algorithm III.6 is used. By adding $l, v := \varepsilon, \varepsilon$ to the initial assignments of the algorithm, we turn this into an invariant of the outer repetition. This also turns $l = \varepsilon$ **cor** $(l\lceil 1)v \notin$ **suff**(P)—the negation of the guard of the inner repetition—into an invariant of the outer repetition.

Since shift functions may depend on l, v and r, we will write $k(l, v, r)$.

We arrive at the following algorithm skeleton:

Algorithm IV.4(P$_+$, S$_+$, GS=S, EGC=RSA, SSD)

$u, r := \varepsilon, S;$
$O := \varnothing;$
$l, v := \varepsilon, \varepsilon;$
$\{$ **inv** $ur = S$

$$\wedge\ O = \left(\bigcup x, y, z : \begin{array}{ll} xyz = S & : \{(x, y, z)\} \\ \wedge\ xy \leq_p u & \\ \wedge\ y \in P & \end{array}\right)$$

$\wedge\ u = lv \wedge v \in$ **suff**(P)
$\wedge\ \big(l = \varepsilon$ **cor** $(l\lceil 1)v \notin$ **suff**$(P)\big)\ \}$
do $r \neq \varepsilon \rightarrow$
$u, r := u(r\lceil k(l,v,r)), r\lfloor k(l,v,r);$
$l, v := u, \varepsilon;$
$q := \delta_{R,\textbf{suff}}(q_0, l\lceil 1);$
$\{$ **inv** $q = \delta^*_{R,\textbf{suff}}(q_0, ((l\lceil 1)v)^R)\ \}$
do $l \neq \varepsilon$ **cand** $q \neq \bot \rightarrow$
$l, v := l\lfloor 1, (l\lceil 1)v;$
$q := \delta_{R,\textbf{suff}}(q, l\lceil 1);$
as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
od$\{\ R\ \}$

Based on this algorithm skeleton, various shift functions are derived in [WZ96][3] by weakening the predicate **suff**$(u(r\lceil n)) \cap P \neq \varnothing$ from the maximal safe shift distance. There, this leads among others to the Commentz-Walter, Fu-San and multiple keyword Boyer-Moore algorithms.

V. The multiple-keyword Horspool algorithm

In this section, we consider two particular weakenings of the range predicate **suff**$(u(r\lceil n)) \cap P \neq \varnothing$:

$$V^* v V^n \cap V^* P \neq \varnothing$$

(used in algorithm detail (NLA) given in [WZ96, Section 3.7]) and

$$V^* (l\lceil 1)v V^n \cap V^* P \neq \varnothing$$

(used in [WZ96, Section 3.2, page 13]). Note that the new predicates only refer to l and v, but not to r. Informally, this amounts to discarding any *right lookahead* when determining a shift. We now further weaken the first predicate, assuming $v \neq \varepsilon$:

$$\begin{array}{ll} & V^* v V^n \cap V^* P \neq \varnothing \\ \Rightarrow & \quad \{\ v = (v\lfloor 1)(v\lceil 1)\ \} \\ & V^* (v\lfloor 1)(v\lceil 1) V^n \cap V^* P \neq \varnothing \\ \Rightarrow & \quad \{\ v\lfloor 1 \in V^*\ \} \\ & V^* (v\lceil 1) V^n \cap V^* P \neq \varnothing \end{array}$$

We now further weaken the second predicate, assuming $v = \varepsilon$:

[3]The algorithm skeleton is called (P$_+$, S$_+$, RT, SSD) there.

$$
\begin{array}{ll}
& V^*(l\restriction 1)vV^n \cap V^*P \neq \varnothing \\
\Rightarrow & \quad \{\, v = \varepsilon \,\} \\
& V^*(l\restriction 1)V^n \cap V^*P \neq \varnothing
\end{array}
$$

Note the close resemblance between the two weakened predicates: the only difference is that the first refers to $(v\restriction 1)$ assuming $v \neq \varepsilon$ whereas the second refers to $(l\restriction 1)$ assuming $v = \varepsilon$. Using the two weakened predicates (depending on whether $v = \varepsilon$ or $v \neq \varepsilon$) we get a practical safe shift distance. We show the case $v \neq \varepsilon$ here:

$$
\begin{array}{ll}
& \big(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing : n\big) \\
\geq & \quad \{\ \text{weakening steps above}\ \} \\
& \big(\mathbf{MIN}\, n : 1 \leq n \wedge (V^*(v\restriction 1)V^n \cap V^*P \neq \varnothing) : n\big)
\end{array}
$$

In [WZ96, Section 3.4] $char_{bm} \in V \to \mathbb{N}$ is defined for all $a \in V$ by

$$char_{bm}(a) = \big(\mathbf{MIN}\, n : n \geq 1 \wedge V^*aV^n \cap V^*P \neq \varnothing : n\big)$$

and hence we have $char_{bm}(v\restriction 1)$ as a safe shift for case $v \neq \varepsilon$. Similarly, for case $v = \varepsilon$ we have $char_{bm}(l\restriction 1)$ as a safe shift. Function $char_{bm}$ is the *bad character shift* function used on $l\restriction 1$ in the original Boyer-Moore algorithm.

Algorithm IV.4 makes a shift at the beginning of the main loop. As noted above, $|l| \geq 1$ does not hold initially, and both l and v equal ε. The initial shift therefore cannot be based on either $v\restriction 1$ or $l\restriction 1$. Given $\varepsilon \notin P$, we set the initial shift to 1 (in fact, it could be set to the length of the shortest keyword in P)[4].

This then allows the use of

Definition V.1 (Shift function k_{bmh})**.** *Shift function k_{bmh} is defined as:*

$$k_{bmh}(l, v) = \begin{cases} char_{bm}(v\restriction 1) & \text{if } v \neq \varepsilon, \\ char_{bm}(l\restriction 1) & \text{if } v = \varepsilon \wedge l \neq \varepsilon, \\ 1 & \text{if } v = \varepsilon \wedge l = \varepsilon. \end{cases}$$

□

Example V.2. *For $P = \{her, his, she\}$ as before, we have*

a	e	h	i	r	s	y
$char_{bm}(a)$	1	1	1	3	2	3

□

Algorithm detail V.3. *(*BMH*). (*Boyer-Moore-Horspool*). Calculating the shift distance using k_{bmh} is algorithm detail (*BMH*).* □

The above derivation and presentation show that algorithms using shift function k_{bmh} use just the bad character shift function of the original Boyer-Moore algorithm, instead of also using the more complex *good suffix shift* function as the original algorithm does.

The use of shift function k_{bmh} yields algorithm (P+, S+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, BMH), the Set Horspool algorithm as presented in e.g. [NR02, Subsection 3.3.2]. Finally adding problem detail (OKW) ("one keyword", i.e. restricting set P to contain exactly one keyword) leads to the single-keyword Horspool algorithm [NR02, Subsection 2.3.2] originally developed by Horspool [Hor80].

Example V.4. *We present part of a complete run of the Boyer-Moore-Horspool algorithm here.*

Consider the situation when $(l, v, r) = (hisher, \varepsilon, shey)$ (and hence $u = lv = hisher$ and $q = q4$) and execution is at the beginning of the inner loop. After updating l, v, q they equal $hishe, r, q5$. Since the guard of the inner loop still holds true, it is executed again, updating l, v, q to $hish, er, q6$. Since the guard of the inner loop still holds true, it is executed again, updating l, v, q to $his, her, \perp$. As $her \in P$, the triple $(his, her, shey)$ is added to set O, i.e. a match of her is detected and registered.

Since the guard of the inner loop no longer holds true, the algorithm executes the outer loop again (as $r \neq \varepsilon$). As $v \neq \varepsilon$, $k_{bmh}(l, v) = char_{bm}(v\restriction 1) = char_{bm}(r) = 3$ i.e. a shift of 3 positions occurs, updating u, r to $hishershe, y$, updating l, v to $hishershe, \varepsilon$, and q to $q7$. The inner loop is executed three times again, leading to detection and registration of a match of she.

The guard of the inner loop no longer holds true by then, so the algorithm executes the outer loop again. A shift of 1 position is made based on $char_{bm}(e)$, before u, r are updated to $hishershey, \varepsilon, l, v$ to $hishershey, \varepsilon$, and q to $\perp$. The guard of the inner loop does not hold true, and the guard of the outer loop no longer holds true either, ending program execution. □

VI. Concluding remarks

In this paper we presented the Boyer-Moore-Horspool algorithm in the context of the taxonomy of keyword pattern matching algorithms developed by Watson and Zwaan and later extended by Cleophas. The presentation shows that, given such a taxonomy, the addition of algorithms similar to those in the taxonomies is fairly straightforward. A different presentation and placement of the *single-keyword* Horspool algorithm in the keyword pattern matching taxonomy can be found in [Cle03].

[4] The presentation of the Boyer-Moore-Horspool algorithm in the taxonomy as presented in [Cle03], [CWZ04b], [CWZ04a] does not deal with this initialization correctly. The definition of the shift function and algorithm given there was incorrect and may lead to matches at the beginning of the input string being skipped. The implementation of the algorithm in the toolkit accompanying the taxonomy *is* correct, dealing with the problem as we do here.

References

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):62–72, 1977.

[Cle03] Loek G. W. A. Cleophas. Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms. Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, August 2003.

[CWZ04a] Loek Cleophas, Bruce W. Watson, and Gerard Zwaan. A new taxonomy of sublinear keyword pattern matching algorithms. Technical Report 04/07, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2004.

[CWZ04b] Loek Cleophas, Bruce W. Watson, and Gerard Zwaan. Automaton-based sublinear keyword pattern matching. In *Proceedings of the 11th international conference on String Processing and Information REtrieval (SPIRE 2004)*, volume 3246 of *LNCS*. Springer, October 2004.

[Hor80] R. Nigel Horspool. Practical fast searching in strings. *Software—Practice & Experience*, 10(6):501–506, 1980.

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, September 1995.

[WZ96] B. W. Watson and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Science of Computer Programming*, 27(2):85–118, 1996.