String Searching over Small Alphabets

Mátyás A. Sustik J S. Moore

Technical Report
TR-07-62
Department of Computer Sciences
University of Texas at Austin

December 11, 2007

***Abstract*— We propose a new string searching procedure inspired by the Boyer–Moore algorithm. The two key ideas of our improvement are to keep track of *all* the previously matched characters within the current alignment and not to move the reading position unconditionally to the end of the pattern when a mismatch occurs. The result is an algorithm with increased average shift amounts and a guarantee that any character of the text is read at most once. The algorithm performs especially well when the alphabet size is small. We also discuss and test variants of the original idea aimed at practical implementations.**

***Index Terms*— string searching, Boyer-Moore algorithm, pattern matching, finite automaton**

1. Introduction

The task of finding the first, or all of the occurrences of a pattern in a text—*string searching*—arises in many computing applications. A string searching algorithm aligns the pattern with the beginning of the text and keeps shifting the pattern forward until a match or the end of the text is reached.

The Boyer-Moore algorithm [4] and its variants [1], [2], [5], [8], [11], [12] perform best when preprocessing of the text is not possible or not desired. The search is linear in the size of the text, and in fact it has been shown [9] that no character needs to be read (and compared) more than three times. Subsequent variants guarantee at most $2n$ comparisons for a text consisting of n characters [7], [8]. In a typical case, fewer than n character reads are needed because the average shift amount is linear in the alphabet size [3], [13]. One of the first implementations of the algorithm, searching English text, carried out fewer machine instructions than the number of characters in the input. We find it natural to describe these types of algorithms with an automaton [2], [10] and hence will take advantage of this representation to describe our contribution.

In this paper, we propose a new variant of the Boyer-Moore algorithm, which performs well for small alphabets because the shift amounts increase with the pattern length. An example of an application where small alphabets arise is searching DNA sequences.

The new algorithm reads the characters of the text at most once, and the preprocessing step that builds the corresponding automaton is polynomial in the size of the pattern. We also develop and test variants aimed at practical implementations.

Recent research on string searching proposed using probabilities [12] and ordered binary decision diagrams [6]. Two–dimensional string searching has also been considered [14].

2. Algorithm

A. Boyer–Moore algorithm

We present a slightly modified version of the original Boyer-Moore string searching algorithm of [4] as *Algorithm 1*. A single lookup table replaces the two tables of the original version, providing equal or larger shift amounts for an improved average case behavior. Note however, that for some well crafted patterns and texts *Algorithm 1* may carry out more comparisons.

A preprocessing step calculates lookup table S from pattern P; the text is not used in this step. In the main loop, we attempt to match the rightmost still unmatched character according to the current alignment of the pattern. A mismatch rules out the current alignment and the pattern shifts to the right by an amount determined from table S. We construct table S to yield the largest possible shift amount the pattern may move forward without missing a subsequent full match. At the beginning of every iteration, the last $m-1-readPos$ characters of the pattern are known to have been previously successfully matched. In the rest of the paper, we will continue to use the notations introduced by *Algorithm 1*.

Algorithm 1 The Boyer-Moore string search algorithm implemented with a single lookup table.

```
Input: $m$ character long pattern $P$, $n$ character long text $T$.
Output: The first matching alignment if any.
 1: Calculate the shift lookup table $S$ from the pattern $P$.
 2: Set $align = 0$ and $readPos = m - 1$.
 3: while $align < n - m$ do
 4:   Read character $T(align + readPos)$ into $c$.
 5:   $shift = S[m - 1 - readPos, c]$.
 6:   if $shift = 0$ then
 7:     if $readPos = 1$ then
 8:       return $align$.
 9:     end if
10:     $readPos = readPos - 1$.
11:   else
12:     $align = align + shift$.
13:     $readPos = m$.
14:   end if
15: end while
16: return $NIL$.
```

Table I shows the shift amounts for the example pattern “CABAB” over the alphabet consisting of characters A, B and

TABLE I. Shift table on the left and the previously matched and the current reading positions for the pattern "CABAB" on the right. A "*" indicates the current reading position; "X" marks the already matched characters.

	A	B	C	C	A	B	A	B
0	1	0	4					$*$
1	0	5	5				$*$	X
2	5	0	2			$*$	X	X
3	0	5	5		$*$	X	X	X
4	5	5	5	$*$	X	X	X	X

C. To the right of the shift table we also indicate the previously matched character positions and the current reading position as an aid to the reader. We use $m - 1 - readPos$ as an index when performing the lookup; clearly $readPos$ could have also been used by reversing the row order. We chose this representation because of its closer visual resemblance to the corresponding state transition table of our improved algorithm.

Here, we are not primarily concerned about the cost of the preprocessing of the pattern. Note however, that the tables used in the original version of the Boyer–Moore algorithm are linear in m and can be computed in $O(m)$ time[1]. The shift table of *Algorithm 1* has mt entries, where t denotes the alphabet size. A naive implementation can compute the table in $O(m^3t)$ complexity, but one may hope for an $O(mt)$ solution.

B. State transition table

In this section, we modify *Algorithm 1* to use the lookup table S to do more of the bookkeeping needed inside the loop. The advantage of the new formulation will become apparent when we introduce our improved algorithm. One may also expect the additional setup costs to be well compensated by the more compact loop body.

In effect, we realize a finite automaton with the lookup table. The Knuth–Morris–Pratt algorithm [10] and variants of the Boyer–Moore algorithm are naturally implemented with a finite automaton [2]. We emphasize that in the latter case the characters of the input text are not read in sequential order.

Algorithm 2 The Boyer-Moore string search algorithm implemented with a state transition table.

```
Input: $m$ character long pattern $P$, $n$ character long text $T$.
Output: The first matching position if any.
 1: Calculate the state transition table $S$ from the pattern $P$.
 2: Set $align = 0$, $state = 0$ and $readPos = m - 1$.
 3: while $align < n - m$ do
 4:     Read character $T(align + readPos)$ into $c$.
 5:     $(shift, state, readPos, match) = S[state, c]$.
 6:     if $match = true$ then
 7:         return $align$.
 8:     end if
 9:     $align = align + shift$.
10: end while
11: return $NIL$.
```

[1]If the number of bits needed to represent the entries are also considered, the complexity becomes $O(m \log m)$.

TABLE II. State transition table for the pattern "CABAB". The triplets contain the shift amount, the next state index, and the new reading position, in that order. Bold typeset indicates a full match.

	A	B	C	C	A	B	A	B
0	$(1,0,4)$	$(0,1,3)$	$(4,0,4)$					$*$
1	$(0,2,2)$	$(5,0,4)$	$(5,0,4)$				$*$	X
2	$(5,0,4)$	$(0,3,1)$	$(2,0,4)$			$*$	X	X
3	$(0,4,0)$	$(5,0,4)$	$(5,0,4)$		$*$	X	X	X
4	$(5,0,4)$	$(5,0,4)$	$\mathbf{(5,0,4)}$	$*$	X	X	X	X

The new table S of *Algorithm 2* determines not only the $shift$ amount, but also the next state, the new reading position and whether or not a full match has been found. Using the pattern $CABAB$, the state transition table appears in Table II. Notice, that the transition from every state is always made to either to the next state (according to the indexing), or to the starting state which has index zero.

Algorithm 2 can be easily adapted to find all matches of the pattern in the text; the state transition table does not need to be modified.

During the execution of the Boyer-Moore algorithm a mismatch moves the relative reading position to the last character of the pattern. No information is kept about the previously read (and matched) characters, unless they are in a single block at the end of the pattern. This suggests that for random texts and patterns, the shift amount is linear in the alphabet size [3], [13] (and limited by the pattern length).

One natural attempt for improvement is to allow the algorithm to remember the previously matched blocks, facilitating longer shifts. A more complicated preprocessing of the pattern (producing an automata) or building a data structure on the fly is needed in this case. Our experiments indicate an improvement of a constant factor in the expected shift amounts for every additional remembered block. Therefore, for longer patterns we need more and more complicated bookkeeping and/or memory management in order to have the shift amount increase with the pattern size.

One of the reasons for the success of the classical Boyer-Moore algorithm is that it allows very fast implementations using a tight loop with a small memory footprint. The memory requirements and the additional processing needed by the larger data structures mentioned above (see the related [6]), can diminish the gains provided by the larger shift amounts.

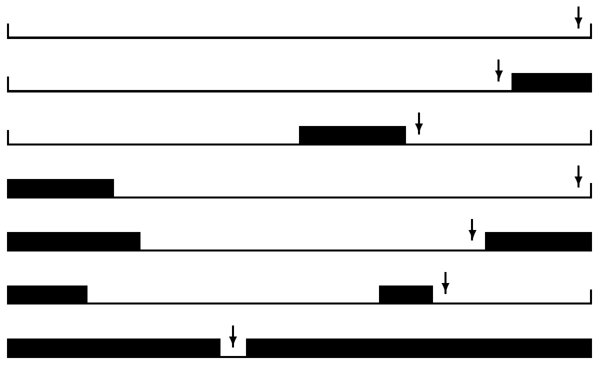
C. New algorithm

Our proposed algorithm manages to achieve larger shifts, remembering only two blocks of the previously matched characters, hence limiting the bookkeeping requirements and memory use.

In case of a mismatch and subsequent shift, we do not move the relative reading position to the end of the pattern. Instead, we attempt to read a character at one end of a block of already matched characters, thereby extending the block. We pick the right end, if possible.

The shift may cause a matching block to reach or pass the

Fig. 1. Block configurations for the new algorithm.



beginning of the pattern. Consider the following situation:

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ . & . & . & . & . & * & B & C & . & . \\ B & C & B & C & C & A & B & C & & \end{array}$$

The pattern in this example is "BCBCCABC" and the first row depicts the text with "."-s indicating the unread characters. The "*" marks the character, that will be read next. If a "B" is read, then we shift the pattern to the position indicated below and prepare to read the character at position 13:

$$\begin{array}{cccccccccc} 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ . & B & B & C & . & . & . & . & . & * \\ & & B & C & B & C & C & A & B & C \end{array}$$

The "B" we have just read, has moved out of scope due to the shift, leaving the two characters at positions 6 and 7 to match the pattern. Another "B" at position 13 would prompt a shift by one in the classical Boyer-Moore algorithm, however positions 6 and 7 rule out that alignment and a shift by 5 becomes possible.

In general, our algorithm remembers two matching blocks of characters: the left block always starts at the beginning of the pattern and we attempt to extend the right block with every read. If possible, we extend the right block to the right. In case of an empty right block, we read the character aligned with the last character of the pattern. Both blocks may degenerate into the empty string leaving the possibilities illustrated on Figure 1. Solid bars indicate the already matched characters, while the current reading position is marked by a downward pointing arrow.

The first row shows the starting state, which also arises whenever all previously matched positions are shifted out of scope. The second row shows the result of three successful matches. The third row indicates that we read on the right end of the right block, if possible. The examples in rows 4, 5, 6 and 7 show the possibilities with a nonempty left block, in particular row 7 depicts a state when a successful match implies a full match of the whole pattern.

Next, we formally describe the information maintained about the previously read and matched characters (recall T, P and $align$ from *Algorithm 2*):

- The number of characters in the left block is indicated by $left$. That is $T(align + i) = P(i)$ for $0 < i \leq left$. If $left = 0$, then the left block is empty; this is also the initial state.
- The right block is described by its endpoints: $rightStart$ and $rightEnd$, and we have $T(align + i) = P(i)$ if $rightStart \leq i < rightEnd$. Note, that $rightStart = rightEnd$ indicates an empty block.

TABLE III. State transition table for pattern $CABAB$ based on the implicit tracking of two blocks. The triplets contain the shift amount, the next state index, and the new reading position in that order. The full match is indicated by bold typesetting. On the right we show the previously matched characters marked with an "X", while a "*" indicates the current reading position.

	A	B	C	C	A	B	A	B
0	$(1,1,4)$	$(0,2,3)$	$(4,3,4)$					$*$
1	$(5,0,4)$	$(0,4,2)$	$(4,3,4)$				X	$*$
2	$(0,4,2)$	$(5,0,4)$	$(5,0,4)$				$*$	X
3	$(1,1,4)$	$(0,5,3)$	$(4,3,4)$	X				$*$
4	$(5,0,4)$	$(0,6,1)$	$(2,7,4)$			$*$	X	X
5	$(0,8,2)$	$(5,0,4)$	$(5,0,4)$	X			$*$	X
6	$(0,9,0)$	$(5,0,4)$	$(5,0,4)$		$*$	X	X	X
7	$(3,10,2)$	$(0,11,3)$	$(4,3,4)$	X	X	X		$*$
8	$(5,0,4)$	$(0,12,1)$	$(2,7,4)$	X		$*$	X	X
9	$(5,0,4)$	$(5,0,4)$	$(\mathbf{5},\mathbf{0},\mathbf{4})$	$*$	X	X	X	X
10	$(3,0,4)$	$(0,13,3)$	$(2,3,4)$		X	$*$		
11	$(\mathbf{5},\mathbf{0},\mathbf{4})$	$(5,0,4)$	$(5,0,4)$	X	X	X	$*$	X
12	$(\mathbf{5},\mathbf{0},\mathbf{4})$	$(5,0,4)$	$(5,0,4)$	X	$*$	X	X	X
13	$(0,14,4)$	$(4,0,4)$	$(3,3,4)$		X	X	$*$	
14	$(5,0,4)$	$(0,9,0)$	$(4,3,4)$		X	X	X	$*$

The current reading position (relative to the alignment), as indicated by $readPos$, is calculated from $left$, $rightStart$ and $rightEnd$.

When the text character specified by the current alignment and the relative reading position does not agree with the pattern, we shift with the smallest possible value, such that in the new alignment the pattern agrees with *all* the previously read characters. A shift by the length of the pattern guarantees no overlap with previously read characters, thereby proving the existence of the minimum. In case of a character match, we extend the right block accordingly.

We handle the transitions between the possible block configurations with particular attention to the cases when we reach the end of the pattern or the left block.

For an efficient implementation, we build a state transition table, as discussed in Section 2-B, that only *implicitly* contains the information on the two blocks of already matched characters. In case of the previously examined pattern $CABAB$, Figure III shows the table. On the right, we also show the blocks of matching characters and the current reading position for reference.

We will refer to *Algorithm 2* using the above–described state transition table as 2BLOCK.

The state transition table S for 2BLOCK has less then tm^3 entries and can be computed in polynomial time. We propose improvements in Section 2-E to reduce the table size in order to help practical implementations. First, we will discuss the theoretical properties of 2BLOCK.

D. Correctness and Complexity

The behavior of *Algorithm 2* almost solely depends on the properties of the corresponding table S. For 2BLOCK The following properties are crucial:

A) The reads do not fragment the blocks, meaning that after the shift[2] two blocks will still be able to represent the set of still matching characters.

B) The shift amounts stored in the table are the minimal possible, considering the previously matched characters.

C) The set of characters the algorithm implicitly keeps track of (the two blocks in case of 2BLOCK), are exactly those characters of the text which were read and match the pattern assuming the current alignment.

Property A ensures that the state transition table for 2BLOCK is well defined; its validity follows from the description in Section 2-C. Property B is also true by definition; it ensures that the first match (if exists) is indeed found by the algorithm.

Case analysis based on the arrangement of the blocks can verify that Property C is preserved during the execution of the loop of *Algorithm 2* and that it trivially holds at the start of the algorithm.

Notice, that the classical Boyer-Moore algorithm does not have Property C, because it "forgets" previous matches which may still be in scope.

We also claim that 2BLOCK is linear in n. In fact, we know more: no character of the text is read twice. Note, that for the classical Boyer-Moore algorithm, the best theoretical result claims no more than three reads for any character of the text [9]. Some variants reduced the number of reads of any character of the text to at most two [7], [8].

Property C above is instrumental to establish our claim. According to this property, the set of previously read characters of the text which overlap the current pattern alignment are in the two blocks the algorithm implicitly keeps track of. Since we read a character at one end of the right block, it follows that no previously read character can be ever read again.

The above argument applies to any variant of *Algorithm 2* satisfying Property C. In order to establish this property for a particular state transition table, one may produce a witness consisting of the set of matching characters for every state. For 2BLOCK we approached the problem from the opposite direction: we derived the state transition table from the witnessing blocks of matching characters.

E. Preprocessing and Further Improvements

We generate the state transition table S of 2BLOCK by starting with the initial state corresponding to both blocks being empty and then we add the new states (corresponding to rows in Table III) one at a time as needed. The process of creating the new block configurations for any possible character read and then calculating the shift amount is quite straightforward. If the configuration has been created before, then it must be referenced, otherwise it is added. Finiteness guarantees termination of the process. The discussion of a detailed implementation is beyond the scope of this paper.

[2]Zero shift amount is also allowed.

Our tests confirmed that a large state transition table hurts the performance, especially when the cache memory is exhausted. The table generation itself may also be of significant cost.

Consider state 4 of the state transition table created for the pattern "CABAB" as shown in Table III. The left block consists of a single matched character; any shift moves this character out of scope. Keeping track of this left block only marginally increase the average shift amount. Of course, if we eliminate this state, then we also remove the assurance that every character is read at most once.

While inspecting larger tables, it is apparent that there are states with the reading position close to the beginning of the string with no characters matched ahead of that position. Consider the read of an"A" in the following example:

$$\begin{array}{cccccccccccc} B & C & A & C & C & . & . & * & . & . & . & . & . \\ B & C & A & C & C & A & C & C \end{array}$$

We shift by 5 and arrive to:

$$\begin{array}{cccccccccccccc} . & . & . & . & . & . & . & A & * & . & . & . & . \\ & & & & & B & C & A & C & C & A & C & C \end{array}$$

Assuming that the text is random, we expect the next shift to be 4, 3 or 0 with $1/3$ probability each. In comparison, a read at the end of the pattern, without assuming any other matching characters, yields shift amounts of 2, 7 and 0. Removing states like these may not only reduce the table size, but may also increase the expected shift amount. Note, that elimination of certain states as we suggest may also yield a state transition table which implicitly tracks the single right block only.

Finally, we notice that many states have the property that a large shift occurs in case of any subsequent mismatch. Typically, the remaining character positions each trigger the addition of a new state with one more matching position in the blocks. This scenario can be represented in a more compact manner by allowing the states to indicate that any further mismatch allows a large shift. The main loop of *Algorithm 3* shows how we accommodate these *smart* state tables. We acknowledge that the shift amount of

Algorithm 3 The Boyer-Moore string searching variant implemented with a smart state transition table.

```
Input: $m$ character long pattern $P$, $n$ character long text $T$.
Output: The first matching position if any.
 1: Calculate the state transition table $S$ from the pattern $P$.
 2: Set $align = 0$, $state = 1$ and $readPos = m - 1$.
 3: while $align < n - m$ do
 4:     Read character $T(align + readPos)$ into $c$.
 5:     $(shift, state, readPos, match, smart) = S[state, c]$.
 6:     if $match = true$ then
 7:         return $align$.
 8:     end if
 9:     if $smart = true$ AND remaining characters match then
10:         return $align$.
11:     end if
12:     $align = align + shift$.
13: end while
14: return $NIL$.
```

TABLE IV. The shift amount achieved by the various algorithms averaged from random executions.

	BM	2BLOCK	CUT	SMART	SCUT
10	3.86	4.35	4.48	3.47	3.80
20	5.26	6.98	7.27	5.73	6.18
30	5.37	9.51	9.80	7.69	8.08
40	6.73	11.64	12.14	9.67	10.46
50	6.74	14.01	14.25	11.89	12.39
100	9.22	24.08	24.34	20.73	21.47
150	10.49	33.52	33.82	29.43	30.52
200	11.51	42.42	42.38	37.01	37.97

TABLE V. The number of states used by the various algorithms averaged from random executions.

	BM	2BLOCK	CUT	SMART	SCUT
10	10	55	25	19	9
20	20	186	69	34	19
30	30	398	122	49	25
40	40	697	198	64	33
50	50	1096	275	88	38
100	100	4263	1126	159	80
150	150	9477	2370	233	121
200	200	16443	4145	305	160

a smart state is not optimal; it cannot be larger than the smallest possible shift considering all the characters.

3. Experiments

We implemented various versions of our algorithm using the ideas described in Sections 2-C and 2-E. 2BLOCK and CUT are based on *Algorithm 2* using a state transition table that implicitly tracks two blocks. SMART and SCUT are based on Algorithm 3 employing smart states, whenever any subsequent mismatch results in a shift not less than half the pattern length. CUT and SCUT avoids creating states with the rightmost matching character position in the left half of the pattern; the transition is made to the initial state instead. We have also included *Algorithm 1* (BM) in the experiments.

We run our experiments using randomly generated patterns and text over a four character alphabet. We slightly modified the algorithms to find all matches of the pattern.

Tables IV, V and VI summarize the results averaged from ten executions. We observe that the average shift amount achieved by the Boyer–Moore algorithm falls behind when compared to the new algorithms. Note, that CUT has improved both the shift amounts and the table size when compared to 2BLOCK. SMART and SCUT trades the shift amounts in exchange for the use of fewer states.

In practical implementations, the increased memory use caused by a large state table can substantially effect performance. Table VI shows execution times of the algorithms performing ten string searches consecutively. Note, that our implementations have not been aggressively optimized: the state table generation uses a naive algorithm and the table itself could be implemented using less memory. We did not fine tune the heuristics deciding the cutoff points for eliminating states in CUT, SMART and SCUT. However, the trend for each algorithm is indicative: the performance of 2BLOCK and CUT will suffer for long patterns. The exact point when SMART and SCUT overtakes them depends on implementation details, the exact heuristics chosen and machine specifics.

TABLE VI. Running times of the various algorithms (in seconds) on a text with 10^8 characters executing ten different searches.

	BM	2BLOCK	CUT	SMART	SCUT
10	3.62	3.18	3.32	4.12	3.90
20	3.04	2.58	2.66	3.00	2.93
30	3.18	2.56	2.58	2.81	2.78
40	2.74	2.38	2.28	2.46	2.38
50	2.71	2.46	2.21	2.30	2.27
100	2.50	8.57	2.64	2.13	2.04
150	2.38	52.14	9.05	1.59	1.60
200	2.33	172.53	30.28	1.28	1.29

4. Future Work

The new algorithms increase the average shift amounts as the pattern size increases. A state of the art, highly optimized implementation of the algorithms is the next logical step to pursue.

The improvements discussed in Section 2-E do not exhaust all the possibilities. Future research may consider the use of probability calculations when creating the state transition table with attention to the character distribution as well. The fact, that we track two blocks of previously matched characters may be revisited, as well as whether the left block has to be anchored to the beginning of the string.

Open theoretical questions include the complexity of the CUT, SMART, and the SCUT algorithms.

References

[1] R. A. Baeza-Yates. String searching algorithms revisited. *Lecture Notes in Computer Science*, 382:75–96, 1989.

[2] R. A. Baeza-Yates, C. Choffrut, and G. H. Gonnet. On boyer–moore automata. *Algorithmica*, 12:268–292, October 1994.

[3] R. A. Baeza-Yates and M. Regnier. Average running time of the boyer–moore–horspool algorithm. *Theoretical Computer Science*, 92:19–31, January 1992.

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

[5] D. Cantone and S. Faro. Fast–search: a new efficient variant of the boyer–moore string matching algorithm. *Lecture Notes in Computer Science*, 2647:47–58, 2003.

[6] C. Choffrut and Y. Haddad. String–matching with OBDDs. *Theoretical Computer Science*, 320:187–198, June 2004.

[7] L. Colussi. Fastest pattern matching in strings. *Journal of Algorithms*, 16:163–189, March 1994.

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

[9] A. M. Odlyzko J. L. Guibas. A new proof of the linearity of the boyer–moore string searching algorithm. *SIAM Journal on Computing*, 9:672–682, 1980.

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

[11] T. Lecroq. A variation on the boyer–moore algorithm. *Theoretical Computer Science*, 92:119–144, January 1992.

[12] M. E. Nebel. Fast string matching by using probabilities: On an optimal mismatch variant of horspool's algorithm. *Theoretical Computer Science*, 359:329–343, August 2006.

[13] R. Schaback. On the expected sublinearity of the boyer–moore algorithm. *SIAM Journal on Computing*, 17(4):648–658, 1988.

[14] J. Tarhio. A sublinear algorithm for two-dimensional string matching. *Pattern Recognition Letters*, 17:833–838, July 1996.