Fast Substring Matching

Andreas Klein*

1 **Abstract**

2 The substring matching problem occurs in several applications. Two
3 of the well-known solutions are the Knuth-Morris-Pratt algorithm (which
4 guarantees a good worst case bound) and the Boyer-Moore algorithm
5 (which has a better average running time). In this work we will introduce
6 the general class of left-to-right searching algorithms which contains these
7 and several other algorithms as special cases. We show how to find the
8 optimal algorithm from this class by solving a linear programming prob-
9 lem. In particular, the presented method provides a matching algorithm
10 which for large texts is significantly better than known procedures.

11 ## 1 Introduction

12 Pattern matching is one of the basic problems in computer science. For example,
13 it arises naturally in text-processing, language recognition, bio-computing or
14 image processing. In this article we study the important special case of substring
15 matching, i.e. find all occurrences of a given substring of length m in a text of
16 length n. Further we assume that n is large in comparison with m. Therefore,
17 the time for preprocessing the substring can be neglected. This assumption is a
18 natural one, since the size of the text can be several gigabytes (internet research
19 or e-books) but the size of the substring does not exceed ten letters in most

*Ghent University, Dept. of Pure Mathematics and Computer Algebra, Krijgslaan 281-S22, 9000 Ghent, Belgium

20 cases. Additionally we assume that the relative frequency of the letters in the
21 text is known in advance. In most applications this assumption is evident, for
22 example, in Internet research and e-books the language of the text and therefore
23 the frequency of the letters are known. Another example in bio-computing
24 is the typical distribution of triples of DNA-bases. In all these examples the
25 distribution differs significantly from the equality distribution so we can archive
26 an improvement if we use a specialized algorithm that benefits from the different
27 probabilities.

28 This paper is organized as follows. First we give an overview of some of the
29 previously known algorithms. In section 2 we will introduce the class of the
30 left-to-right searching algorithms and demonstrate that the previously known
31 algorithms do belong to this class. In section 3 we show how to find the best
32 algorithm of this class. Finally, we compare the new optimal algorithm with
33 previously known algorithms to give an impression of the improvement.

34 The naive substring searching compares every letter of the pattern with every
35 letter of the text. This yields to the worst case bound of $O(nm)$ comparisons. A
36 natural improvement is to use a deterministic finite automaton for the pattern
37 matching which gives the worst case bound of $O(n)$ comparisons. The Knuth-
38 Morris-Pratt algorithm [5] uses the special structure of the pattern to construct
39 the automaton in only $O(m|\mathcal{A}|)$ steps. The Boyer-Moore algorithm [2] uses
40 another strategy. As the naive algorithm it compares every letter of the pattern
41 with every letter of text. But it starts with the letter at the right end of the
42 substring. If a mismatch occurs it uses two rules known as mismatch heuristic
43 and occurrence heuristic to move the search window as far as possible to the
44 right. The worst case bound remains by $O(nm)$, but under some reasonable
45 assumptions the average running time is as low as $O(n/m)$. Actually, in practice
46 the Boyer-Moore algorithm or a close variant is used in most substring matching

47 routines.

48 The Boyer-Moore-Horspool algorithm [4] differs from the Boyer-Moore algo-
49 rithm only in ignoring the match heuristic. This reduces the preprocessing time
50 without increasing the running time too much. It is also possible to combine
51 the ideas from the Boyer-Moor algorithm with ideas of the Knuth-Morris-Pratt
52 algorithm. This is for example done in the Turbo-BM algorithm described in
53 [3]. In a recent work M. Nebel [6] shows that the probability of the application
54 of the match heuristic can be increased by examining the most improbable let-
55 ters of the pattern first. If different letters have different probabilities this can
56 significantly reduce the average running time.

57 A detailed survey over these and other matching algorithms can be found in
58 [1].

59 Now we want to describe a general class of algorithms that contains in par-
60 ticular all of the above algorithms. The task is to identify the fastest algorithm.

61 2 The general algorithm

62 The algorithm has a search window that moves from left to right. If the search
63 window is at position i the algorithm tests if the substring starting at the i-
64 th position matches the pattern. Especially this procedure assures that the
65 matchings are found in order from left to right. Thus we speak of the class of
66 left-to-right searching algorithms. The algorithm has an internal state in which
67 the part of the pattern that has already been tested is marked.

68 In each step the algorithm chooses a letter which has not been tested so far at
69 random. (The probability distribution for the random decision will be specified
70 later.) If the letter matches the pattern, the algorithm changes its state to mark
71 that it has been tested. If the letter does not match, the pattern the algorithm
72 moves its search window as far as possible to the right. At this point it uses a

variation of the match and occurrence heuristic of the Boyer-Moore algorithm. If possible, already tested letters are stored in the new state.

So the algorithm need up to $2^m - 1$ internal stages. For each state we have to store the information which letter has to tested next. Therefore the algorithm needs a storage that is exponential in m. Since m is small, e.g. for $m \approx 10$ a few k-bytes will be sufficient, this is acceptable.

Let us have a look at a simple example. We search for the pattern 10 in a text which consists solely of the letters 0, 1 and 2. Assume that the letters in the text are independent and identically distributed with the probabilities $p(0) = a$, $p(1) = b$ and $p(2) = c$ (obviously we have $a + b + c = 1$). Our algorithm can be in one of the following three states.

- In the state ?? we have no information about the letters in question. In algorithm can choose to test the first letter with probability h_1. (Shown by dashed lines in Figure 1). As a second possibility the algorithm can choose with probability $h_2 = 1 - h_1$ to test the second letter. (If we choose $h_1 = 1$ and $h_2 = 0$ we obtain the Knuth-Morris-Pratt algorithm.)
- In the state X? we know that the first letter matches the pattern, i.e. it is 1.
- In the state ?X we know that the second letter is 0.

Thus we can describe the general matching algorithm for the pattern 10 with the following state-graph.

Now we give a formal description of the algorithm in pseudo-code and show how to compute the next state and the step width of the search window.

The internal state of the algorithm can be fully described by the position P of the search window and the letters of the substring we have tested so far. We encode a tested letter as 1 and an untested letter as 0. By this encoding we

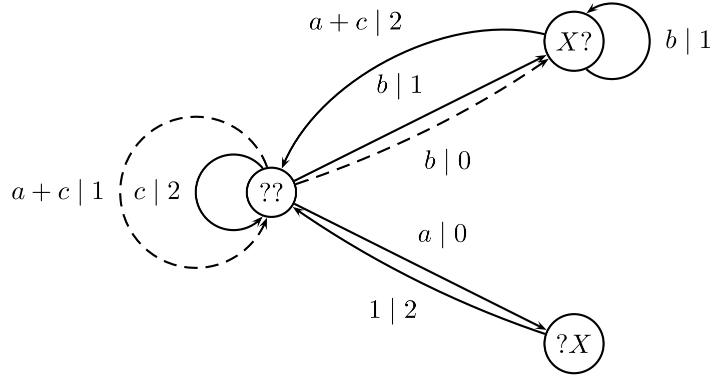

Figure 1: The search graph for the pattern 10

99 map each state to a (binary) number between 0 and $2^m - 2$. (In Figure 1 we
100 number the state ?? by 0, the state $?X$ by 1 and the state $X?$ by 2. Note that
101 we need no state labeled by XX, since if we find a matching, we will print a
102 success message and then move our search window to the right reaching state
103 ??.)

Algorithm 1 left to right searching algorithm

1: $P = j$, $S = 0$ {initialize}
2: **while** $P \leq n - m$ **do**
3: {loop until the search window reach the end of the text}
4: choose J {we will later show, that in the optimal algorithm $J = J(S)$ and compute the function $J(S)$}
5: **if** Pattern[J] = Text[P+J] **then**
6: set the j-th Bit of S to 1 {pattern matches}
7: **if** $S = 2^m - 1$ **then**
8: **print** Matching found at position P
9: Increase P and reset S {see discussion below}
10: **end if**
11: **else**
12: Increase P and change S according to Algorithm 2. {Precomputed values}
13: **end if**
14: **end while**

104 In line 9 we have to determine the largest prefix of the pattern that is also
105 a suffix. If the length of that prefix is i we have to increase J by i and set S to

106 2^{i-1}. (In the example of the pattern 10 there is no such prefix, i.e. $i = 0$ and
107 we will move the search window by 2 after each successful match.)

108 We now describe how to adopt the mismatch heuristic and occurrence heuris-
109 tic to our general case.

Algorithm 2 How to shift the search window

1: **INPUTS:** a state S, a mismatch $X = P[P+J]$ at position J {We may run the algorithm for all possible inputs to build a lookup table}
2: $i = 0$
3: increase i by one
4: **if** $j - i \geq 0$ and $Pattern[J - i] \neq X$ **then**
5:   **goto** line 3 {Mismatch heuristic}
6: **end if**
7: **for all** k **do**
8:   **if** the k-th Bit of S equals 1, $k - i \geq 0$ and $Pattern[k - i] \neq Pattern[k]$ **then**
9:     **goto** line 3 {Mismatch heuristic}
10:   **end if**
11: **end for**
12: **print** increase P by i {move the search window}
13: set the j-th Bit of S to 1
14: dived S by 2^i (integer division)
15: **print** switch to new state S

110 Algorithm 2 simply does the following. It shifts the search window by 1 and
111 compares if the letters we have already seen equal the letters we expect to find.

112 Note that we can do the computations of Algorithm 2 in the preprocessing
113 phase, i.e. Algorithm 2 gives no contribution to the running time. Therefore, if
114 X does not occur in the pattern the exact value of X does matter for Algorithm
115 2. Since a typical pattern will contain only a few letters of the alphabet this
116 means that the preprocessing is independent from the alphabet size.

117 Now we demonstrate that Algorithm 1 contains many other pattern match-
118 ing algorithms as a special case.

119 Choosing (in step 4) always the first untested letter we obtain exactly the
120 Knuth-Morris-Pratt algorithm. If we choose the last untested letter we obtain

121 a variation of the Boyer-Moore algorithm. To obtain the exact Boyer-Moor
122 algorithm we have to forget after each "shift operation" the information about
123 all tested letters, i.e. we must add some unnecessary comparisons. This shows
124 that we can look at the Knuth-Morris-Pratt algorithm and the Boyer-Moore
125 Algorithm as a special case of a left-to-right searching algorithm.

126 Another algorithm we can find in this class is the algorithm due to M. Nebel
127 cited in the introduction.

128 The goal of this paper is to show a method to determine (in dependency
129 of the known probability distribution on the alphabet) the optimal substring
130 matching algorithm in the class described above.

131 As we will see in the next section the optimal algorithm is a deterministic
132 one, i.e. the randomized algorithms allowed in our general description obtain no
133 advantage. (However they are needed as a technical part of the proof.)

134 It should be noted that there are possible situations in which other algo-
135 rithms (for example an algorithm that searches from the right to the left) can
136 be faster than all algorithms in the class described above. Another one that
137 is not in our class is the variant of the Boyer-Moore algorithm described by
138 D. Sunday [9]. This variant tests the leftmost letter after every mismatch and
139 then continues to test the letters from right to left as the original Boyer-Moore
140 algorithm. This is not a special case of our class since we require that each action
141 of the automaton depends only on the current state and not on the preceding
142 action. In section 5 we will demonstrate how to integrate such an extension in
143 our framework.

144 3 The optimal algorithm

145 The general algorithm from section 2 can be described as a stochastic finite
146 automaton. Each step is joined by an award (the number of letters the search

147 window moves to the right). We are looking for the automaton that maximizes
148 the expected award per step for a random input.

149 We use the encoding of the stages by (binary) numbers between 0 and $2^m - 2$
150 described in the last section.

Let $p_{i,t}$ denote the probability for the automaton to be in the state i at time step t $(0 \le i \le 2^m - 2)$. The theory of Markov chains tells us that the limit

$$p_i = \lim_{n \to \infty} \frac{1}{n} \sum_{t=1}^{n} p_{i,t}$$

151 exists. The probability p_i is called the limit probability and describes how often
152 the automaton will be in the state i. (In this article we need only basic results
153 from the theory of Markov chains, that can be found in many textbooks for
154 example see [8].)

155 The probability that the randomized algorithm tests the letter at position
156 j if it is in the state i is denoted by $h_{i,j}$. Necessarily we have $h_{i,j} \ge 0$ and
157 $\sum_{j=1}^{m} h_{i,j} = 1$. (In the example we have $h_{1,1} = h_{2,2} = 1$ since in these states
158 there is only one possibility left. The probabilities $h_{0,1}$ and $h_{0,2}$ can be chosen
159 freely and our goal in this section is to determine the best decision.)

160 With $a_{i,j,k}$ we denote the probability for the algorithm to switch into the
161 successor state k, if it is in the state i and tests the letter at position j. The
162 probabilities $a_{i,j,k}$ depend only on the known probabilities of the different letters
163 in the text. In Figure 1 we have written these probabilities on the edges of the
164 graph (the labeling in the first position). For example $a_{0,2,0} = c$ since the only
165 way for the algorithm to obtain no information on the pattern is to read a 2
166 (and then move the search window two steps to the right). $a_{0,2,2} = b$ since the
167 algorithm tests the second letter, it changes its state from ?? to X? only if it
168 reads a 1 (and moves its search window one letter to the right). To compute
169 $a_{i,j,k}$ we simply have to sum the probabilities of all letters X for which Algorithm

170 2 computes the successor state k if the inputs are s, X and j.

171 By $b_{i,j,k}$ we denote the (average) number of letters the search window ad-
172 vances from the left to the right, if the algorithm switches from state i to state
173 k while testing the letter j. In Figure 1 these numbers are shown as the second
174 edge label, i.e. $b_{0,1,0} = 2$ and $b_{0,1,2} = 1$. To compute $b_{i,j,k}$ we sum up $p(X)i(X)$
175 over all letters X for which Algorithm 2 computes the successor state k if the
176 inputs are s, X and j. Here $p(X)$ denotes the probability of X and $i(X)$ is the
177 value, which is computed as step width in Algorithm 2.

178 All these computations can be done in the preprocessing phase and extending
179 Algorithm 2 to compute this values cost almost no extra time.

Now we may apply the theory of Markov chains to conclude that the limit probabilities p_i are given by the following system of equations.

$$p_k = \sum_{i=0}^{2^m-2} \sum_{j=1}^{m} p_i h_{i,j} a_{i,j,k} \tag{1}$$

To determine the optimal algorithm we have to maximize the expected step width for the search window, i.e. we have to solve the following optimization problem.

$$\text{Maximize:} \quad \sum_{i=0}^{2^m-2} \sum_{j=1}^{m} \sum_{k=0}^{2^m-2} p_i h_{i,j} a_{i,j,k} b_{i,j,k}$$

under the restrictions given by equation (1) ($k \in \{0, \ldots, 2^m - 2\}$) and

$$\sum_{j=1}^{m} h_{i,j} = 1 \quad \text{for} \quad i \in \{0, \ldots, 2^m - 2\}$$

180 and the sign conditions $p_i \geq 0$ and $h_{i,j} \geq 0$.

This a nonlinear optimization problem but we can transform it into a linear one by the following trick. We introduce new variables $w_{i,j} = p_i h_{i,j}$. Since $\sum_{j=1}^{m} h_{i,j} = 1$ we have $p_i = \sum_{j=1}^{m} w_{i,j}$ and $h_{i,j} = \frac{w_{i,j}}{p_i} = \frac{w_{i,j}}{\sum_{j=1}^{m} w_{i,j}}$. Thus we can

reconstruct the variables p_i and $h_{i,j}$ from the new variables $w_{i,j}$. With these new variables we may write (1) in an equivalent form as

$$\sum_{j=1}^{m} w_{k,j} = \sum_{i=0}^{2^m-2} \sum_{j=1}^{m} w_{i,j} a_{i,j,k} \tag{2}$$

This yields the following linear program.

$$\text{Maximize: } Z = \sum_{i=0}^{2^m-2} \sum_{j=1}^{m} \sum_{k=0}^{2^m-2} w_{i,j} a_{i,j,k} b_{i,j,k}$$

181 under the restrictions given by equation (2) ($k \in \{0, \dots, 2^m - 2\}$) and the sign
182 conditions $w_{i,j} \geq 0$.

183 Since the linear programming problem belongs to the complexity class $\mathcal{P}$
184 (see for example [7]) we have proven the following theorem.

Theorem 1

185 *One can determine the optimal algorithm in the class of left-to-right searching*
186 *algorithms with a preprocessing in* $\mathcal{EXL}(m)$ *(exponential time). The algorithm*
187 *itself needs only* n/Z *comparisons in the average case. (*Z *is the maximum*
188 *determined by the optimization problem.)*

189 The representation as linear programming problem shows why we allow ran-
190 domized algorithms in our class of the left-to-right searching algorithms. If we
191 require non-randomized algorithms we have to add additional nonlinear restric-
192 tions. In other words, we would obtain a non-linear optimization problem. But
193 we can show that there always exists a non-randomized algorithm with minimal
194 expected running time.

Theorem 2

195 *There exists a non randomized algorithm with an expected running time of* n/Z
196 *steps.*

197 **Proof**

198 Suppose that some of the probabilities p_i are equal to 0 in the optimal algorithm.
199 (In the example of Figure 1 this could happen if the optimal algorithm tests
200 the first letter if it is in state ??. In this case it will never reach the state
201 $?X$, i.e. $p_1 = 0$.) Let $\mathcal{S}$ be the set of all i with $p_i = 0$ in the optimal solution.
202 Furthermore we assume that there exists no other optimal solution that satisfies
203 $p_i = 0$ for all $i \in \mathcal{S}$ and $p_j = 0$ for at least one $j \notin \mathcal{S}$. In other words, we assume
204 that $\mathcal{S}$ is maximal.

In this case that optimal solution is also an optimal solution of the following linear program.

$$\text{Maximize: } Z = \sum_{i \in M} \sum_{j \in \{0,\ldots,2^m-2\} \setminus \mathcal{S}} \sum_{k=0}^{2^m-2} w_{i,j} a_{i,j,k} b_{i,j,k}$$

under

$$\sum_{j=1}^{m} w_{k,j} = \sum_{i \in \{0,\ldots,2^m-2\} \setminus \mathcal{S}} \sum_{j=1}^{m} w_{i,j} a_{i,j,k} \qquad \text{for } k \in \{0,\ldots,2^m-2\} \setminus \mathcal{S}$$

205 and the sign conditions $w_{i,j} \geq 0$.

As it is known each linear program has an optimal basic solution. For the linear program described above a basic solution has at most $(2^m - 1) - |\mathcal{S}|$ variables that are nonzero. On the other hand

$$0 < p_i = \sum_{j=1}^{m} w_{i,j}$$

206 for all $i \in \{0,\ldots,2^m-2\} \setminus \mathcal{S}$ (by definition of $\mathcal{S}$). This proves that for every
207 $i \in \{0,\ldots,2^m-2\} \setminus \mathcal{S}$ exactly one $w_{i,j}$ in the basic solution is nonzero, i.e.
208 exactly one $h_{i,j} = 1$ and all other $h_{i,j'} = 0$. This shows that the algorithm
209 described by the optimal basic solution is non-randomized.

4 Comparison with other algorithms

Now we compare the algorithm with previously known algorithms. First we prove a theorem about the worst case bound.

Theorem 3

Each left-to-right searching algorithm needs at most n comparisons.

Proof

By definition a left-to-right searching algorithm memorizes all letters it has already tested, i.e. if it has seen a letter of the text it will remember this letter as long as it is located in the search window. In other words, it has to read each letter at most once which bounds the number of comparisons to n.

(It should be noted that this worst case bound cannot be improved, as we can see if the pattern consists of just one letter.)

Theorem 3 proves that each left-to-right searching algorithm satisfies the same worst case bound as the Knuth-Morris-Pratt algorithm. But for the Knuth-Morris-Pratt Algorithm the worst case complexity is identically to average case complexity, while all other left-to-right searching algorithms have an average running time less or equal n. In other words the Knuth-Morris-Pratt algorithm can never be optimal in the class of left-to-right searching algorithms.

In [6] M.Nebel proves that his algorithm is faster than the Boyer-Moore algorithm in many cases but that there exist cases in which the Boyer-Moore algorithm is faster. Especially non of these algorithms is optimal.

Now we take a closer look on the strategy "test the most improbable letter first" suggested by M. Nebel [6] and an alternative strategy "maximize the local step width". Both strategies have the advantage that they require less preprocessing time, but we will see that they will be not as good as the general algorithm. Suppose we want to search the pattern 123 in a text which contains only the letters $1, 2$ and 3. Furthermore suppose $p(1) \approx 0$, $p(2) \approx \frac{2}{3}$ and

$p(3) \approx \frac{1}{3}$. Which letter should be tested if the automaton is in the state ??, i.e. it has no knowledge on the text. The "test the most improbable letter first" and the "maximize the local step width" strategy will recommend the first letter (with an average step width $p(2) + p(3) \approx 1$ compared with a step width of $2p(3) + p(1) \approx \frac{2}{3}$ (if we test the second letter) or of $p(2) + 2p(1) \approx \frac{2}{3}$ (if we test the third letter). Most probably after this test the automaton will be in the state ?? again and therefore the average step width will be ≈ 1. We can increase the average step width to ≈ 1.5 if we first test the letter 3 and then the letter 1. This procedure will most probably shift the search window by 3 letters every 2 steps.

This shows that finding the optimal algorithm is not obvious and that the optimal algorithm can be significantly faster than an algorithm based on a simple heuristic. But to find the optimal algorithm we need a very expensive preprocessing. For $m \approx 10$ this is no problem since the preprocessing is fast enough to be negligible. For larger m we can choose between the following suboptimal strategies.

- The size of the linear program does not depend on the size of the alphabet $\mathcal{A}$. Therefore we may interpret two signs of the original alphabet as one sign of a new larger alphabet and search the optimal algorithm for this alphabet. This reduces the search space (we require that two letters at adjacent positions must be tested at the same time), i.e. the new search algorithm may be non-optimal. But this restricted class still contains the Knuth-Morris-Pratt algorithm and the Boyer-Moore algorithm, so we can be sure to get an improvement of these algorithms.

- First we search for a sub-pattern of length ≈ 10. This sub-pattern will occur on a few positions only. We will then use the naive algorithm to check this few positions. If we choose the sub-pattern of the last 10 letters

we can still be sure to get an improvement of the Boyer-Moore algorithm.

We conclude this section with a larger example. Assume we want to match the pattern 123 in a text over the alphabet $\{1, 2, 3\}$. Let $p(1) = a$, $p(2) = b$ and $p(3) = b$. In the state 0 we may test the letter at the first, second or third position (3 choices), in the states number by 1, 2 and 4 we have two possible positions to check. In a total we have the choice between $3 \cdot 2 \cdot 2 \cdot 2 = 24$ possible algorithms.

Diagram 2 shows for that different algorithms are optimal for different choices of a, b and c. Since $a + b + c = 1$ we can represent all possible values in a plane. The vertices of the triangle representing the three extreme cases $a = 1$, $b = 1$ or $c = 1$, respectively. The center is cases $a = b = c = \frac{1}{3}$ and so on. Since the regions have a very complex shape, the picture is only drawn as a raster with width 0.05 for a, b and c.

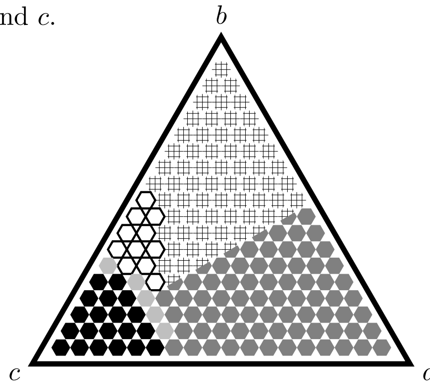

Figure 2: Optimal Algorithms for the pattern 123

There are 5 different kind of optimal algorithms. The three main cases are:

If $c \approx 1$ (black area) we should test in state 0 for the 2 (second letter) and for the first possible letter in all other states. If $a \approx 1$ (gray area) we should always test for the right most letter (Boyer-Moore). For $b \approx 1$ (hatched area) we should test for the fist letter in the state 0 but follow in all other states the Boyer-Moore strategy (this is similar to an algorithm suggested by Sunday).

The two other algorithms are only optimal for a small set of parameters: If $a \approx 0$, $b \approx 0.4$, $c \approx 0.6$ (the area marked by white hexagons) we should test the right most letter if we are in the state 0 or 2 and the left most letter in the states 1 and 3. Finally we have a very small area (light gray) where the following algorithm is optimal. Test in state 0 test the letter in the middle, in state 2 test the right most letter, and in state 4 test the left most letter

Again we see that is fare form obvious to decide which algorithm is the fastest.

5 Possible extensions

As noted in section 2, the Sunday's algorithm is not in the class of left-to-right searching algorithms. Sunday's algorithm is a variation of the Boyer-Moore algorithm that tests the first letter after each mismatch and then continues to test the letters from the right to the left. A simple way to model Sunday's algorithm as a left-to-right searching algorithm is to test the first letter in the state 0 and the right most letter in all other states. This yields a good but not perfect approximation of Sunday's algorithm. In a perfect model we should test the first letter after each shift of the search window. But this is not a left-to-right searching algorithm, since the decision which letter is to test next depends not only on the current state but also on the last action. To model such algorithms we can extend our class of left-to-right searching algorithms in the following way. Instead of $2^m - 1$ states (numbered by 0 to $2^m - 2$) we use $2(2^m - 1)$ states (labeled by (i, j) with $0 \leq i \leq 2^m - 2$ and $j \in \{M, S\}$). The marker j shows us if the last operation was a match (M) or a mismatch resulting in a shift (S). We now proceed as in Section 3. We write a linear program that describes the optimal algorithm and solve this program to find the optimal algorithm. For this algorithm we can be sure that it is an improvement of the simple optimal

308 left-to-right searching algorithm described previously in this paper and Sunday's
309 algorithm. However we have doubled the number of equations and variables in
310 the linear program, i.e. we need more time for preprocessing.

311 There are several other possible extensions of the class of left-to-right search-
312 ing algorithms. For example, the algorithm may decide the next step on the
313 two proceeding states (modeled by an automaton with $(2^m - 1)^2$ states). But it
314 seems that the additional effort to determine the optimum in such larger classes
315 is not worth the improvement.

316 An other variation of left-to-right searching algorithms is the following, after
317 each shift we forget the tested positions, i.e. we start always in the state 0. This
318 has the advantage that the final algorithmen, can described by the order in
319 which it should test the characters of the pattern, the description therefor just
320 a permutation of the number 1 to m. The disadvantage is that the optimization
321 problem we have to keep track over which positions are untested, which positions
322 have been tested but forgotten and positions are tested. We need 3^m therefore
323 variables in the preprocessing phase. Thus there is no hope to find the optimal
324 algorithm without an expensive preprocessing.

325 6 Conclusion

326 We described a general class of substring matching algorithms which contains
327 many of the popular matching algorithms. We have shown that it is possible
328 to determine the optimal algorithm in that class by solving a linear program-
329 ming problem. The determination of the optimal algorithm needs an expensive
330 preprocessing, but for large texts and small patterns the speedup will compen-
331 sate for the preprocessing. Furthermore, we have shown how to reduce the
332 preprocessing for larger patterns but still improve the known algorithms.

References

[1] A.V. Aho. Algorithms for finding patterns in strings. In J. van Leeuven, editor, *Handbook of Theoretical Computer Science*, chapter 5, pages 255–300. Elsecier Science Publishers, 1990.

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

[3] M. Crochemore, A. Czumaj, S. Jarominek L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. Speeding Up Two String Matching Algorithmens. *Algorithmica*, 12(4-5):247–267, 1994.

[4] R.N. Horspool. Practical fast searching in strings. *Software-Practice and Experience*, 10(6):501–506, 1980.

[5] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[6] M.E. Nebel. Fast String Matching by Using Probabilities. Technical report, Johann Wolfgang Goethe-Universität, 2004. Draft extended abstract of the material presented at the 10th Seminar on the Analysis of Algorithms, June 2004, MSRI, Berkeley.

[7] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1982.

[8] M. Rosenblatt. *Markov processes, structure and asymptotic behavior.* Grundlehren der mathematischen Wissenschaften. Springer, Berlin, 1971.

[9] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.