On Boyer-Moore Automata

*Ricardo A. Baeza-Yates**
Dpto. de Ciencias de la Computación
Universidad de Chile, Casilla 2777
Santiago, Chile

Christian Choffrut†
Laboratoire d'Informatique Théorique
et de Programmation, Université Paris 7,
2 Pl. Jussieu, Paris 75 251 Cedex 05, France

Gaston H. Gonnet
Informatik, ETH Zürich
Switzerland

Abstract

The notion of Boyer-Moore automaton was introduced by Knuth, Morris and Pratt in their historical paper on fast pattern matching. It leads to an algorithm that requires more preprocessing but is more efficient than the original Boyer-Moore's algorithm. We formalize the notion of Boyer-Moore automaton, and we give an efficient building algorithm. Also, bounds on the number of states are presented, and the concept of potential of a transition is introduced to improve the worst and average case behavior of these machines. We show that looking at the rightmost unknown character, as suggested by Knuth *et al.*, is not necessarily optimal.

Keywords: string searching, pattern matching, finite automaton, average case analysis.

1 Introduction

String searching is a very important component of many problems, including text editing, data retrieval and letter manipulation. Formally, the string searching or string matching problem consists in finding all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over some alphabet. It is well known that to search for a pattern of length m in a text of length n (where $n \geq m$) the search time is $O(n+m)$ in the worst case [KMP77]. Moreover, in the worst case at least $n - m + 1$ characters must be inspected [Riv77].

However, Boyer and Moore [BM77] improved drastically the average case by searching in the pattern from *right to left*. Given a text and a pattern, Boyer-Moore's (BM) algorithm for finding an occurrence of the pattern in the text consists in sliding the pattern along the text from left to right and attempting to match it from right to left with the portion of the text currently covered. In case of a mismatch a heuristic is given as to by how much the pattern has to be shifted to the right. Once in its new position the same procedure is applied but no memory is kept of the possible

*The author gratefully acknowledges the support of Grant C-11001 from Fundación Andes.
†The author gratefully acknowledges the support of the PRC Mathématiques et Informatique.

partial matches obtained in the previous steps. For example if the mismatch occurs in the second rightmost position of the pattern - say this is the i-th character of the text - and if the heuristic leads to shifting the pattern one position to the right, then the information that the $(i+1)$-th character of the text equals the rightmost character of the pattern is lost. Nevertheless, with this algorithm, searching is faster on average, because in many cases the shift is proportional to the length of the pattern, and in the best case only $O(n/m)$ comparisons are needed.

Knuth, Morris and Pratt [KMP77] showed that an improved version of the algorithm was linear in the worst case (see also [GO80]). Later, Galil [Gal79] and Apostolico and Giancarlo [AG86] made further improvements, such that by having only m states, a $2n - m + 1$ worst case number of comparisons is achieved, as in the Knuth-Morris-Pratt (KMP) algorithm [KMP77]. The same number of comparisons is achieved by a hybrid algorithm [BY89b] between the KMP algorithm and Horspool's variant [Hor80] of the BM algorithm.

In [KMP77] it is suggested to keep track of all information gathered while searching the text. At any given point some (possibly none) of the last m current characters of the text (where m is the length of the pattern) are known and the algorithm consists of inquiring about the unknown letters only. This leads to defining a finite automaton where each state carries partial information on the text, which we call *Boyer-Moore automaton* (BMA). We formalize this class of machines, and we present several improvements for both the worst and the average case behavior, using a novel concept of transition *potential*. In particular, we show that looking at the rightmost character (as suggested in [KMP77]) is not always optimal, and we propose several local optimization techniques. We also present an average case analysis of these automata, and several variations with a bounded number of states, that appear to be practical.

In [Gal79] it is argued that the benefit of remembering the characters of the text that have been successfully matched against the pattern is outweighted by the cost of the construction of the automaton. Only a careful study of the complexity of the automaton could definitely settle the matter. This can be divided into two tasks: determining the cost per state of the transition function and evaluating the size of the automaton, that is, the number of states as a function of the length of the pattern, m. Here we propose an efficient construction algorithm, which needs $O(m^2|Q|)$ time and $O(m|Q|)$ space, being $|Q|$ the number of the states of the automaton. Unfortunately the second question on the size of the automaton remains open. We also tackle this problem for which no upper bound different from the straightforward 2^m is known. We give a polynomial upper bound under the assumption that the number of occurrences of each letter is fixed (of course the number of letters is arbitrary). On the other hand, we give a class of patterns which has $O(m^3)$ states for a binary alphabet. This improves a previous result by Guibas and Odlyzko of $O(m^3)$ for a ternary alphabet, mentioned in [Gal85] as a private communication.

Some of the results of this paper were presented in a preliminary form in [BYGR90] and [Cho90].

2 Preliminaries

In the sequel $\Sigma = \{a, b, ...\}$ is a finite alphabet and Σ^* is the set of all words or finite sequences of elements in Σ. The word of length 0 is the empty word and is denoted by ϵ. We denote by Σ^+ the set of all nonempty words. A special letter $\#$, not belonging to Σ, may be viewed as a "don't know" letter. It is convenient to envision a word w of length $m = |w| \geq 0$ as a function of the

integer interval $[1..m]$ into the set Σ. This enables us to denote by $w[i]$ the i-th character of w. Thus, if $w = aab\#cb$ then $m = 6$, $w[1] = w[2] = a$, $w[3] = w[6] = b$, $w[5] = c$ and $w[4] = \#$. Given the words w, w_1, w_2, w_3 with $w = w_1w_2w_3$ we say that w_1 (resp. w_2, w_3) is a *prefix* (resp. *subword, suffix*) of w. A suffix is *proper* if it is different from the word itself. An occurrence is a subword along with its position in the word. For example *bababa* has two occurrences of *aba*. A word has a *border* if it has a nonempty proper prefix equal to its suffix of the same length. Thus $w = aaba$ has a border but $w = aab$ has not. We use w^i to denote the word w concatenated i times with itself.

Let $w \in \Sigma^*$ be a pattern of length m and let $t \in \Sigma^*$ be a text. Searching w in t consists in determining whether or not t contains an occurrence of w, that is, whether or not $t = t_1wt_2$ holds for some t_1 and t_2. Numerous algorithms have been designed for solving this problem. Here we use an extension of Boyer-Moore's algorithm proposed in [KMP].

Given a pattern $w \in \Sigma^*$ ($|w| = m$), by the Boyer-Moore automaton associated with w we mean the following deterministic finite automaton (Q, Σ, δ, q_0):

1. A state $q \in Q$ is a word of length m satisfying for all $0 < i \leq m$, $q[i] = w[i]$ or $q[i] = \#$, which can be viewed as a partial information on the text. Thus, if $w = abbabab$, then $a\#\#ab\#b$, $a\#bab\#b$, $abbab\#b$ are states of increasing information. A state records among the last m characters of the text those that have been successfully tested against the pattern, that is, whose values are known. The letter # is a "don't know" character and must be interpreted as a lack of information on the actual value of the corresponding position. The first state of the above example conveys the information that the leftmost character is an a, the second leftmost character is unknown and so is the third leftmost character, etc.

   In Knuth *et al.* definition the next character to be read is the text position corresponding to the rightmost # in q. That is the standard BMA. We extend this definition by generalizing the position of the next character to be read. Hence, we associate with each state the next position $0 < P(q) \leq m$ of the pattern to be compared, which implies what character of the text must be read. $P(q)$ must be one of the "don't know" characters in q. In this case, we have an *extended BMA*.

2. q_0 is the initial state, that is, $q_0 = \#^m$ (at the beginning of the computation no information what so ever is known on the text).

3. $\delta : Q \times \Sigma \to Q$ is the transition function that associates with $q \in Q$ ($q \neq w$) and $a \in \Sigma$ the next state $q' = \delta(q, a)$ defined as follows. Consider the occurrence of # in q at position $i = P(q)$, that is, $q = q_1q[i]q_2$ where $q[i] = \#$, and $q_1, q_2 \in (\Sigma \cup \{\#\})^*$. Let r be $q_1w[i]q_2$.

   If $a = w[i]$ then the new information is consistent with the rest and we set $q' = r$. Otherwise, there is a mismatch and the pattern must be shifted. Consider the shortest shift that is consistent with the information obtained so far on the text. Formally, let s be the smallest integer j satisfying the condition:

   (2.1) For all $j < k \leq m$, if $r[k] \neq \#$ then $r[k] = w[k - j]$.

   Then $q' = \delta(q, a)$ is the word defined by:

   (2.2) For all $0 < k \leq m - s$, $q'[k] = r[k + s]$ holds and for all $m - s < k \leq m$, $q'[k] = \#$ holds.

Figure 1 shows an example. With each transition we associate the shortest shift $s \in [0..m]$, and a logical variable $bool \in (true, false)$ that indicates if there was a match or not (automaton output). We use δ_{bool}, and δ_s to refer to the above values associated with $\delta(q, a)$. The transitions for the pattern aab are shown in Table 1.

state	reading position	output, shift, and next state $(\delta_{bool}, \delta_s, \delta)$		
q	$P(q)$	a	b	x
$q_0 = \#\#\#$	3	$(f,1,1)$	$(f,0,2)$	$(f,3,0)$
$q_1 = \#a\#$	3	$(f,1,3)$	$(f,0,4)$	$(f,3,0)$
$q_2 = \#\#b$	2	$(f,0,4)$	$(f,3,0)$	$(f,3,0)$
$q_3 = a\ a\#$	3	$(f,1,3)$	$(t,3,0)$	$(f,3,0)$
$q_4 = \#a\ b$	1	$(t,3,0)$	$(f,3,0)$	$(f,3,0)$

Table 1: Transitions for the pattern aab (t and f denote true and false).

There are no final states, because we want to find all occurrences of the pattern in the text. In practice, we can suppose that there is an implicit final state that is reached from every state after reading a special *end of text* letter. To simplify the exposition, we use the letter $x \notin \Sigma$ to denote any letter in the alphabet that is not present in the pattern.

Definition: The *main chain* or *skeleton* is the shortest sequence of states from the initial state q_0 to the matching transition. In other words, all states corresponding to a match of any proper suffix of the pattern. Any successful sequence has a nonempty intersection with the skeleton, and so these states are absolutely necessary if we want to find all occurrences of the pattern. ■

Example: Let $w = aab$ be the pattern. The transition function δ was already shown in Table 1. Figure 2 depicts this automaton, where in each transition the associated letter(s) and shift are given. Additionally, a ! indicates that a match was found after reading the current letter. Each state shows the information recorded and a number, $P(q)$. The skeleton is the set of states 0, 2, and 4.

The use of the BMA for searching w in a text t is quite obvious. The pattern w occurs in t if and only if it reaches a position in the text such that the last unknown character is read and matches the corresponding character of the text in the current position (that is, when δ_{bool} is *true*). The pseudocode of the algorithm is shown in Figure 3.

For extended BMA, different strategies could be adopted according to which unknown position is queried. For example, if we define $P(q)$ as the position of the *left most* unknown character of q, we obtain the classical deterministic automaton that reads the text from left to right as the Knuth-Morris-Pratt algorithm [KMP77]. However, in all cases, Q consists only of the states that are accessible from the initial state by successive applications of the transition function. In the rest of the paper, with the exception of Section 6, we use a standard BMA, that is, the value of $P(q)$ is always the position of the rightmost # in q.

We now establish two technical properties that only apply in the standard case and that will prove useful later on to bound the number of states under some additional hypothesis. An arbitrary

state is of the form:

$$\#^{i_0} w_1 \#^{i_1} \cdots w_{k-1} \#^{i_{k-1}} w_k \tag{1}$$

where $k > 0, i_0 \geq 0, i_s > 0$, for $s = 1, \ldots, k-1, w_1, w_2, \ldots, w_{k-1} \in \Sigma^+$ and $w_k \in \Sigma^*$. Then the maximal factors $w_1, w_2, \ldots, w_{k-1} \in \Sigma^+$ carry an information on w that is made explicit by the following two propositions which are verified by induction on the length d of a shortest path leading from the initial state to the current state q. In other words, if m denotes the length of the pattern w, then d is the minimal number of transitions used for reaching q from the state $\#^m$. If $d = 0$, then $q = \#^m$, else there exists a state q' at distance $d-1$ and a letter $a \in \Sigma$, such that: $q' = \delta(q, a)$. The following observations are valuable for the proof of the propositions. By definition of the transition function in the standard BMA, there are two possibilities for the general form (1) of q':
Case A: $w_k \neq \epsilon$. Then q' differs from q by an occurrence of the letter a in the suffix w_k of w. If the occurrence a is the first letter of w_k (resp. is not the first letter of w_k), by setting $w_k = az_k$ with $z_k \in \Sigma^*$ (resp. $w_k = z_k a z_{k+1}$ with $z_k \in \Sigma^+, z_{k+1} \in \Sigma^*$) we have:
Subcase A.1: $q' = \#^{i_0} z_1 \#^{i_1} \ldots z_{k-1} \#^{i_{k-1}+1} z_k$
(resp. Subcase A.2: $q' = \#^{i_0} z_1 \#^{i_1} \ldots z_{k-1} \#^{i_{k-1}} z_k \# z_{k+1}$) with

$$z_1 = w_1, z_2 = w_2, \ldots, z_{k-1} = w_{k-1} \ . \tag{2}$$

Case B: $w_k = \epsilon$. In this case, q is obtained from q' by a shift. If the occurrence a is the first letter of w_{k-1} (resp. is not the first letter of w_{k-1}), for some integer $r \geq k-1$ and some $z_r \in \Sigma^*, w_{k-1} = az_r$ (resp. for some $z_r \in \Sigma^+, z_{r+1} \in \Sigma^*, w_{k-1} = z_r a z_{r+1}$) we have:
Subcase B.1: $q' = \#^{j_0} z_1 \#^{j_1} \ldots z_{r-2} \#^{j_{r-2}} z_{r-1} \#^{j_{r-1}} z_r$
(resp. Subcase B.2: $q' = \#^{j_0} z_1 \#^{j_1} \ldots z_{r-1} \#^{j_{r-1}} z_r \# ^{j_r} z_{r+1}$)
with: $j_{r-k+1} \geq i_0, j_{r-k+2} = i_1, \ldots, j_{r-2} = i_{k-3}$ and $j_{r-1} = i_{k-2} + 1$ (resp.$j_{r-1} = i_{k-2}, j_r = 1$), and

$$z_{r-k+2} = x w_1 \text{ for some } x \in \Sigma^*, z_{r-k+3} = w_2, \ldots, z_{r-1} = w_{k-2} \ . \tag{3}$$

Proposition 2.3: The words $w_2, \ldots, w_{k-1}$ disagree in exactly one position with the suffix of w of the same length. That is, for all $2 \leq j \leq k-1$, there exists a unique $1 \leq i \leq m_j = |w_j|$ such that $w_j[i] \neq w[m - m_j + i]$. If $i_0 > 0$, then the same applies to w_1 and if $i_0 = 0$, then w_1 has at most one position where it disagrees with the suffix of w of the same length.
Proof: By induction on the length d of a shortest path leading from the initial state to q. If $d = 0$ there is nothing to prove since $k = 1$. Now we fall in either case treated above. Case A follows from (2). Case B follows from (3) and the equalities $w_{k-1} = az_r$ (subcase B.1) and $w_{k-1} = z_r a z_{r+1}$ (subcase B.2). ■

Proposition 2.4: There exist words $u_1, u_2, \ldots, u_{k-1} \in (\Sigma \cup \{\#\})^*$ with $|u_1| > |u_2| > \ldots > |u_{k-1}|$, words $w'_1, w'_2, \ldots, w'_{k-1} \in (\Sigma \cup \{\#\})^*$ and a path from the initial state to the current state q visiting successively $k-1$ states of the form:

$$u_1 w_1 \#^{i_1} w'_1, u_2 w_1 \#^{i_1} w_2 \#^{i_2} w'_2, \ldots, u_{k-1} w_1 \#^{i_1} \ldots w_{k-1} \#^{i_{k-1}} w'_{k-1} \ .$$

Proof: As in the previous proposition we argue by induction on the length d of a shortest path leading from the initial state to q. Observe that if $k = 1$ then there is nothing to prove. Furthermore, we may assume $d > 0$ else we have $q = \#^m$, i.e., $k = 1$.

Subcase A.1: Clearly, the $k-1$ states obtained by applying the induction hypothesis to q' also do for q.
Subcase A.2: It suffices to consider the first $k-1$ steps of the k-step sequence obtained from q' by the induction hypothesis.
Case B: We may treat the subcases B.1 and B.2 at the same time. Applying the induction hypothesis to q', there exist words $v_1, v_2, \ldots, v_{r-1} \in (\Sigma \cup \{\#\})^*$ with $|v_1| > |v_2| > \ldots > |v_{r-1}|$, words $z'_1, z'_2, \ldots, z'_{r-1} \in (\Sigma \cup \{\#\})^*$, and a path from the initial state to the state q' going successively through the $r-1$ intermediate states

$$\begin{aligned}
q_1 &= v_1 z_1 \#^{j_1} z'_1 \\
q_2 &= v_2 z_1 \#^{j_1} z_2 \#^{j_2} z'_2 \\
&\ldots \\
q_{r-k} &= v_{r-k} z_1 \#^{j_1} \ldots z_{r-k} \#^{j_{r-k}} z'_{r-k} \\
q_{r-k+1} &= v_{r-k+1} z_1 \#^{j_1} \ldots z_{r-k+1} \#^{j_{r-k+1}} z'_{r-k+1} \\
&\ldots \\
q_{r-2} &= v_{r-2} z_1 \ldots \#^{j_1} \ldots z_{r-2} \#^{j_{r-2}} z'_{r-2} \\
q_{r-1} &= v_{r-1} z_1 \ldots \#^{j_1} \ldots z_{r-1} \#^{j_{r-1}} z'_{r-1}
\end{aligned}$$

We define:

$$\begin{aligned}
u_1 &= v_{r-k+2} z_1 \#^{j_1} \ldots z_{r-k+1} \#^{j_{r-k+1}} \\
u_2 &= v_{r-k+3} z_1 \#^{j_1} \ldots z_{r-k+1} \#^{j_{r-k+1}} \\
&\ldots \\
u_{k-2} &= v_{r-1} z_1 \#^{j_1} \ldots z_{r-k+1} \#^{j_{r-k+1}}
\end{aligned}$$

Then in view of (3) we have:

$$\begin{aligned}
q_{r-k+2} &= u_1 z_{r-k+2} \#^{j_{r-k+2}} z'_{r-k+2} = u_1 x w_1 \#^{j_{r-k+2}} z'_{r-k+2} \\
&\ldots \\
q_{r-2} &= u_{k-3} z_{r-k+2} \#^{j_{r-k+1}} \ldots z_{r-2} \#^{j_{r-2}} z'_{r-2} = u_{k-3} x w_1 \#^{i_1} \ldots w_{k-3} \#^{i_{k-3}} z'_{r-2} \\
q_{r-1} &= u_{k-2} z_{r-k+2} \#^{j_{r-k+1}} \ldots z_{r-1} \#^{j_{r-1}} z'_{r-1} = u_{k-2} x w_1 \#^{i_1} \ldots w_{k-2} \#^{i_{k-2}} z'_{r-1}
\end{aligned}$$

Thus, the $k-1$ states $q_{r-k+1}, q_{r-k+2}, \ldots, q_{r-1}, q$ satisfy the condition. ■

Proposition 2.3 defines a necessary condition for a state of the automaton to be accessible. Proposition 2.4 implies that w has $k-1$ occurrences of w_1, $k-2$ occurrences of w_2, etc.

3 Building the Automaton

Let w be a pattern of length m. In Section 2 we gave the conditions necessary to find, for every transition, $\delta(q, a)$, the next state, q', and its associated shift s. To compute $\delta_{bool}(q, a)$ we use

$$\delta_{bool}(q, a) = \begin{cases} true & \text{if } q.a = w \\ false & \text{otherwise} \end{cases}$$

where $q.a$ is the string q with the letter a in position $P(q)$.

Thus, starting from $q_0 = \#^m$ we can easily construct the BMA in a recursive way. Let $|Q|$ be the total number of states. For each state transition we have to compute:

- whether or not it matches ($O(m)$ time),
- the amount to shift ($O(m^2)$ time, by using a naive approach), and
- the next state ($O(m \log |Q|)$ time, by using a balanced search tree to store the current generated set of states).

For every state, there are at most $m + 1$ different transitions. Every character in the alphabet is mapped to one of those transitions. Therefore, using a simple approach, the total time is $O(|Q|m(m^2 + m \log |Q|))$. We improve this by presenting an $O(m^2|Q|)$ building algorithm.

We improve first the time to compute the shift by designing an algorithm that computes every new state in time proportional to the length of the pattern at the cost of doubling the storage required for each state.

Let q be a state, let $i = P(q)$ be the position of its rightmost occurrence of $\#$ and let a be a letter. Now assume that with every state $q \in Q$ is associated the queue $Shift$ consisting of all its nonzero shifts sorted in increasing order. Formally, we have:

(3.1) $s \in Shift$ if and only if $0 < s \leq m$ and for all $s < k \leq m$, $q[k] \neq \#$ implies $q[k] = w[k - s]$.

In case of a mismatch, that is, $a \neq w[i]$, the $O(m)$ test of condition (2.1) can be replaced by a single test:

(3.2) $r[i] = w[i - s]$ for all possible shifts s.

Altogether this requires time $O(m)$. It needs only to be shown that updating the queue from q to q' can be achieved in time $O(m)$.

Denote by $NewShift$ the queue of shifts associated with q'. We make use of the standard primitives on queues (for example, the operation $DeQueue$ retrieves the first element of the queue and deletes it).

Case 1. If $a = w[i]$ then $NewShift$ is obtained by retaining those elements in $Shift$ that are compatible with the occurrence of a. This algorithm is shown in Figure 4.

Case 2. If $a \neq w[i]$ then denote by min the shortest element in $Shift$ that is compatible with the occurrence $a = w[i - min]$. Then observe that if s is a shift of q', then $s + min$ is a shift of q (if $s < m - min$), but that the converse is not necessarily true. We divide the set $NewShift$ into three subsets:

- $NewShift1 = \{s \in NewShift \mid 0 < s < i - min\}$. An integer $0 < s < i - min$ belongs to $NewShift1$ if and only if $s + min$ belongs to $Shift$ and $a = w[i - s - min]$.
- $NewShift2 = \{s \in NewShift \mid i - min \leq s < m - min\}$. An integer $i - min \leq s < m - min$ belongs to $NewShift2$ if and only if $s + min$ belongs to $Shift$.
- $NewShift3 = \{s \mid m - min \leq s \leq m\}$.

Figure 5 shows an example and the algorithm is given in Figure 6.

As a result we have:

THEOREM **3.1** *Every state of the BMA associated with a pattern of length m can be computed in time $O(m)$.*

Finally, we improve the time to find whether the new state already was generated or not, and to what number it should be mapped. For this, we simply replace the balanced search tree (state dictionary) by a trie. This trie stores every state q, having a reference to its associated enumeration. Note that the trie is binary, because every position is either a # or the corresponding character in the pattern. It is not difficult to show that the number of nodes in this trie is proportional to $|Q|$ (one leaf per state). The search time for a given state is in the worst case $O(m)$ letter comparisons (instead of $O(m \log |Q|)$).

The total space needed by the automaton is $O(m)$ for each state (the transitions), hence $O(m|Q|)$ total space is needed. During the construction, a similar amount of space is needed. In Section 7 we discuss how to compute the BMA "on the fly", thus generating only the states needed to search a specific text.

According to the discussion above, we state the following result:

Lemma 3.1 *A BMA having $|Q|$ states can be built using $O(m^2|Q|)$ worst case time and $O(m|Q|)$ space.*

We conjecture that this algorithm is, both, time and space optimal. Unless there is an implicit representation for the automaton, in the worst case there are $O(m)$ different letters in the pattern. This implies that $O(m)$ transitions must be computed, which implies $\Omega(m|Q|)$ space for the automaton. In addition, for every transition, we have to generate a new state, q', which has m letters. Thus, computing the next state implies at least $O(m)$ time, which gives $\Omega(m^2|Q|)$ total time.

4 Bounds on the Number of States

In this section we present some results concerning the number of states of a BMA. Clearly, the number of states, $|Q|$, of a standard BMA (extended BMA too) is bounded by

$$m \leq |Q| \leq 2^m - 1 \ ,$$

for any alphabet size. We need m states to recognize the pattern (the m possible proper suffixes of the pattern) and the upper bound is given by every possible subset of the pattern with the exception of the pattern itself. In the following we assume that $|\Sigma| > 1$, otherwise the problem is trivial.

Another interesting question is the size of the BMA for fixed m, but large $|\Sigma|$ ($|\Sigma| > m$). For example, for any m, the minimum number of states is $2m - 1$, achieved by the pattern $a^{m-1}b$. If all characters are different or all the characters are equal, we have $|Q| = m(m+1)/2$ [KMP77]. We can generalize this result for a pattern with j different letters:

Lemma 4.1 *For a pattern of length m with j different letters*

$$|Q| \geq m + \sum_{i=1}^{j} (m - Pos(x_i)) \ ,$$

where $Pos(x_i)$ is the position of the rightmost position of the i-th different letter x_i. This bound is exact when $j = m$.

Proof: First we have the m proper prefixes of the pattern. Consider now the i-th different letter x_i of the pattern. From state $\#^m$ if we find x_i in the text, we go to state $\#^{Pos(x_i)-1} x_i \#^{m-Pos(x_i)}$. If we match the next $m - Pos(x_i) - 1$ rightmost characters of the pattern, we have that many new states. So, in total, $m - Pos(x_i)$ new states. Adding over all different letters we obtain the stated result. ■

Let $|Q|_{max}$ be the maximum number of states for a BMA. Table 2 gives the value of $|Q|_{max}$ for some m and $|\Sigma|$. This table suggests that for a given m, $|Q|_{max}$ is the same for all $|\Sigma| \geq \lceil \log_2 m \rceil$. This is obviously true for $|\Sigma| \geq m$ because there are at most m different characters in a pattern of length m.

	m												
$\|\Sigma\|$	1	2	3	4	5	6	7	8	9	10	11	12	13
2	1	3	6	12	20	30	42	57	83	106	155	196	281
3		3	6	12	21	33	50	69	93	131	186		
4			6	12	21	33	50	69	99	137			
5				12	21	33	50	69	99				
6					21	33	50						

Table 2: Maximum number of states for fixed m and $|\Sigma|$.

THEOREM **4.1** *For a pattern of length m, and an alphabet of size $|\Sigma| \geq 2$,*

$$|Q|_{max} = \Omega(m^3) \ .$$

Proof: We show that this bound is valid for $|\Sigma| = 2$. The same result applies trivially for $|\Sigma| > 2$ (we use just two letters). Consider the pattern $w = a^{i_1} b a^{i_2}$ with $i_2 > i_1$. The following sequence of transitions leads to states of the form $a^j \#^+ b a^k \#^+ a^\ell$ for a range of values of j, k, and ℓ:

- We start matching the pattern obtaining states of the form $\#^+ a^j$.
- If $j < i_2$ and we find a b, we go to state $\#^+ b a^j \#^+$. Again, if we start matching the pattern we obtain states of type $\#^+ b a^j \#^+ a^k$.
- If $j + k < i_2$ and we find a b, we go to state $a^j \#^+ b a^k \#^+$ where j is less or equal than the previous j.

- If we start matching the pattern, we obtain states of the form $a^j\#^+ba^k\#^+a^\ell$, where at least $1 \leq j \leq \min(i_1, i_2 - k - 1)$, $1 \leq k \leq i_2 - 2$, and $1 \leq \ell \leq i_2 - k - 1$. Thus, the total number of states is $\Omega(m^3)$.

A detailed analysis shows that the worst case occurs when $i_1 \approx i_2/2$, giving $|Q|_{max} = 2m^3/27 + O(m^2)$. ■

This class of patterns is actually the worst case for $m = 3 \ldots 8$, but does not seem to be tight for large m. The reader will certainly be convinced of how more intricate is the computation of the automaton of a pattern possessing exactly two occurrences of b by trying to figure it out by him/herself. In order to prove lower bounds, it is tempting to study how the size of automata behaves with respect to operations on words such as concatenation. However, it is not clear how the size of the automaton of w_1w_2 compares with the sizes of the automata of w_1 and w_2. Substitution of letters may lead to bigger or smaller automata depending on the pattern. Some recent "computational" results seem to show that there may exist patterns with larger number of states, at least $O(m^6)$ or perhaps exponential ($O((4/3)^m)$) [Bru91, Sch92].

As discussed in the introduction, no polynomial upper bound on the number of states of the BMA is known so far. We are only able to prove that under a certain restriction a polynomial upper bound exists. Here we consider the parameter k equal to the maximum number of times a letter may occur in a word. When k is fixed, it can be shown that the size of the automaton as a function of the length m is bounded by a polynomial of degree k. Our proof is based on the simple necessary condition (Proposition 2.3) for a word in $(\Sigma \cup \{\#\})^*$ to be a state of the automaton. It is interesting to observe that this condition is strong enough to prove the upper bound established in [KMP77] in the case where all letters are distinct. Thus, our general assumption is that every letter of Σ occurs at most k times in the pattern w.

Lemma 4.2 *Let $k > 0$ be a fixed integer and $w \in \Sigma^*$. Then w has $O(k)$ occurrences $u \in \Sigma^*$ satisfying the condition:*

(4.1) *u disagrees in exactly one position with the suffix of w of length $|u|$.*

Proof: Let $1 < \ell < m$ be an integer. We show that w has at most $2k$ occurrences of length ℓ satisfying the condition (4.1). Indeed, let a and b be the leftmost and rightmost letters of the suffix of w of length ℓ. If there were more than $2k$ occurrences of length ℓ, then at least $k + 1$ of them would be starting with an a or ending with a b, thus violating the condition on the number of occurrences. Clearly the condition on the number of occurrences is necessary as shown by $w = a^mb$. ■

We now prove our polynomial upper bound.

THEOREM **4.2** *If every letter of the pattern $w \in \Sigma^*$ occurs at most k times, then the BMA has m^{k+1} states.*

Proof: Because of Proposition 2.4 an arbitrary state is of the form: $\#^{i_0}w_1\#^{i_1} \cdots w_{r-1}\#^{i_{r-1}}w_r$ with at most $k + 1$ occurrences w_j different from ϵ. Since the maximum number of occurrences is $O(m)$, the result follows. ■

Although this upper bound is exponential if k depends on m, it still allows us to compute an improved bound for $|Q|_{max}$ averaged over all possible patterns for a uniform random alphabet, when $|\Sigma| \geq m$.

Lemma 4.3 *Let f_k be the number of patterns with every letter repeated at most k times and at least one letter repeated k times. Then,*

$$f_k \leq |\Sigma| \binom{m}{k} (|\Sigma| - 1)^{m-k} \ .$$

Proof: There are $|\Sigma|$ choices for the letter repeated k times, and there are $\binom{m}{k}$ different configurations for it. Every other letter must be one of the remaining $|\Sigma| - 1$ letters. Because for $k \leq m/2$ we are counting some patterns that have a letter repeated more than k times, we obtain an upper bound. However, this number is tight for $k > m/2$, which is the range of interest. ■

THEOREM **4.3** *The maximal number of states averaged over all possible patterns is bounded by*

$$\overline{|Q|}_{max} \leq m|\Sigma|\beta^m$$

where $\beta = 1 + \frac{m-1}{|\Sigma|}$. Note that $\beta < 2$ for $|\Sigma| \geq m$.

Proof: There are $|\Sigma|^m$ different patterns. Then,

$$\overline{|Q|}_{max} \leq \frac{1}{|\Sigma|^m} \sum_{k=1}^{m} m^{k+1} f_k \leq m|\Sigma|^{1-m} \sum_{k=1}^{m} \binom{m}{k} m^k (|\Sigma| - 1)^{m-k} \leq m|\Sigma|^{1-m}(m + \Sigma - 1)^m$$

from which the result follows. ■

5 Average and Worst Case Analysis

To analyze the searching time of this automaton, we define a *potential function* that applies to each transition. Let $K(q)$ (knowledge) be the number of known characters in the text that is recorded by a state q. We say that the potential of a transition is

$$\phi(q, a) = \delta_s(q, a) + K(\delta(q, a)) - K(q) \ .$$

This quantity represents how much we shift in each step plus how much information we gain or lose about the characters in the text corresponding to the current position of the pattern.

Lemma 5.1 $1 \leq \phi(q, a) \leq m$ *for every state $q \in Q$ and $a \in \Sigma$, where m is the length of the pattern.*

Proof: If the shift is zero, we know one more character, thus the potential is one. Similarly, if we shift $m - j$ characters, $1 \leq j < m$, and if k_1 is the knowledge that we keep, k_2 the knowledge that we lose, then

$$\phi(q, a) = m - j \ + \ k_1 - (k_1 + k_2 - 1) = m - j + 1 - k_2 \ .$$

Because $0 \leq k_2 \leq m - j$, we have

$$1 \leq \phi(q, a) \leq m \ .$$

Finally, if we shift m characters, we do not keep any knowledge, and the previous knowledge is at most $m - 1$ characters. Therefore the same bounds apply. ■

From the previous lemma, we have that the total number of transitions is bounded by

$$\left\lfloor \frac{n}{m} \right\rfloor \leq t_n \leq n \ ,$$

because the potential is bounded by 1 and m.

If the alphabet is known in advance (as in most practical cases), we can implement the transition function as several tables. In that case, the worst case search time is n table lookups. Otherwise, there are at most m different letters in the pattern. If the alphabet is large, we can implement the transition function as an ordered table, where we can retrieve in $O(\log m)$ time.

A natural approach to the average case analysis of automata is to use a Markov chain with the same states as the automaton. This approach is used by Baeza-Yates [BY89a, BY89b]. First, we need some basic results from Markov chains. A stochastic process is a Markov process if the probability of one event depends only on the previous event. A Markov chain is a Markov process in discrete time with a discrete state space. In a Markov chain, each event is generally associated with a state. The above definition is equivalent to saying that the probability of a transition from time j to time $j + 1$ depends on the state at time j.

Let $S = \{s_1, \cdots, s_r\}$ be the possible states in a Markov chain. Then, the *transition matrix* of the process is an $r \times r$ matrix defined as

$$\mathbf{T} = [p_{ij} = Prob\{i \rightarrow j | i\}] \ .$$

That is, p_{ij} is the conditional probability of a transition from state i to state j given that the process is in state i. Let $\vec{p}^{\,(j)} = (p_1^{(j)}, \cdots, p_r^{(j)})$ be the state probability vector at time j (for example, $p_i^{(j)}$ is the probability of being in state i at time j). Then,

$$\vec{p}^{\,(j)} = \vec{p}^{\,(0)} \mathbf{T}^j \quad (j = 1, 2, \ldots) \ ,$$

where $\vec{p}^{\,(0)}$ is the initial vector of state probabilities.

We are interested in Markov chains with no absorbing states; that is, for all states there exists at least one transition to another state. In this case, $\vec{p}^{\,(j)}$ converges to a stationary vector for large j. If $\vec{\pi}$ is the stationary vector of the state probabilities, that is

$$\lim_{j \rightarrow \infty} \vec{p}^{\,(j)} = \vec{\pi} \ ,$$

then $\vec{\pi}$ is the solution of the linear system of equations

$$\vec{\pi}(\mathbf{T} - \mathbf{I}) = 0 \quad , \quad \sum_k \pi_k = 1 \ ,$$

where $\mathbf{I}$ is the identity matrix.

For example, for our pattern $w = aab$, and assuming that the probability of the letters a and b appearing in the text are p_a and p_b respectively, the matrix $\mathbf{T}$ (see Figure 2) is

$$\mathbf{T} = \begin{bmatrix} 1-p_a-p_b & p_a & p_b & 0 & 0 \\ 1-p_a-p_b & 0 & 0 & p_a & p_b \\ 1-p_a & 0 & 0 & 0 & p_a \\ 1-p_a & 0 & 0 & p_a & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} .$$

After t transitions, the probability of being in each state is

$$\vec{p}^{\,(t)} = \vec{p}^{\,(0)}\mathbf{T}^t$$

where $\vec{p}^{\,(0)} = [1, 0, \ldots, 0]$. The expected potential in the t-th transition is computed with

$$\overline{\phi}_t = \sum_{i \in Q} p_i^{(t)} \sum_{a \in \Sigma} \phi(q_i, a) p_a \;=\; \vec{p}^{\,(t)} \otimes \vec{\Phi} \;,$$

where $\otimes$ denotes dot product and $\vec{\Phi}$ is the vector of average transition potential per state.

To search the whole text, the potential should be at least n. Thus, because we have one transition per state, the expected number of transitions $\overline{t}_n$ is defined as the minimal t such that

$$\sum_{i=0}^{t-1} \overline{\phi}_i \geq n \;.$$

For large n, we approximate $\overline{\phi}$ with the steady state solution. Thus, the weighted steady state potential is

$$\overline{\phi} = \vec{\pi} \otimes \vec{\Phi} \;.$$

Note that $\overline{\phi} = \overline{shift}$ because on average $\Delta(K) = 0$ (otherwise we gain or we lose information for free).

THEOREM **5.1** *The expected number of transitions to search a text of length n is*

$$\overline{t}_n = \frac{n}{\overline{\phi}} \;+\; O\left(\frac{m|Q|}{\overline{\phi}}\right) \;.$$

Although this result is of interest when $|Q| = o(n)$, the correction term is pessimistic.

Proof: We start from the recurrence equation

$$\vec{p}^{\,(t)} = \vec{p}^{\,(t-1)}\mathbf{T} \;= \vec{p}^{\,(0)}\mathbf{T}^t.$$

The dimension of the recurrence is $|Q|$. By definition, $\overline{\phi}_i = \vec{p}^{\,(i)} \otimes \vec{\Phi}$, so we compute

$$\sum_{i=0}^{t-1} \overline{\phi}_i = \vec{p}^{\,(0)} \left(\sum_{i=0}^{t-1} \mathbf{T}^i \right) \otimes \vec{\Phi}$$

However, $\mathbf{T}$ is singular and the sum cannot be computed directly. Let $\mathbf{A}$ be a matrix with all its rows equal to $\vec{\pi}$. But $\mathbf{T}^n = \mathbf{A} + (\mathbf{T} - \mathbf{A})^n$ for $n > 0$ (see [KS83, p.75]). So,

$$\sum_{i=0}^{t-1} \mathbf{T}^i = \Big(t\mathbf{A} - \mathbf{A} + (\mathbf{I} - (\mathbf{T} - \mathbf{A})^t)\mathbf{Z}\Big) = \Big(t\mathbf{A} - \mathbf{A} + (\mathbf{I} - \mathbf{T}^t + \mathbf{A})\mathbf{Z}\Big)$$

where $\mathbf{Z} = (\mathbf{I} - (\mathbf{T} - \mathbf{A}))^{-1}$ is a matrix that always exists ans is called the *fundamental matrix* of the Markov process [KS83]. Because $\vec{p}^{\,(0)}$ is a probability vector $\vec{p}^{\,(0)}\mathbf{A} = \vec{\pi}$. Also, because $\vec{\pi}\mathbf{Z} = \vec{\pi}$ and $\mathbf{ZT} = \mathbf{TZ}$ [KS83, p.75] we have $\vec{p}^{\,(0)}\mathbf{AZ} = \vec{\pi}$ and $\mathbf{T}^t\mathbf{Z} = \mathbf{ZT}^t$. Then, the first equation is reduced to:

$$\sum_{i=0}^{t-1} \overline{\phi}_i = \Big(t\vec{\pi} - \vec{\pi} + \vec{p}^{\,(0)}\mathbf{Z}(\mathbf{I} - \mathbf{T}^t) + \vec{\pi}\Big) \otimes \vec{\Phi}$$

Because all rows of $\mathbf{Z}$ add to 1 [KS83, p.75], and $\vec{p}^{\,(0)}$ is a probability vector, $\vec{v} = \vec{p}^{\,(0)}\mathbf{Z}$ is a vector which elements add to 1 (but not necessarily a probability vector). So, we have

$$\sum_{i=0}^{t-1} \overline{\phi}_i = t\vec{\pi} \otimes \vec{\Phi} + \Big(\vec{v}(\mathbf{I} - \mathbf{T}^t)\Big) \otimes \vec{\Phi}$$

But $\mathbf{I}$ and $\mathbf{T}^t$ are matrices with values between 0 and 1 with rows that add to 1, so because $\vec{v}$ is a vector that adds to 1, $\vec{v}(\mathbf{I} - \mathbf{T}^t)$ is a vector with elements of size $O(1)$ with rows that add to 0. Because the elements of $\vec{\Phi}$ are of size $O(m)$, and the dot product is the sum of $|Q|$ terms, the right hand term is of $O(m|Q|)$. Note that this bound is pessimistic, because in the limit ($t \to \infty$) the right hand term is of $O(m)$. Solving $\sum_{i=0}^{t-1} \overline{\phi}_i \geq n$ for t we obtain the claimed result. ■

In the previous example, the steady state solution is

$$D\ \vec{\pi} = [(1-p_a), (1-p_a)p_a, (1-p_a)p_b, p_a^2, (1-p_a)p_a p_b]\ ,$$

where $D = 1 + p_b + p_a p_b - 2p_a^2 p_b$. Hence

$$\overline{\phi} = \frac{(3 - 2p_a)}{D}\ .$$

If we consider $p_a = p_b = 1/|\Sigma|$ (uniform distribution), we obtain

$$\overline{\phi} = \frac{|\Sigma|^2(3|\Sigma| - 2)}{|\Sigma|^3 + |\Sigma|^2 + |\Sigma| - 2} = 3 - O(1/|\Sigma|)\ .$$

With this analysis we can also obtain the best or worst case distribution of probabilities of the underlying alphabet for a given pattern. For example, $\overline{\phi}$ is obviously maximal when $p_a = p_b = 0$. If $|\Sigma| = 2$, the maximal shift $(3/2)$ is given by $p_a = 0$ and $p_b = 1$.

6 Optimizing the Automaton

In this section we consider extended BMA. We can improve these automata on the average by optimizing $P(q)$ in a local manner. Global optimization for the average case is also possible,

but very expensive. In fact, we can analyze all possible values of $P(q)$, assigning a probability of choosing every one of them. Hence, the optimal $P(q)$ for all q are given by a non-linear optimization problem over those probabilities, which can take the value 0 or 1. This can be solved by simply substituting all possible valid 0/1 combinations. However, the number of variables is in general $O(2^m)$, and the number of combinations could be doubly exponential. For example, Figure 7 shows the BMA for the pattern *aba* ($|\Sigma| = 2$). The optimal automaton on the average is obtained by using 5 variables, and substituting 24 valid combinations. If $p_a \geq 1/2$, the solid line transitions of state q_2 are used. However, if $p_a < 1/2$, it is better to use $P(q_2) = 1$, which produces the dashed line transitions. This example shows that the standard value of $P(q)$ is not always the best choice.

It is simpler to optimize $P(q)$ locally with respect to the worst case transition or with respect to the average case if we know the probability distribution of the letters in the text. For each case, we can optimize either the shift or the potential of each transition. The building time in all cases is the same as before, but multiplied by m.

6.1 Worst Case Optimization

Let $\mathcal{P}(q)$ be the set of all valid values for $P(q)$ (all the places with # in q). To optimize the worst case, we consider all transitions with positive shift. Otherwise, the worst case is realized by matching one more character of the pattern. For each state q, we optimize the shift

$$P(q) = \max_{p \in \mathcal{P}(q)} \left(\min_{\substack{a \in \Sigma \\ \delta_s(q,a) > 0}} (\delta_s(q,a)) \right) ,$$

or the potential (using ϕ instead of δ_s). Table 3 shows the maximum number of states for both cases for small values of m and $|\Sigma|$. Intuitively, the number of states should be larger than the standard BMA because there are more choices for $P(q)$, and this is true for the cases computed.

shift	$\|\Sigma\|$					potential	$\|\Sigma\|$				
m	2	3	4	5	6	m	2	3	4	5	6
1	1					1	1				
2	3	3				2	3	3			
3	6	6	6			3	6	6	6		
4	11	12	12	12		4	11	12	12	12	
5	17	20	20	20	20	5	19	21	21	21	21
6	27	31	31	31	31	6	31	32	32	32	32
7	41	47	47	47		7	48	54	54	54	
8	59	76	76			8	71	79	79		
9	85	111				9	104	117			
10	116	162				10	133	177			
11	162					11	196				

Table 3: Maximum number of states for worst case optimization.

6.2 Average Case Optimization

For the average case we assume that the probability of finding the letter a in the text is p_a. We then optimize the shift with

$$P(q) = \max_{p \in \mathcal{P}(q)} \left(\sum_{a \in \Sigma} \delta_s(q, a) p_a \right) ,$$

or the potential (using ϕ instead of δ_s). For a large alphabet this converges to the rightmost position (standard case), because the probability of finding a character that is not in the pattern increases. Table 4 shows the maximum number of states for small m and $|\Sigma|$, assuming a uniform alphabet distribution.

shift	$\|\Sigma\|$					potential	$\|\Sigma\|$				
m	2	3	4	5	6	m	2	3	4	5	6
1	1					1	1				
2	3	3				2	3	3			
3	6	6	6			3	6	6	6		
4	11	12	12	12		4	11	12	12	12	
5	17	20	21	21	21	5	19	21	21	21	21
6	27	33	31	31	33	6	31	35	30	33	33
7	41	49	48	48		7	48	56	55	50	
8	59	75	71			8	71	75	78		
9	85	112				9	104	123			
10	116	149				10	133	165			
11	162					11	196				

Table 4: Maximum number of states for average case optimization (uniform distribution).

Comparing both tables, it appears that optimizing the potential results in the production of more states. Another interesting fact obtained during the computation of the above tables is that the minimum number of possible states is greater than $2m - 1$, the value for the standard case.

Table 5 gives the number of states and the expected shift when searching the patterns a^3ba^6 with $|\Sigma| = 2$ and $|\Sigma| = 3$, and the pattern *abracadabra* with $|\Sigma| = 5$ and $|\Sigma| = 6$ in a text with uniform distribution. It seems that optimizing the shift on average is the best choice, but in general, that will depend on the pattern. Clearly, as expected, the worst case optimization does not improve the average case. From the table we can see that the BMA can be improved up to 7% from KMP's suggestion in some cases, and that for second pattern there is no difference between optimizing the shift or the potential.

7 Bounding the Number of States

The main objection to this class of automata is that the number of states may be too large. One solution is to bound the number of states. For example, we may keep the main chain plus all states which obey certain properties.

Pattern	$P(q)$	Number of states		Expected shift	
		shift	potential	shift	potential
aaabaaaaaa ($\|\Sigma\| = 2$)	Standard	89		2.8008	
	Worst Case	76	109	3.0163	2.9443
	Average Case	76	109	3.0163	2.9443
aaabaaaaaa ($\|\Sigma\| = 3$)	Standard	104		5.0359	
	Worst Case	88	121	4.9485	5.0573
	Average Case	92	111	5.0629	5.0529
abracadabra ($\|\Sigma\| = 5$)	Standard	74		5.6424	
	Worst Case	101		5.3616	
	Average Case	82		5.4192	
abracadabra ($\|\Sigma\| = 6$)	Standard	74		6.2267	
	Worst Case	101		5.9222	
	Average Case	81		6.2508	

Table 5: Results for the optimized BMA of two patterns and different alphabet sizes.

If we know the probability of being in each state, the obvious choice is to keep the states with higher probabilities. We can know this through the average case analysis. However, we would like to decide *a priori* the best states to keep. A first approximation is to assume that the text is random with a known probability distribution. Thus, the probability of being in state q is proportional to $\prod_{a \in q} p_a$. For a uniform distribution (alphabet of size $|\Sigma|$), if we know k characters in state q, the probability is proportional to $|\Sigma|^{-k}$. Thus, the simplest heuristic is to keep the states where we know at most k characters (plus the main chain). To normalize a given state to a valid state, we keep the known suffix and/or the rightmost known k characters. Therefore, the number of states is bounded by

$$|Q| \leq m - k + \sum_{i=1}^{k} \binom{m}{i} \;=\; O(m^k) \;.$$

The case $k = 0$ is the classical *Boyer-Moore* algorithm (m states). If $k = 1$, we obtain $m \leq |Q| \leq 2m - 1$, and when $k = m - 1$ we obtain the complete automaton.

Table 6 gives the number of states and the expected shift when searching two different patterns in a uniform random text. The average case analysis for a BMA with a bounded number of states is not trivial (see [BYR92]). In this case, it is not possible to use a Markov model in the reduced set of states because we have lost information, and every comparison is not random (not always is a new letter). For this reason, the expected shift of the bounded versions was obtained by simulation, which consisted in several runs of searching large randomly generated texts (the difference for $k = 10$ is due to experimental variations).

Note that for the pattern *abracadabra* and $k = 1$, the expected shift is almost 40% larger than the one for the Boyer-Moore algorithm, and that this is very close to the case of $k = m - 1$.

For all the variations presented, the set of states generated is not necessarily minimal. For example, for the pattern a^3ba^6 and $k = 1$, eleven states are sufficient and necessary. To minimize the

Pattern	*abracadabra* ($\|\Sigma\| = 6$)		*aaabaaaaaa* ($\|\Sigma\| = 3$)	
k	Number of states	Expected shift	Number of States	Expected Shift
0	11	4.3752	10	4.9603
1	21	6.1158	19	4.9618
2	28	6.2259	27	4.9807
10 $(m-1)$	74	6.2267	104	5.0321
(exact)		(6.2267)		(5.0359)

Table 6: Simulation results for the bounded BMA of two patterns.

number of states of a bounded BMA, we can use any standard deterministic finite automaton state minimization algorithm in each group of states with identical $P(q)$, by extending class equivalence to the δ_{bool} and δ_s components of the transitions.

The main drawback of the previous approach is that in practical cases we do not know the probability distribution of the underlying text. One solution is to build the automaton "on the fly", that is, while searching the text. This is similar to the lazy evaluation proposed by Aho [Aho90] for regular expressions. The main ideas are:

- Set a maximum number of possible states (for example $5m$). Build the main chain of the automaton, keeping Q in a dictionary (a trie or a hash table), and setting all unknown transitions as undefined.
- While searching, if a transition is undefined, we either create a new state if we have not reached the maximum, or we throw away information (leftmost known character) until we find a known state. This new transition will now replace the undefined one. This is trivially done using the binary trie mentioned in Section 3.

If the text is homogeneous, the most probable states will be generated, and if the maximum number of states is large enough, we have an automaton that is adapted to the current input with a bounded number of states. In contrast with Aho's scheme, we do not need to use a replacement policy when there is no space left for a new transition, because we are just searching for a string. On the other hand, we can also use a replacement scheme which may improve the adaptivity of the algorithm.

8 Concluding Remarks

We have presented several variations of Boyer-Moore type algorithms described as an automaton. All classical algorithms for string matching are particular cases (the Knuth-Morris-Pratt [KMP77], Boyer-Moore [BM77], Boyer-Moore-Galil [Gal79], and Boyer-Moore-Horspool [Hor80] algorithms). This type of automata is asymptotically optimal on the average because it is similar to the algorithms presented by Yao [Yao79]. That is, $O((\log_{|\Sigma|} m)/m)$ inspections per character in the text are needed. A BMA is also optimal in the worst case because at least $n - m + 1$ inspections are needed in general [Riv77].

Possible drawbacks are the processing time and additional memory for the states. Bounding *a priori* the number of states, with an "on the fly" construction, limits both. From our experience, these machines have very good practical value and improve upon other Boyer-Moore-type algorithms. In practice, for reasonable pattern and alphabet sizes, bounding linearly the number of states yields an $O(m^3)$ preprocessing time. This is insignificant when searching very large pieces of text.

It was said in Section 2 that the BMA was defined as consisting only of all states that are accessible from the initial state by the transition function. However, we suspect it would be interesting to investigate the transition function acting on all possible states (that is, all 2^m words u of length m satisfying $u[i] = w[i]$ or $u[i] = \#$). Experimentally this automaton usually consists of very few recurrent states, that is, states for which there exists a nonempty word z such that $\delta(q, z) = q$. Furthermore, those form a unique strongly connected component (with some exceptions, such as the automata associated with the words of the form $ab^n a$ that possess several components). As a consequence, the number of states that are accessible from a given starting state seem to be of the order of $O(m^3)$, independently of the state. Let us say that a state is isolated if it is the image (under the transition function) of no state. Here we establish under which condition the initial state $\#^m$ is isolated.

Lemma 8.1 *Let w be a pattern. Then the initial state of the BMA is isolated if and only if the following two conditions are satisfied:*

(1) *w has a border (denote by $u \neq \epsilon$ the minimum word of a factorization $w = uw' = w''u$).*

(2) *for all $i = 1, \ldots, \ell = |u|$ the words obtained from w by modifying the $(m - i + 1)$-th letter of w are a factor of w or have a nonempty suffix that is a prefix of w.*

Proof: Assume w has no border. Consider the state $q = \#w[2]w[3]\ldots w[m]$ and a letter $b \neq w[1]$. The word obtained from q by substituting b for $\#$ certainly has no suffix that is a prefix of w, else w would have a border. Thus condition (1) is necessary. Furthermore, assume the condition does not hold, that is, for some $0 < i \leq \ell$ and some $b \neq w[m-i+1]$ the word $w[m-\ell+1]...w[m-\ell+i]w[m-\ell+1]b\ w[m-\ell+2]...w[m]$ is not a factor of w or has no suffix that is prefix of w. Then the transition defined by the letter b on the state $q = \#^{m-\ell}w[m-\ell+1]...w[m-\ell+i]w[m-\ell+1]\#w[m-\ell+2]...w[m]$ yields the initial state which also proves the necessity of (2). Conversely, assume the two conditions are satisfied. Consider an arbitrary state

$$q = q[1]\ldots\#w[k+1]\ldots w[m]$$

and a letter $b \neq w[k]$. Let r be the word obtained from q by substituting b for $\#$. If $k < m - \ell$ then q has a suffix equal to u, and so r has a nonempty suffix that is a prefix of w. If $m - \ell \leq k \leq m$, then the word r also has a nonempty suffix that is a prefix of w. In all cases the new state is different from the initial one. ■

Further research is being done to find better bounds for the maximum number of states, and to extend the construction to the case of multiple patterns.

Acknowledgements

We acknowledge the helpful comments of Véronique Bruyère, Robert Dailey, Mireille Régnier, Rodrigo Scheihing, and the anonymous referees.

References

[AG86] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J on Computing*, 15:98–105, 1986.

[Aho90] A.V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier, 1990.

[BM77] R. Boyer and S. Moore. A fast string searching algorithm. *C.ACM*, 20:762–772, 1977.

[Bru91] Véronique Bruyère. Thèse annexe, automates de Boyer-Moore. Technical report, Institut de Mathématique et d'Informatique, Université de Mons-hainaut, 1991.

[BY89a] R.A. Baeza-Yates. *Efficient Text Searching.* PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.

[BY89b] R.A. Baeza-Yates. String searching algorithms revisited. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Workshop in Algorithms and Data Structures*, pages 75–96, Ottawa, Canada, August 1989. Springer Verlag Lecture Notes on Computer Science 382.

[BYGR90] R. Baeza-Yates, G. Gonnet, and M. Régnier. Analysis of Boyer-Moore-type string searching algorithms. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 328–343, San Francisco, January 1990.

[BYR92] R. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theoretical Computer Science*, 92(1):19–31, January 1992.

[Cho90] C. Choffrut. An optimal algorithm for building the Boyer-Moore automaton. *Bull. of EATCS*, 40:217–224, Jan 1990.

[Gal79] Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *C.ACM*, 22:505–508, 1979.

[Gal85] Z. Galil. Open problems in stringology. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 1–8. Springer-Verlag, 1985.

[GO80] L. Guibas and A. Odlyzko. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM J on Computing*, 9:672–682, 1980.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10:501–506, 1980.

[KMP77] D.E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J on Computing*, 6:323–350, 1977.

[KS83] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer-Verlag, New York, 1983.

[Riv77] R. Rivest. On the worst-case behavior of string-searching algorithms. *SIAM J on Computing*, 6:669–674, 1977.

[Sch92] R. Scheihing. personal communication. 1992.

[Yao79] A.C. Yao. The complexity of pattern matching for a random string. *SIAM J on Computing*, 8:368–387, 1979.

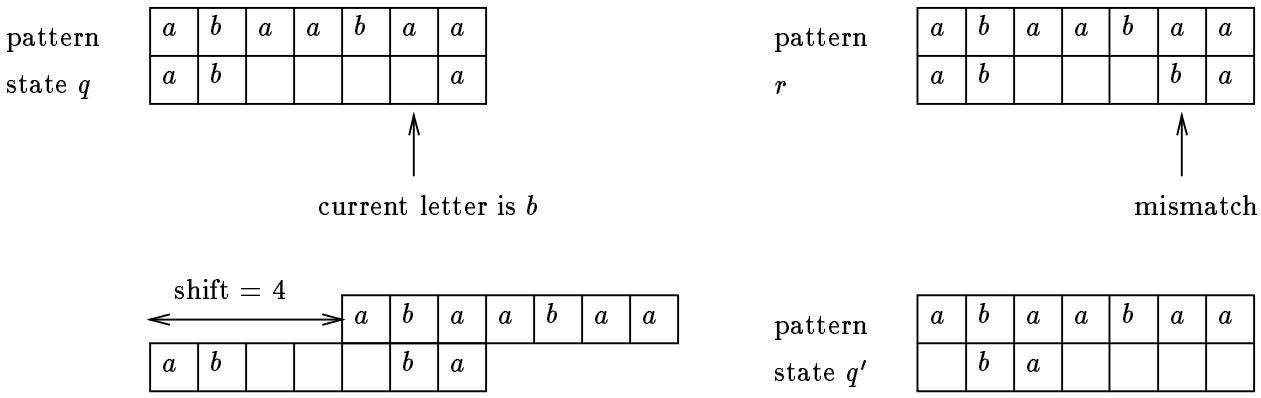

Figure 1: Example for pattern $abaabaa$.

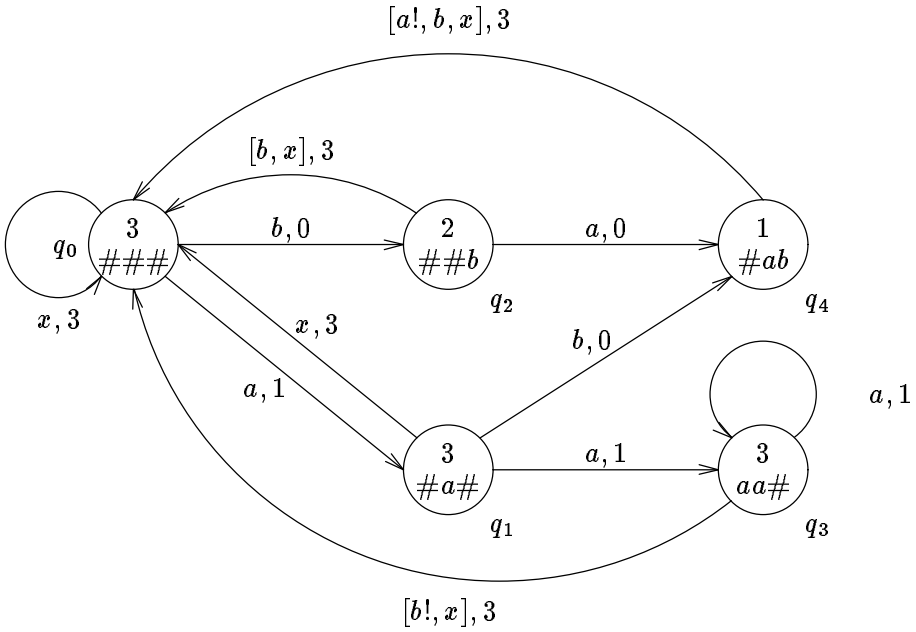

Figure 2: BMA for aab.

```
Search( (text, n), (pattern, m) )
{
    (δ, P) ← BuildAutomaton( pattern )
    k ← 0
    q ← 0
    while k < n − m + 1 do
    {
        a ← text[k + P(q)]
        if δ_bool(q, a) then print( match at position k + 1 )
        k ← k + δ_s(q, a)
        q ← δ(q, a)
    }
}
```

Figure 3: BMA searching algorithm.

```
while NotEmpty(Shift) do
{
    s ← DeQueue(Shift)
    if s ≥ i or a = w[i − s] then
        EnQueue(s, NewShift)
}
```

Figure 4: Computation of $NewShift$ (Case 1).

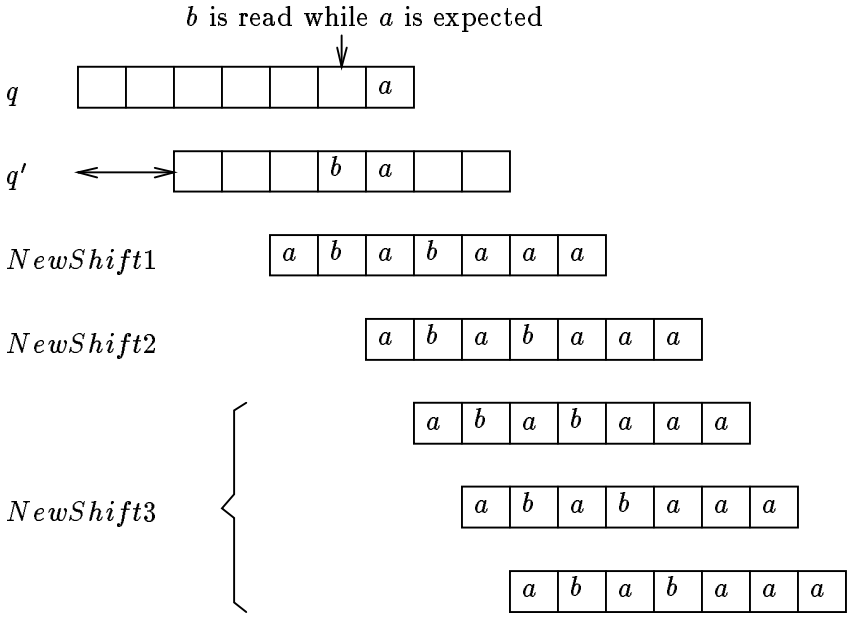

Figure 5: Example for pattern *ababaaa*.

```
repeat
    min ← DeQueue(Shift)
until (min ≥ i) or (a = w[i − min])
if NotEmpty(Shift) then
{
    s ← DeQueue(Shift)
    while s < i do            % NewShift1
    {
        if w[i − s] = a then EnQueue(s − min, NewShift)
        s ← DeQueue(Shift)
    }
    while s < m do            % NewShift2
    {
        EnQueue(s − min, NewShift)
        s ← DeQueue(Shift)
    }
}
s ← m − min
while s ≤ m do                % NewShift3
{
    EnQueue(s, NewShift)
    s ← s + 1
}
```

Figure 6: Computation of $NewShift$ (Case 2).

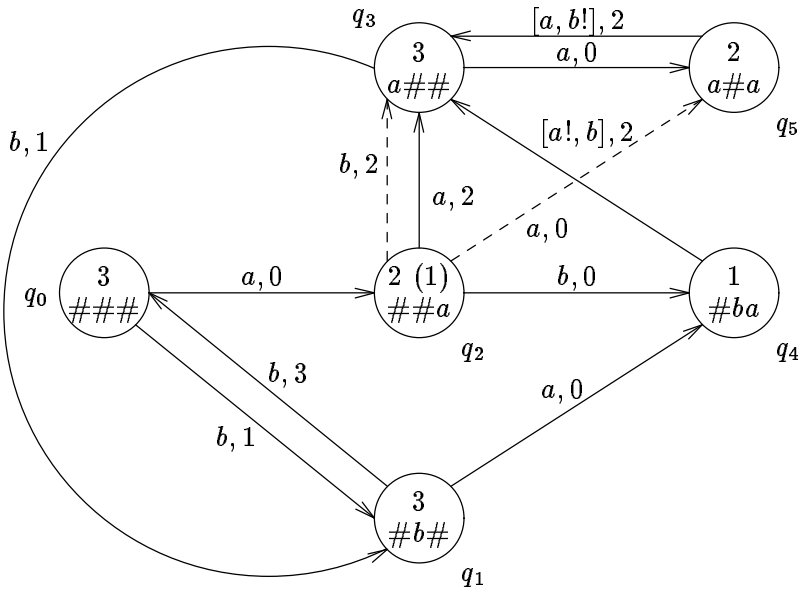

Figure 7: Optimal BMA (average case) for aba.