Exact Online Two-Dimensional Pattern Matching Using Multiple Pattern Matching Algorithms

CHARALAMPOS S. KOUZINOPOULOS and KONSTANTINOS G. MARGARITIS,
University of Macedonia

Baker and Bird and Baeza-Yates and Regnier are two of the most efficient and widely used algorithms for exact online two-dimensional pattern matching. Both use the automaton of the Aho-Corasick multiple pattern matching algorithm to locate all the occurrences of a two-dimensional pattern in a two-dimensional input string, a data structure that is considered by many as inefficient, especially when used to process long patterns or data using large alphabet sizes. This article presents variants of the Baker and Bird and the Baeza-Yates and Regnier algorithms that use the data structures of the Set Horspool, Wu-Manber, Set Backward Oracle Matching, and SOG multiple pattern matching algorithms in place of the automaton of Aho-Corasick and evaluates their performance experimentally in terms of preprocessing and searching time.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Algorithms, Two-dimensional pattern matching, multiple pattern matching, String Matching, Multiple Keyword Matching

ACM Reference Format:
Kouzinopoulos, C. S. and Margaritis, K. G. 2013. Exact online two-dimensional pattern matching using multiple pattern matching algorithms. ACM J. Exp. Algor. 18, 2, Article 2.4 (September 2013), 34 pages.
DOI: http://dx.doi.org/10.1145/2513148

1. INTRODUCTION

The expansion of imaging, graphics and multimedia often requires the use of string matching to higher than one dimension leading to the two- and multidimensional pattern matching problem were input strings and patterns of two or more dimensions are involved. The two-dimensional pattern matching problem can be defined as follows.

Definition. Given an input string $T = t_{0,0} \dots t_{n_1-1,n_2-1}$ *of size* $n_1 n_2$ *and a pattern* $P = p_{0,0} \dots p_{m_1-1,m_2-1}$ *of size* $m_1 m_2$ *over a finite character set* Σ*, the task is to find all occurrences of the pattern in the input string. More formally, find all pairs* i_1, i_2*, where* $0 \leq i_1 < n_1 - m_1 + 1$ *and* $0 \leq i_2 < n_2 - m_2 + 1$ *such that for all* j_1, j_2*, where* $0 \leq j_1 < m_1$ *and* $0 \leq j_2 < m_2$*, it holds that* $t_{i_1+j_1,i_2+j_2} = p_{j_1,j_2}$.

The 2D pattern matching problem has many applications, especially in image processing. It is used for content based information retrieval from image databases, image analysis, and medical diagnostics. It is also used by some methods of detecting edges, where a set of edge detectors is matched against a picture and by some OCR systems

Author's address: Parallel and Distributed Processing Laboratory, Department of Applied Informatics, University of Macedonia, 156 Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece; email: {ckouz, kmarg}@uom.gr.

DOI: http://dx.doi.org/10.1145/2513148

[Polcar and Melichar 2004]. The use of pattern matching is also important in areas like motion analysis, cartography, aerial photography, remote sensing, document analysis, and object tracking.

The solution to the two-dimensional 2D pattern matching problem differs if the algorithm used is online, where the input string is not known in advance and only the pattern is preprocessed, or offline, with both the pattern and the input string being preprocessed. This paper focuses on online algorithms and on the exact case where the pattern must be located as it is (i.e., without approximation) in the input string. The algorithms presented in this article can easily handle rectangular arrays where each dimension has a different size, but for the sake of simplicity only square arrays are considered for both input string and pattern, where $n_1 = n_2 = n$ and $m_1 = m_2 = m$. For data that involve irregular shapes or for problems that include approximation [Baeza-Yates and Perleberg 1992], rotation [Amir et al. 2006; Fredriksson et al. 2002], or scaling [Amir et al. 1992], different algorithms can be used.

Baker and Bird [Baker 1978; Bird 1977] and Baeza-Yates and Regnier [1993] are regarded to be among the most efficient two-dimensional pattern matching algorithms. They both handle the two-dimensional pattern matching problem as a special case of multiple pattern matching, where the two-dimensional pattern array is considered as a finite set of different patterns. To preprocess the pattern and locate all of its occurrences in the input string, the automaton of the Aho-Corasick algorithm [Aho and Corasick 1975] is used, a data structure that is considered by many as inefficient (i.e., Crochemore and Rytter [1994] refers to the efficiency of Aho-Corasick as not entirely satisfactory). This article presents efficient variants of the Baker and Bird [Baker 1978; Bird 1977] and the Baeza-Yates and Regnier [1993] algorithms that use the Set Horspool [Navarro and Raffinot 2002], Wu-Manber [Wu and Manber 1994], Set Backward Oracle Matching [Allauzen and Raffinot 1999] and Multipattern Shift-Or with Q-Grams (SOG) [Salmela et al. 2006] multiple pattern matching algorithms in place of the Aho-Corasick algorithm to perform two-dimensional pattern matching. The Karp-Rabin [Karp and Rabin 1987] and the Zhu and Takaoka [1989] algorithms were also considered but were omitted, as previous studies (i.e., Kouzinopoulos and Margaritis [2009]) showed that they were not as efficient for two-dimensional pattern matching when compared to Baker and Bird and Baeza-Yates and Regnier. The specific multiple pattern matching algorithms were chosen because they are efficient and are frequently encountered in other research papers. Set Horspool [Navarro and Raffinot 2002] is a classic algorithm that has a sublinear search phase in the average case. Commentz-Walter [1979], the multiple pattern matching algorithm upon which Set Horspool is based, is substantially faster in practice than Aho-Corasick, particularly when long patterns are involved [Watson 1995; Wu and Manber 1994]. Set Backward Oracle Matching also has a sublinear search phase in the average case. It appears to be very efficient when used on large pattern sets and has the same worst case complexity as Set Backward Dawg Matching but uses a much simpler automaton and is faster in all cases [Navarro and Raffinot 2002]. The Wu-Manber algorithm was chosen as it is considered a very fast algorithm in practice [Navarro and Raffinot 2002]. Finally, Salmela-Tarhio-Kytöjoki is a recently introduced family of algorithms that has a reportedly good performance on specific types of data [Kouzinopoulos and Margaritis 2010]. SOG has a linear search phase in the average case.

A survey of algorithms for exact, approximate, scaled and compressed two-dimensional pattern matching was presented in Amir [1992] and Crochemore and Rytter [2002]. The Baeza-Yates and Regnier [1993] algorithm was introduced in 1993, and its running time was compared to the Brute Force, Zhu and Takaoka and the Baker and Bird algorithms for a randomly generated pattern and input string with a binary alphabet. The pattern had a size of $m = 2$ to 20, while the input string had a size of

$n = 1{,}000$. For the specific problem parameters, it was shown that Baeza-Yates and Regnier was the fastest algorithm for all pattern sizes except for $m = 2$, where the Brute Force algorithm had the best performance. The same paper also suggested that the use of a multiple pattern matching algorithm based on Boyer-Moore [Boyer and Moore 1977] instead of Aho-Corasick should result in the improvement of the searching phase of the Baeza-Yates and Regnier algorithm. This motivated us to introduce the variants of the Baker and Bird and the Baeza-Yates and Regnier algorithms presented in this article. The Baker and Bird algorithm was detailed in Crochemore and Rytter [1994], while Tarhio [1996] introduced a two-dimensional pattern matching algorithm based on Boyer-Moore and compared it to the Naive and the Baeza-Yates and Regnier algorithms for a binary alphabet data set where the pattern had a size of $m = 2$ to 64 and the input string had a size of $n = 1{,}000$. In that article, it was shown that the Brute Force algorithm was the best for a pattern of a size up to $m = 5$ while for larger pattern sizes, the newly introduced algorithm was preferable. The subject of exact, scaled and approximate two-dimensional pattern matching was discussed in [Apostolico and Galil 1997]. The Baker and Bird and the Baeza-Yates and Regnier algorithms were detailed in [Tao 2005]. Finally, a variant of the Baker and Bird algorithm that uses solely finite automata was introduced by Zdarek [2010].

2. PATTERN MATCHING ALGORITHMS

Baker and Bird and Baeza-Yates and Regnier are two of the most well-known algorithms for exact online two-dimensional pattern matching. They treat each of the m rows of the pattern as a different pattern $p^0, p^1, \ldots, p^{m-1}$ from a finite pattern set P, turning in practice two-dimensional pattern matching into a multiple pattern matching problem. To preprocess the pattern and then scan the input string to locate all the positions where it occurs, Baeza-Yates and Regnier uses the Aho-Corasick algorithm, while Baker and Bird combines the Aho-Corasick and the Knuth-Morris-Pratt [Knuth et al. 1977] algorithms.

2.1. Aho-Corasick

Aho-Corasick is an extension of the Knuth-Morris-Pratt [Knuth et al. 1977] algorithm for a set of patterns P. It uses a deterministic finite state pattern matching machine; a rooted directed tree or *trie* of P with a *goto* function g and an additional *supply* function *Supply*. The *goto* function maps a pair consisting of an existing state q and a symbol character into the next state. It is a generalization of the *next* table or the *success* link of the Knuth-Morris-Pratt algorithm for a set of patterns where a parent state can lead to its child states by σ where σ is a matching character. Each state of the trie is labeled after a single character of a pattern $p^r \in P$. If $L(q)$ denotes the label of the path between the initial state and a state q, then $L(q)$ is also a prefix of one of the patterns. For each pattern p^r, there is a state q such that $L(q) = p^r$. This state is marked as terminal and when visited during the search phase indicates that a complete match of p^r was found. The *supply* function of Aho-Corasick is based on the *supply* function of the Knuth-Morris-Pratt algorithm. It is used to visit a previous state of the automaton when there is no transition from the current state to a child state via the *goto* function.

The *goto* function and the *supply* function are constructed during the preprocessing phase. To build the *goto* function, the trie is depth-first traversed and extended for each character of the patterns from a finite pattern set P at the same time the outgoing transitions to each state are created. The *supply* function is built in transversal order from the trie until it has been computed for all states. For each state q, the *supply* link can be determined based on the longest suffix of $L(q)$ that is also a prefix of *any* pattern from P. Assume that for the parent state q_{parent} of q, $g(q_{parent}, \sigma) = q$. If $Supply(q_{parent})$ also has an outgoing transition to a state h by σ, then the *supply* state of q can be set

to h. In any other case, $Supply(Supply(q_{parent}))$ must be checked for a transition to a state by σ and so on, until one such state is found or it is determined that no such state exists; in that case, the *supply* state of q is set to the initial state.

Let u_{i-1} be the longest suffix of the input string $t_0 \dots t_{i-1}$ that is also a prefix of *any* pattern $\in P$. The character σ located at position i of the input string is scanned next. If there is an outgoing transition from the current state q to another state f, as indicated by the *goto* function, then $L(f) = u_{i-1}\sigma$ is the new longest suffix of the input string at position i that is a prefix of one of the patterns. A match of a pattern exists in the input string if $|u_{i-1}\sigma| = m$. If, on the other hand $g(q, \sigma) = fail$, then $g(Supply(q), \sigma)$ is checked for an outgoing transition by σ. If $g(Supply(q), \sigma)$ leads to a state f', then $u_{i-1} = L(f')$. If $g(Supply(q), \sigma) = fail$, then $g(Supply(Supply(q)), \sigma)$ is considered and so on, until an outgoing transition by σ is found or until the supply state of the initial state is reached; in that case, the search will start again from the initial state. The construction of the *goto* function of the Aho-Corasick automaton is given in Algorithm 1, the computation of the *supply* function is presented in Algorithm 2, while the search phase of the Aho-Corasick algorithm is detailed in Algorithm 3. The output function returns $L(q)$ for each terminal state q and is denoted as $Output()$. A transition that does not point to a state is denoted as $fail$. An in-depth analysis of the Aho-Corasick algorithm is presented in Navarro and Raffinot [2002].

ALGORITHM 1: The construction of the *goto* function g of the Aho-Corasick automaton

```
Function AC_Preproc_Goto (p, m, Σ)
create state q_0
forall the α ∈ Σ do
  | g(q_0, α) := fail
end
for i := 0; i < m; i := i + 1 do
    j := 0; state := q_0
    while newState := g(state, p_j^i) ≠ fail do
      | state := newState; j := j + 1
    end
    for k := j; k < m; k := k + 1 do
        create state q_current
        forall the α ∈ Σ do
          | g(q_current, α) := fail
        end
        newState := q_current
        g(state, p_k^i) := newState
        state := newState
    end
    Output(q_current) := Output(q_current) ∪ {p^i}
    Add terminal state on q_current
end
```

The *goto* function can be implemented using any of the following data structures: an array of size $|\Sigma|$, where each state has an outgoing transition for every character of the alphabet by precomputing all the transitions simulated by the *supply* function [Navarro and Raffinot 2002]; a linked list that is space efficient but not time efficient; or a balanced search tree that is considered as a heavy-duty compromise and often not practical [Dori and Landau 2006]. The implementation used for the experiments of this article was based on code from the Streamline system I/O software layer [Streamline

ALGORITHM 2: The construction of the *supply* function *Supply* of the Aho-Corasick automaton

```
Function AC_Preproc_Supply (Σ)
forall the α ∈ Σ do
    if g(q_0, α) = fail then
        g(q_0, α) := q_0
    else
        Supply(g(q_0, α)) := q_0
    end
end
forall the currentState ∈ trie states in transversal order do
    forall the α ∈ Σ do
        s := g(currentState, α)
        if s ≠ fail then
            state := Supply(currentState)
            while g(state, α) = fail do
                state := Supply(state)
            end
            Supply(s) := g(state, α)
        end
    end
end
```

ALGORITHM 3: The search phase of the Aho-Corasick automaton

```
Function AC_Search (t, m, n)
state := q_0
for i := 0; i < n; i := i + 1 do
    while newState := g(state, t_i) = fail do
        state := Supply(state)
    end
    state := newState
    if Output(state) is not empty then
        report match at i − m + 1
    end
end
```

2012]. It uses a linked list for the *supply* function and a linked list of arrays to represent the transitions of the *goto* function with each cell of the arrays potentially containing a pointer to the next node. Each node of the list corresponds to a different state of the trie and has an array of size $|\Sigma|$ with an outgoing transition for every character of Σ. The trie of P can then be built for all m patterns in $\mathcal{O}(|\Sigma|m^2)$ time, with a total size of $\mathcal{O}(|\Sigma|m^2)$. The time to pass through a transition of the *goto* function is $\mathcal{O}(1)$ in the worst and average case, while the search phase has a cost of $\mathcal{O}(n)$ in the worst and average case.

2.2. Baker and Bird

Baker and Bird utilizes the Aho-Corasick and Knuth-Morris-Pratt algorithms to perform two-dimensional pattern matching in worst- and average-case linear time. The general idea behind the algorithm is to construct the trie of Aho-Corasick from the pattern rows and then assign an index to each distinct row. During the preprocessing phase of the algorithm, the trie of the Aho-Corasick algorithm is built from each pattern row $p^0, p^1, \ldots, p^{m-1} \in P$ and a unique index is assigned to each terminal state q. Consequently, the two-dimensional pattern P can be reduced to a one-dimensional array

P' of indices. Array P' is then preprocessed using the Knuth-Morris-Pratt algorithm so that the longest *border* v of each suffix of P' can be computed. This information is stored in a *next* table.

To locate all the occurrences of the pattern in the input string, two distinct steps are used: *row-matching* and *column-matching*. During the row-matching step, the input string is scanned horizontally. For each position j, k, it is determined the state of the Aho-Corasick trie that corresponds to the longest suffix u_k of characters $t_{j,0} \dots t_{j,k}$. If that state is terminal, then a pattern row p^r matches with u_k. For every such position, it must be determined if pattern rows $p^0, \dots, p^{r-1}$ also occur at positions $t_{j-r,k-m+1} \dots t_{j-r,k}, \dots, t_{j-1,k-m+1} \dots t_{j-1,k}$. This is done in the column-matching step with the use of the Knuth-Morris-Pratt algorithm vertically. For this purpose, a table a of size n is maintained. The table is used to store values for each of the n columns of the input string such that when $a[k] = r$, pattern rows $0, \dots, r-1$ exist immediately above the current. If $a[k] = r$ and the suffix of the input string at position j, k matches with pattern row p^r, then $a[k]$ is set to $r+1$. If the input string suffix does not match with p^r, $a[k]$ is set to $s+1$, where s is the longest border of $P'_0, \dots, P'_{r-1}$, as determined based on the values of the *next* table. A complete match of the pattern is found with its top left corner at position $j-m+1, k-m+1$ of the input string when $a[k] \geq m-1$ at the end of the column-matching step. If the row-matching step does not locate an occurrence of any pattern row at a column k of the input string, then $a[k]$ is set to 0. Algorithms 4 and 5 depict the row-matching and column-matching steps of the search phase of Baker and Bird. The implementation used for the experiments of this article is based on code from Tucker [2004].

The Baker and Bird algorithm uses $\mathcal{O}(n+|\Sigma|m^2)$ extra space to store tables a and P', and the trie of the Aho-Corasick algorithm. The trie is built in $\mathcal{O}(|\Sigma|m^2)$ time during the preprocessing phase, as detailed previously. Array P' is preprocessed using the Knuth-Morris-Pratt algorithm in $\mathcal{O}(m)$ time. The algorithm requires an $\mathcal{O}(n^2)$ theoretical searching time in the worst and average case to scan the input string.

Consider the following example: Pattern P is used during the preprocessing phase to create the trie of the Aho-Corasick algorithm.

ALGORITHM 4: The row-matching step of the Baker and Bird algorithm

```
Function BB_Row_Matching (p, t, m, n)
head := q_0
for k := 0; k < n; k := k + 1 do
    a[k] := 0
end
for j := 0; j < n; j := j + 1 do
    r := head
    for k := 0; k < n; k := k + 1 do
        while (s := g(r, t_{j,k})) = fail do
            r := Supply(r)
        end
        r := s
        if Output(r) is not empty then
            BB_Column_Matching (p, a, k, j, r, m)
        else
            a[k] := 0
        end
    end
end
```

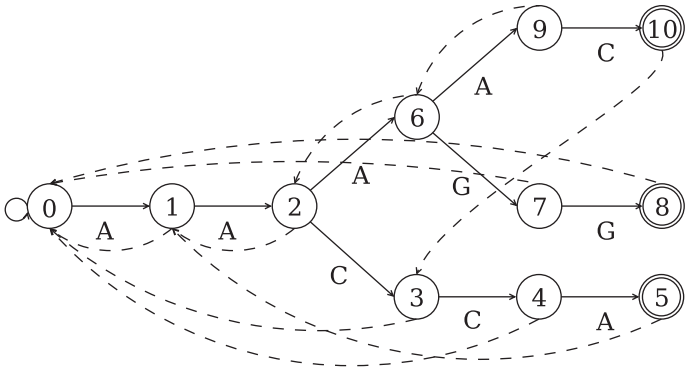

Fig. 1. The automaton of the Aho-Corasick algorithm for the example pattern P.

ALGORITHM 5: The column-matching step of the Baker and Bird algorithm

```
Function BB_Column_Matching (p, a, k, j, r, m)
i := a[k]
while i > 0 AND p^i_{0...m-1} ≠ Output(r) do
 |  i := next[i]
end
if i + 1 < m then
 |  a[k] := i + 1
else
 |  a[k] := 0
end
if i >= m − 1 then
 |  report match at j − m + 1, k − m + 1
end
```

$$P = \begin{array}{|c|c|c|c|c|}\hline A & A & C & C & A \\ \hline A & A & A & G & G \\ \hline A & A & C & C & A \\ \hline A & A & A & G & G \\ \hline A & A & A & A & C \\ \hline \end{array}$$

As shown in Figure 1, $L(5) = p^0$, $L(8) = p^1$, and $L(10) = p^4$. Index 0 can be assigned to pattern p^0, index 1 can be assigned to pattern p^2, and index 2 can be assigned to pattern p^4. Note that $p^2 = p^0$ and $p^3 = p^1$ and, therefore, they share the same index. Hence, the two-dimensional pattern P can be reduced to a one-dimensional array of indices P'.

$$P' = \begin{array}{|c|}\hline 0 \\ \hline 1 \\ \hline 0 \\ \hline 1 \\ \hline 2 \\ \hline \end{array}$$

The array P' is then preprocessed using the Knuth-Morris-Pratt algorithm to compute the *next* table:

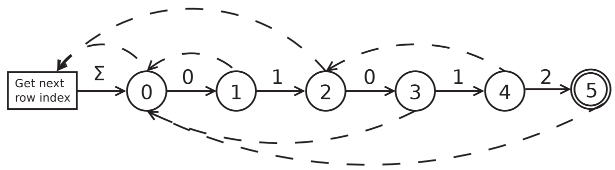

Fig. 2. The automaton of the Knuth-Morris-Pratt algorithm for the row indices of P'.

i =	0	1	2	3	4	5
P' =	0	1	0	1	2	
next[i] =	−1	0	−1	0	2	0

The *next* table can also be represented as a finite automaton, as depicted in Figure 2. State 0 is the initial state, each character of P' labels the path between two different states of the automaton while $L(q)$ is the label of the path between the initial and a given state q. The solid lines represent the *success* links that are used to visit the next node of the automaton after a successful match of a pattern character, similar to the *goto* function of the Aho-Corasick algorithm. The dashed lines represent the *supply* links that are used to return to a previous state of the automaton after a mismatch between a pattern character and an input string character.

During the row-matching step, each row of the input string is scanned horizontally, starting from $t_{0,0}$. At each position, j, k is determined the state of the trie that corresponds to the longest suffix of $t_{j,0} \ldots t_{j,k}$. If this state is terminal, the index that corresponds to the matching pattern row is assigned to the suffix of the input string. In any other case, −1 is assigned. Subsequently, the input string T can be converted to T'.

$T =$

A	A	C	C	A	A	A	C	C
A	A	A	G	G	A	A	A	G
A	A	C	C	A	A	A	C	C
A	A	A	G	G	A	A	A	G
A	A	C	C	A	A	A	C	C
A	A	A	G	G	A	A	A	G
A	A	C	C	A	A	A	C	C
A	A	A	G	G	A	A	A	G
A	A	A	A	C	A	A	A	A

$T' =$

−1	−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	1	−1	−1	−1	−1
−1	−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	1	−1	−1	−1	−1
−1	−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	1	−1	−1	−1	−1
−1	−1	−1	−1	0	−1	−1	−1	−1
−1	−1	−1	−1	1	−1	−1	−1	−1
−1	−1	−1	−1	2	−1	−1	−1	−1

Note that the index for each suffix of the input string is calculated online, and thus table T' is not stored in memory. At position $t_{0,4}$ of the input string, the Baker and Bird algorithm determines that a pattern row occurs as a suffix of $t_{0,0} \ldots t_{0,4}$ because index 0 was encountered. As the value of $a[4]$ was initially set to 0 and there is an occurrence of any pattern row, the value of $a[4]$ is incremented to 1. By using the *goto* and the *supply* functions of the trie, the indices for the subsequent input string positions are calculated until index 1 is encountered at position $t_{1,4}$. Since the value of $a[k]$ was previously set to 1 and p^1 is equal to the m-character suffix of $t_{1,0} \ldots t_{1,4}$, the value of $a[4]$ is incremented to 2. This procedure is repeated for positions $t_{2,4}$ and $t_{3,4}$ of the input string where the value of $a[4]$ is set to 3 and 4, respectively. At position $t_{4,4}$, index 0 is encountered, but p^4 does not match with the suffix of $t_{4,0} \ldots t_{4,4}$. Based on the information of the *next* table, the longest border v of P'_5 is $P'_0 P'_1$; therefore, P'_2 can be aligned with the input string at position $t_{4,4}$. The algorithm continues until it is determined that a complete

match of the pattern occurs ending at position $t_{8,4}$ of the input string. The values of table a for column 4 of the specific input string will be the following.

$$a[4] = \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline 4 \\ \hline 3 \\ \hline 4 \\ \hline 3 \\ \hline 4 \\ \hline 5 \\ \hline \end{array}$$

Since the value of $a[4] = 5$ at row 8 of the input string, a complete match of P is reported at $t_{4,0} \dots t_{4,4}, \dots, t_{8,0} \dots t_{8,4}$.

2.3. Baeza-Yates and Regnier

The Baeza-Yates and Regnier algorithm is similar to the Baker and Bird algorithm. It uses the Aho-Corasick algorithm to create a trie from the pattern rows and assign an index to each distinct row. The input string is then scanned horizontally for the occurrences of the pattern indices. A significant difference though is that during the search phase only $\lfloor \frac{n}{m} \rfloor$ primary rows of the input string are scanned, since they cover all possible positions where a pattern row may occur. If a match is found on a primary row, then m characters on each of the $m-1$ secondary rows are also scanned for the pattern indices using the Aho-Corasick algorithm, to determine if a complete match occurs.

During the preprocessing phase of the algorithm, the trie of the Aho-Corasick algorithm is built from each pattern row $p^0, p^1, \dots, p^{m-1} \in P$ and a unique index is assigned to each terminal state q. Consequently, the two-dimensional pattern P can be reduced to a one-dimensional array P' of indices. The indices assigned to the terminal states are accessible through an $Id()$ function.

For each position j, k of a primary row of the input string, the algorithm uses the Aho-Corasick trie to determine the state that corresponds to the longest suffix u_k of $t_{j,0} \dots t_{j,k}$, as depicted in Algorithm 6. Note that the search of the primary rows of the input string is identical to the row-matching step of the Baker and Bird algorithm, as shown in Algorithm 4, with the difference that it is performed on only $\lfloor \frac{n}{m} \rfloor$ rows. If this state is terminal, pattern row p^r occurs in the input string at that position. Then it must be determined if pattern rows $p^0, \dots, p^{r-1}$ occur at positions $t_{j-r,k-m+1} \dots t_{j-r,k}, \dots, t_{j-1,k-m+1} \dots t_{j-1,k}$ and $p^{r+1}, \dots, p^{m-1}$ at positions $t_{j+1,k-m+1} \dots t_{j+1,k}, \dots, t_{j+m-r-1,k-m+1} \dots t_{j+m-r-1,k}$ of the input string using again the Aho-Corasick trie, as presented in Algorithm 7. When two or more rows of the pattern are identical (e.g., $p^r = p^l$), scanning on the secondary rows of the input string should be repeated for each of the identical rows. In the worst case (where $p^0 = \dots = p^{m-1}$), $2m-1$ possible rows must be scanned (the primary row, $m-1$ secondary rows above and $m-1$ secondary rows below the primary). In that case, the search on each position of the input string would be $\mathcal{O}(m^3)$ for a total cost of $\mathcal{O}(n^2m^2)$ for the algorithm. To improve the efficiency of the vertical search, a table b of size $2m-1$ is used to store and reuse the indices of the primary and the $2m-2$ secondary rows. That way, the input string is scanned only for the occurrence of the distinct rows of the pattern, as can be seen in Algorithm 8.

ALGORITHM 6: The search of the Baeza-Yates and Regnier algorithm on the primary rows

```
Function BYR_Search_Primary_Rows (t, m, n)
head := q_0
for j := m − 1; j < n; j := j + m do
    r := head
    for k := 0; k < n; k := k + 1 do
        while (s := g(r, t_{j,k})) = fail do
            r := Supply(r)
        end
        r := s
        if Output(r) is not empty then
            if P'_{m−1} = m − 1 then
                BYR_Search_Secondary_Rows_Unique (Id(r), j, k − m + 1, t, m, n)
            else
                BYR_Search_Secondary_Rows_Duplicate (Id(r), j, k − m + 1, t, m, n)
            end
        end
    end
end
```

ALGORITHM 7: The search of the Baeza-Yates and Regnier algorithm on the secondary rows when all pattern rows are unique

```
Function BYR_Search_Secondary_Rows_Unique (index, j, k, t, m, n)
head := q_0
for tt := j − index, pp := 0; pp < m; tt := tt + 1, pp := pp + 1 do
    if pp = index then
        continue
    end
    r := head
    for c := k; c ≤ k + m − 1; c := c + 1 do
        while (s := g(r, t_{tt,c})) = fail do
            r := Supply(r)
        end
        r := s
    end
    if Id(r) ≠ P'_{pp} then
        return
    end
    report match at j − index + 1, k
end
```

The Baeza-Yates and Regnier algorithm uses $\mathcal{O}(|\Sigma|m^2)$ extra space to store the trie of the Aho-Corasick algorithm, $\mathcal{O}(m)$ extra space for table b and $\mathcal{O}(m)$ extra space for array P'. The trie is built in $\mathcal{O}(|\Sigma|m^2)$ time during the preprocessing phase. The algorithms uses $\mathcal{O}(\frac{n^2}{m})$ time in the worst case to scan all characters of the primary rows and $\mathcal{O}(m^2)$ for all characters of the secondary rows for a searching cost of $\mathcal{O}(n^2m)$. On average, the probability that a row of the pattern is matched at a given position of the input string is given by $\frac{m}{|\Sigma|^m}$ with an upper bound of $\frac{n^2}{m}$ for the scanning of the primary rows and a total search phase cost of $\frac{n^2m^2}{|\Sigma|^m}$. Since $\frac{m^2}{|\Sigma|^m} < \frac{4}{m}$ for each $m \geq 1$ and $|\Sigma| \geq 2$, the search phase complexity of the Baeza-Yates and Regnier algorithm is $\mathcal{O}(\frac{n^2}{m})$ in the average case. This

ALGORITHM 8: The search of the Baeza-Yates and Regnier algorithm on the secondary rows when duplicate pattern rows exist

```
Function BYR_Search_Secondary_Rows_Duplicate (index, j, k, t, m, n)
head := q_0
for i := 0; i < 2m − 1; i := i + 1 do
    b[i] := −1
end
b[m − 1] := index
for i := 0; i < m; i := i + 1 do
    if P'_i ≠ index then
        continue
    end
    for d := j − i, w := m − 1 − i; d ≤ j − i + m − 1 AND d < n; d := d + 1, w := w + 1 do
        if b[w] = P'_{d−j+i} then
            continue
        end
        r := head
        for c := k; c ≤ k + m − 1; c := c + 1 do
            while (s := g(r, t_{d,c})) = fail do
                r := Supply(r)
            end
            r := s
        end
        b[w] := Output(r)
        if b[w] ≠ P'_{d−j+i} then
            break
        end
    end
    if d = j − i + m then
        report match at d − m, k
    end
end
```

article uses an implementation based on code from the Xaa project by Baeza-Yates and Fuentes [1996].

Consider the following example with the same pattern P as the one used in Figure 1. As shown in Figure 1, $L(5) = p^0$, $L(8) = p^1$ and $L(10) = p^4$. Index 0 can be assigned to pattern p^0, index 1 can be assigned to pattern p^2 and index 2 can be assigned to pattern p^4. Remember that since $p^2 = p^0$ and $p^3 = p^1$, they share the same index. Hence, the two-dimensional pattern P can be reduced to a one-dimensional array of indices P' and subsequently the input string T can be converted to T'. As with the Baker and Bird example, the index for each suffix of the input string is calculated online and, therefore, table T' is not stored in memory. The values of P, P', T and T' are identical to the respective tables given in Section 2.2.

The Baeza-Yates and Regnier algorithm starts scanning from position $t_{4,0}$ of the input string. At $t_{4,4}$, index 0 is encountered. Pattern rows p^0 and p^2 share the same index because they are identical, and thus table b is used. That way, it is ensured that pattern rows will be compared to substrings of the input string at most $2m - 1$ times. The initial value of $b[4]$ is set to the index of the matching pattern row. Pattern row p^0 is aligned next with the 4^{th} row of the input string. In that case, the indices of the input string suffixes $t_{4,0} \ldots t_{4,4}, \ldots, t_{8,0} \ldots t_{8,4}$ are calculated and compared with the indices of pattern rows p^0 to p^4. Since $b[4] = P'[0]$, the index of the suffix of the input string at positions $t_{4,0} \ldots t_{4,4}$ can be reused. The indices of the input string suffixes at positions

$t_{5,0} \dots t_{5,4}, \dots, t_{8,0} \dots t_{8,4}$ are calculated and stored in $b[5], \dots, b[8]$. Since all indices of b now match with the indices stored in P', it is determined that a complete match occurs.

$$b = \begin{array}{|c|} \hline -1 \\ \hline -1 \\ \hline -1 \\ \hline -1 \\ \hline 0 \\ \hline -1 \\ \hline -1 \\ \hline -1 \\ \hline -1 \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline & 0 \\ \hline & 1 \\ \hline & 0 \\ \hline & 1 \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline 2 & 2 \\ \hline \end{array}$$

Pattern row p^2 is then aligned with the fifth row of the input string. The indices of the input string suffixes $t_{2,0} \dots t_{2,4}, \dots, t_{6,0} \dots t_{6,4}$ are calculated and compared with the indices of pattern rows p^0 to p^4. The indices of input string suffixes $t_{2,0} \dots t_{2,4}$ and $t_{3,0} \dots t_{3,4}$ were not determined before and are, therefore, calculated and stored in $b[2]$ and $b[3]$. The values of $b[4]$ and $b[5]$ were calculated in the previous step, and because $b[4] = P'[2]$ and $b[5] = P'[3]$, the indices of input string suffixes $t_{4,0} \dots t_{4,4}$ and $t_{5,0} \dots t_{5,4}$ can be reused. The value of $b[6]$ was previously set but is different from $P'[4]$; therefore, is calculated again using the trie of Aho-Corasick. Since the new value of $b[6]$ is equal to 0 and $P'[4] = 2$, it is determined that a mismatch occurs between the pattern and the input string:

$$b = \begin{array}{|c|} \hline -1 \\ \hline -1 \\ \hline -1 \\ \hline -1 \\ \hline 0 \\ \hline 1 \\ \hline 0 \\ \hline 1 \\ \hline 2 \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline & 0 \\ \hline & 1 \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline 2 & 0 \\ \hline & 1 \\ \hline & 2 \\ \hline \end{array}$$

3. ALGORITHM VARIANTS

This section briefly presents the Set Horspool, Set Backward Oracle Matching, Wu-Manber and the SOG multiple pattern matching algorithms, discusses implementation details for the variants of the Baker and Bird and the Baeza-Yates and Regnier two-dimensional pattern matching algorithms and highlights their differences to the original implementations. The analysis of the algorithms assumes that the pattern set consists of m patterns where each pattern has a length of m characters for a total size of m^2 characters.

Table I summarizes the theoretical extra space, preprocessing, worst and average time complexity of the multiple pattern matching algorithms used.

Experimental results on multiple pattern matching algorithms have been reported in the past. The performance of a number of algorithms including Aho-Corasick, Set Horspool, Set Backward Oracle Matching, and Wu-Manber was evaluated in Navarro and Raffinot [2002] for a randomly generated data set. A variant of the Wu-Manber algorithm called QWM was presented in Dong-hong et al. [2006], and its performance was compared to Aho-Corasick, Commentz-Walter, and the original Wu-Manber algorithm

Table I. Known Theoretical Extra Space, Preprocessing, Worst and Average Searching Time Complexity of Multiple Pattern Matching Algorithms

Algorithm	Extra space	Preprocessing	Worst case	Average case
Aho Corasick	$\|\Sigma\|m^2$	$\|\Sigma\|m^2$	n	n
Set Horspool	$\|\Sigma\|m^2$	$\|\Sigma\|m^2$	nm	*sublinear*
Set Backward Oracle Matching	$\|\Sigma\|m^2$	$\|\Sigma\|m^2$	nm^2	*sublinear*
Wu-Manber	$m \times \sum_{i=0}^{B-1} \|\Sigma\| \times (2^{bitshift})^i$	m^2	$nm^2 \log_{\|\Sigma\|} m^2$	$\frac{n \log_{\|\Sigma\|} m^2}{m}$
SOG	$\|\Sigma\|^B + m^2$	m^2	nm^2	n

for randomly generated data with a binary alphabet and an alphabet of size $|\Sigma| = 4$ as well as for data with an English and a Chinese language alphabet. HMA, a hierarchical multiple pattern matching algorithm was introduced in Sheu et al. [2008], and its performance was compared among others to the performance of a compressed version of the Aho-Corasick algorithm on data for intrusion detection systems. The Aho-Corasick, Set Horspool, Set Backward Oracle Matching, and Wu-Manber algorithms were compared in Salmela et al. [2006] in terms of searching time for biological sequence databases and random input strings for sets consisting of 100 to 100,000 patterns. Each pattern had a length of $m = 8$ and 32 characters, while the input string had a size of approximately 32MB. Finally, a performance study of the Commentz-Walter, Wu-Manber, Set Backward Oracle Matching, and the Salmela-Tarhio-Kytöjoki algorithms for biological sequence databases was presented in Kouzinopoulos and Margaritis [2011].

3.1. Two-Dimensional Pattern Matching Using Set Horspool

Set Horspool combines a deterministic finite state pattern matching machine with the *shift* function of the Horspool [1980] algorithm to search for the occurrence of multiple patterns in the input string in sublinear time, on average. The pattern matching machine used is a trie with a *goto* function, created from each pattern of the pattern set in reverse. The search for the occurrences of the patterns is then performed backwards, similar to Horspool. When a mismatch or a complete match occurs, a number of input string positions can be safely skipped based on the bad character shift of the Horspool algorithm generalized for a set of patterns. The bad character shift is calculated for each different character $\sigma \in \Sigma$ as the distance of the rightmost occurrence of σ in any pattern to the end of that pattern and is stored in a table bc of size $|\Sigma|$. If no such character exists, the corresponding bad character shift is set to m. The construction of the bad character shift is given in Algorithm 9.

ALGORITHM 9: The construction of the bad character shift of the Set Horspool algorithm

```
Function SH_Bad_Character_Shift (P, m, |Σ|)
for i := 0; i < |Σ|; i := i + 1 do
  | bc(i) := m
end
for i := 0; i < m; i := i + 1 do
  | for j := 0; j < m − 1; j := j + 1 do
  |   | bc(p_j^i) := MIN(m − j − 1, bc(p_j^i))
  | end
end
```

Scanning the input string for the occurrences of the patterns is performed backwards, starting from character t_{m-1}. For each position i of the input string, the

algorithm computes the longest suffix u_i of $t_0 \dots t_i$ that is also a suffix of any pattern. When a complete match is found or a mismatch occurs when reading character σ of the input string, the trie is shifted to the right according to character β at position i of the input string until β is aligned with the next state of the trie that is labeled after β. The implementation of Set Horspool uses a linked list of arrays to represent the transitions of the *goto* function with each cell of the arrays potentially containing a pointer to the next node. Each node of the list corresponds to a different state of the trie and has an array of size $|\Sigma|$ with an outgoing transition for every character of Σ with $\mathcal{O}(1)$ time in the worst and average case to pass through a transition of the *goto* function. The construction of the trie and the shift function requires $\mathcal{O}(|\Sigma|m^2)$ time and space, whereas the search phase of the algorithm is $\mathcal{O}(nm)$ worst-case time or is sublinear, on average.

Using the Set Horspool trie instead of the Aho-Corasick trie in the Baker and Bird algorithm is fairly straightforward. During the preprocessing phase, the trie is built from each pattern row $p^0, p^1, \dots, p^{m-1} \in P$ in reverse and a unique index is assigned to each terminal state. Similar to the original implementation, the indices are used by the Knuth-Morris-Pratt algorithm to create a *next* table that is then utilized during the column-matching step of Baker and Bird. Moreover, the bad character shift is computed for each different character $\sigma \in \Sigma$. In the row-matching step, each row j of the input string is scanned backwards from character $t_{j,m-1}$ to $t_{j,n-1}$. For each position k of the input string row, the algorithm computes the longest suffix u of $t_{j,0} \dots t_{j,k}$ that is also a suffix of any pattern. When a complete match is found or a mismatch occurs between character σ of the input string and α of the trie, the trie is shifted to the right according to character β at position k of the input string until β is aligned with the next state of the trie that is labeled after β. If no such β exists, the trie is shifted to the right by m positions. The search phase of the Set Horspool variant of the Baker and Bird algorithm is $\mathcal{O}(n^2m)$ in the worst case. Since, on average, the searching phase of the Set Horspool algorithm is sublinear when used for multiple pattern matching, it is expected for the Set Horspool variant of the Baker and Bird algorithm to be $\mathcal{O}(Kn)$, where K a parameter such that $K < n$.

Replacing the Aho-Corasick trie with the Set Horspool trie in the Baeza-Yates and Regnier algorithm is also straightforward. The trie is built during the preprocessing phase from each pattern row $p^0, p^1, \dots, p^{m-1} \in P$ in reverse and the two-dimensional pattern P is reduced to a one-dimensional array P' of indices. Additionally, the bad character shift is computed for each different character $\sigma \in \Sigma$. The scanning on each primary row j of the input string is performed backwards, starting from character $t_{j,m-1}$. Then, for each position k of the primary row, the Set Horspool trie is used to determine the longest suffix u of $t_{j,0} \dots t_{j,k}$ that is also a suffix of any pattern from the pattern set. When a complete match or a mismatch occurs, the trie is shifted to the right by a number of positions based on the bad character shift function. If pattern row p^r occurs in a primary row of the input string, the trie of Set Horspool is also used backwards in the secondary rows to determine if pattern rows $p^0, \dots, p^{r-1}$ and $p^{r+1}, \dots, p^{m-1}$ occur immediately above and below the current row. The search phase of the Set Horspool variant of the Baeza-Yates and Regnier algorithm is $\mathcal{O}(n^2m^2)$ in the worst case and $\mathcal{O}(\frac{Kn}{m})$ in the average case.

3.2. Two-Dimensional Pattern Matching Using Set Backward Oracle Matching

The Set Backward Oracle Matching algorithm extends the Backward Oracle Matching [Allauzen et al. 1999] string matching algorithm to search for the occurrence of multiple patterns in the input string in sublinear time, on average. It uses a factor oracle, a deterministic acyclic automaton created from each pattern $p^r \in P$ in reverse that

is based on the notion of weak factor recognition. The automaton consists of a *goto* function g and at most m^2 additional external transitions such that at least any factor of a pattern can be recognized. The *goto* function is constructed during the preprocessing phase from the set of the reversed patterns. For each character σ at position i of a pattern p^r, the trie is depth-first traversed. If u is the suffix $p^r_{i+1} \dots p^r_{m-1}$ of a pattern p^r and σu does not exist as a label $L(q)$ of a path of the trie, then the trie is extended; a new state q is created and is labeled by σ and at the same time the outgoing transitions to q are constructed from the states at all levels between the initial and q. To construct the external transitions, a *supply* function *Supply* is used to specify a *supply* state for each state q with the *supply* state of the initial state being set to $\varnothing$. Assume that the *goto* and *supply* functions for all states up to the parent state q_{parent} of q were already computed and that $g(q, \sigma) = h$. Then, the *supply* state of q is visited to determine if there is an outgoing transition by σ. If there is no such transition, then $g(Supply(q), \sigma)$ is set to h as an external transition. In that case, $Supply(Supply(q))$ is visited next and so on, until a *supply* state with an outgoing transition by σ is found or until there are no more states to visit.

During the search phase, the algorithm reads backwards the longest suffix u of the input string that is also a suffix of any pattern. When a mismatch occurs at position k of the input string, the oracle can be safely shifted past k. If a terminal state is reached, a match of some pattern in the input string has potentially been found since there could be terminal states in the oracle that do not correspond to any pattern. Additionally, terminal states could exist that correspond to more than one pattern. For this reason, each terminal state q holds a set of indices $F(q)$ to the patterns it corresponds. Then, all the patterns in $F(q)$ are compared directly with the input string to determine if a complete match is found and the factor oracle is shifted by one position to the right. The implementation of Set Backward Oracle Matching uses a linked list of arrays to represent the transitions of the *goto* function with each cell of the arrays potentially containing a pointer to a different node. Each node of the list corresponds to a different state of the trie and has an array of size $|\Sigma|$ with an outgoing transition for every character of Σ with $\mathcal{O}(1)$ time in the worst and average case to pass through a transition of the *goto* function. The set of indices F is stored to each terminal state using an array of size m because, in the worst case, a terminal state could correspond to all m patterns. An auxiliary table *aux* of size m is also maintained; the inverse of array P' that maps indices to the patterns they correspond. The oracle is created during the preprocessing phase in $\mathcal{O}(|\Sigma|m^2)$ for all m patterns of the pattern set using a size of $\mathcal{O}(|\Sigma|m^2)$. The search phase complexity of the algorithm is $\mathcal{O}(nm^2)$ in worst-case time or sublinear in average time.

To adapt the Baker and Bird algorithm so that it can use the factor oracle of the Set Backward Oracle Matching algorithm, it must be taken under consideration that a terminal state q could correspond to no pattern or to multiple patterns at once. During the preprocessing phase, the factor oracle is created from the patterns in P in reverse and a unique index is assigned to each distinct pattern p^r. The indices are processed by the Knuth-Morris-Pratt algorithm to create a *next* table. In the row-matching step, each row j of the input string is scanned backwards, starting from character $t_{j,m-1}$. If during row-matching a terminal state q of the oracle is reached at position j, k of the input string, all patterns that $F(q)$ points to are compared directly with the input string to determine the index to be assigned to the suffix of $t_{j,0} \dots t_{j,k}$ that corresponds to the matching pattern and the oracle is shifted by one position to the right. If no pattern corresponds to the suffix of the input string or a mismatch occurs between a character σ of the input string and α of the oracle at column k, the factor oracle is shifted past k. The search phase of the Set Backward Oracle Matching

variant of the Baker and Bird algorithm is $\mathcal{O}(n^2m^2)$ in the worst case and $\mathcal{O}(Kn)$ in the average case.

The Baeza-Yates and Regnier algorithm was implemented using the Set Backward Oracle Matching algorithm as follows. During the preprocessing phase, the factor oracle is constructed from the pattern rows in reverse and the two-dimensional pattern is reduced to a one-dimensional array of indices. For each position k of a primary row j of the input string, the factor oracle is used to read backwards the longest suffix u of $t_{j,0} \dots t_{j,k}$ that is also a suffix of a pattern row. When a mismatch occurs at position j, k, the oracle can be safely skipped to the right past k. If a pattern row is matched in a primary row, then the secondary rows are also scanned backwards using the factor oracle to determine if a complete match occurs and the factor oracle is then shifted by one position to the right. Since a terminal state q of the oracle could exist that does not correspond to any pattern or that corresponds to more than one pattern, any potential matches in the primary or the secondary rows must be verified directly with the input string using the set of indices $F(q)$, similar to the Set Backward Oracle Matching implementation of the Baker and Bird algorithm. The theoretical searching time complexity of the Set Backward Oracle Matching implementation of the Baeza-Yates and Regnier algorithm is $\mathcal{O}(n^2m^3)$ in the worst case and $\mathcal{O}(\frac{Kn}{m})$ in the average case.

3.3. Two-Dimensional Pattern Matching Using Wu-Manber

Wu-Manber is a generalization of the Horspool algorithm for multiple pattern matching. It scans the characters of the input string backwards for the occurrences of the patterns, shifting the search window to the right when a mismatch or a complete match occurs. To perform the shift, the bad character shift function of the Horspool algorithm is used. As previously detailed, the bad character shift for a character σ determines the safe number of shifts based on the position of the rightmost occurrence of σ in any pattern. The probability of σ existing in one of the patterns increases in the size of the pattern set and is inversely proportional to the alphabet size and thus the maximum possible shift is decreased. To improve the efficiency of the algorithm, Wu-Manber considers the characters of the patterns and the input string as blocks of size B instead of single characters, essentially enlarging the alphabet size to $|\Sigma|^B$.

During the preprocessing phase, three tables are built from the patterns, the *SHIFT*, *HASH* and *PREFIX* tables. *SHIFT* is the equivalent of the bad character shift of the Horspool algorithm for blocks of characters, generalized for multiple patterns. If B does not appear in any pattern, the search window can be safely shifted by $m - B + 1$ positions to the right. Let h be the hash value of a block of B characters as determined by a hash function $h_1()$. Then, $SHIFT[h]$ is the distance of the rightmost occurrence of B to the end of *any* pattern. The *HASH* and *PREFIX* tables are only used when the shift value stored in $SHIFT[h]$ is equal to 0. *HASH[h]* contains an ordered list of pattern indices whose B-character suffix has a hash value of h. For each of these patterns, let h' be the hash value of their B'-character prefix as determined by a hash function $h_2()$. The hash value h' for each pattern p is stored in *PREFIX[p]*. That way, a potential match of the B-character suffix of a pattern can be verified first with the B'-character prefix of the pattern before comparing the patterns directly with the input string. The complexity of Wu-Manber was not given in the original paper, since hash functions $h_1()$ and $h_2()$ were not specified and the size of the *SHIFT*, *HASH* and *PREFIX* tables was not given [Navarro and Raffinot 2002]. For the experiments in this article, the algorithm was implemented with a block size of $B = 3$ and $B' = 2$ while hash values h and h' were calculated by shift and add; shifting the hash value to the left by bitshift positions and then adding in the ASCII value of a pattern or input string character.

The value of bitshift was set to 2. Finally, the verification of the patterns to the input string was performed using the *memcmp()* function of *string.h*.

The cost of the implementation used in the experiments of this article is as follows. To calculate the values of the *SHIFT*, *HASH* and *PREFIX* tables during the preprocessing phase, the algorithm requires an $\mathcal{O}(m^2)$ time. The space of Wu-Manber depends on the size of *SHIFT*, *HASH* and *PREFIX*. The space needed for the *SHIFT* table is $\sum_{i=0}^{B-1} |\Sigma| \times (2^{bitshift})^i$. In the worst case, there could be m patterns with the same hash value h or h' for their B-character suffix or B'-character prefix, respectively; therefore, *HASH* and *PREFIX* require a $m \times \sum_{i=0}^{B-1} |\Sigma| \times (2^{bitshift})^i$ space for a space complexity of $\mathcal{O}(m \times \sum_{i=0}^{B-1} |\Sigma| \times (2^{bitshift})^i)$. In the worst case for the searching phase of the Wu-Manber algorithm, the input string and $m-1$ characters of all m patterns consist of the same repeating character σ with the character at position $m-B-1$ of each pattern being different. The algorithm will then encounter a potential match on every position of the input string because $SHIFT[h]$ will constantly be 0. Therefore, as hash values h and h' of the patterns will be identical, the $m-B$ characters of all m patterns will be compared directly with the input string using the *memcmp()* function. The worst case searching time of Wu-Manber is given in Chen et al. [2005] as $\mathcal{O}(nm^2 \log_{|\Sigma|} m^2)$. In Navarro and Fredriksson [2004] the lower bound for the average time complexity of exact multiple pattern matching algorithms is given as $\Omega(\frac{n \log_{|\Sigma|} m^2}{m})$, and according to Chen et al. [2005], the searching phase of the Wu-Manber algorithm is optimal in the average case for a time complexity of $\mathcal{O}(\frac{n \log_{|\Sigma|} m^2}{m})$. In Liu et al. [2005] the average time complexity of Wu-Manber was also estimated as $\mathcal{O}(\frac{n}{(m-B+1)\times(1-\frac{(m-B+1)\times m}{2\times|\Sigma|^B})})$.

Implementing Baker and Bird with the Wu-Manber algorithm requires significantly more effort than the Set Horspool and Set Backward Oracle Matching variants presented earlier. Rather than using a trie, the Wu-Manber variant of Baker and Bird is using a search window that slides along each row j of the input string. During the preprocessing phase, a hash value *hs* is computed for each pattern row $p^0, p^1, \ldots, p^{m-1}$ using the hash function of the Wu-Manber algorithm. Both the hash values and the pointers to the pattern rows they correspond are stored in a hashmap that is then used to assign a unique index to each different pattern row in $\mathcal{O}(m^2)$ time. Pattern rows with the same hash value are compared on a character by character basis to determine if they are identical, before they are assigned the same index. That way, the two-dimensional pattern P is reduced to a one-dimensional array P' of indices. Moreover, the *SHIFT, HASH* and *PREFIX* tables of the Wu-Manber algorithm are built from the pattern rows. The array P' is then preprocessed using the Knuth-Morris-Pratt algorithm to compute the values of the *next* table. During the row-matching step and for each position j, k of the input string, a hash value h of the B-character suffix of the search window at $t_{j,k-m+1} \ldots t_{j,k}$ is computed backwards using the $h_1()$ function. If $SHIFT[h] = 0$, the hash value h' of the B'-character prefix of the search window is also computed. If both the prefix and suffix of the search window match the prefix and suffix of one or more pattern rows, the corresponding rows are compared directly with the input string to determine if a match of one of the pattern rows occurs. The search window is then shifted by one position to the right. If, on the other hand, $SHIFT[h] \neq 0$, the search window is subsequently shifted by $SHIFT[h]$ positions. The Wu-Manber variant of the Baker and Bird algorithm is $\mathcal{O}(n^2 m^2 \log_{|\Sigma|} m^2)$ in the worst case and $\mathcal{O}(\frac{n^2 \log_{|\Sigma|} m^2}{m})$ in the average case.

The Baeza-Yates and Regnier algorithm also requires a unique index to be assigned to each different pattern row. During the preprocessing phase, a hash value *hs* is

calculated for each $p \in P$ and together with a pointer to the pattern it corresponds are stored in a hashmap. Each pattern row is then assigned an index in $\mathcal{O}(m^2)$ time using the hashmap as already discussed. Finally, the *SHIFT, HASH* and *PREFIX* tables are computed. The search phase is performed backwards only on the primary rows of the input string. Starting from position $m-1$ of a primary row j and for each character $t_{j,k}$, the hash value h of the B-character suffix of the search window at $t_{j,k-m+1} \dots t_{j,k}$ is calculated. Based on the value of $SHIFT[h]$, the search window is either shifted to the right by $SHIFT[h]$ positions or its B'-character prefix is calculated. If both the prefix and suffix of the search window match the prefix and suffix of some pattern rows, then the corresponding rows are compared directly with the input string. If there is a match on a primary row, then a hash value *hs* is computed for each of the m-character substrings of the at most $2m-2$ secondary rows and based on the hashmap it is determined if they correspond to the appropriate indices of the pattern rows. The search window is then shifted by one position to the right. The theoretical search phase complexity of the Baeza-Yates and Regnier algorithm when implemented using the Wu-Manber algorithm is $\mathcal{O}(n^2m^3 \log_{|\Sigma|} m^2)$ in the worst case and $\mathcal{O}(\frac{n^2 \log_{|\Sigma|} m^2}{m^2})$ in the average case.

3.4. Two-Dimensional Pattern Matching Using SOG

The SOG algorithm extends the Shift-Or [Baeza-Yates and Gonnet 1992] string matching algorithm to perform multiple pattern matching in linear time, on average. SOG is a bit-parallel algorithm simulating a nondeterministic automaton that acts as a character class filter; it constructs a generalized pattern that can simultaneously match all patterns from a finite set. The generalized pattern accepts classes of characters based on the actual position of the characters in the patterns. The searching phase of the algorithm consists of a filtering phase and a verification phase. When a candidate match is found at a given position of the input string during the filtering phase, the patterns are verified using a combination of hashing and binary search to determine if a complete match of a pattern occurs. When the pattern set has a relatively large size, every position of the generalized pattern will accept most characters of the alphabet. In that case, false candidate matches will occur in most positions of the input string. To overcome this problem, SOG increases the alphabet size to $|\Sigma|^B$ by processing the characters in blocks of size B, similar to the methodology used by the Wu-Manber algorithm.

The preprocessing time of the SOG algorithm is $\mathcal{O}(m^2)$ with an $\mathcal{O}(|\Sigma|^B+m^2)$ space. For the verification of the patterns using two-level hashing and binary search, no checking of candidate matches is needed in the best case. In the worst case and assuming that all pattern rows and input string positions have the same hash value, the verification time is $\mathcal{O}(nm^2)$. If the pattern rows have a different hash value, the verification time is $\mathcal{O}(n(\log m + m))$ instead. The filtering phase is linear in n in the worst and average case. The combined filtering and verification phase is then $\mathcal{O}(nm^2)$ when all pattern rows have the same hash value and $\mathcal{O}(nm)$ otherwise in the worst case and linear in n in the average case.

During the preprocessing phase, a hash value h is assigned to each different B-character block of the patterns using a hash function $h_1()$. For each different h, a bit vector $V[h]$ is initialized by setting the i^{th} bit of $V[h]$ to 0 if the B-character block corresponding to h is found in the i^{th} position of any pattern or to 1 otherwise. For the verification phase, a hash value *hs* is computed for each pattern by forming a 32-bit integer of every four bytes of the pattern and then using the XOR logical operation between the integers. For a pattern p^r of length $m = 8$, its hash value *hs* will be:

$$hs = p_0^r p_1^r p_2^r p_3^r \veebar p_4^r p_5^r p_6^r p_7^r.$$

Table II. Known Theoretical Extra Space, Preprocessing, Worst and Average Searching Time Complexity of the Baker and Bird Algorithm Implementations

Implementation	Extra space	Preprocessing	Worst case	Average case
Aho-Corasick	$n + \|\Sigma\|m^2$	$\|\Sigma\|m^2$	n^2	n^2
Set Horspool	$n + \|\Sigma\|m^2$	$\|\Sigma\|m^2$	n^2m	Kn
Set Backward Oracle Matching	$n + \|\Sigma\|m^2$	$\|\Sigma\|m^2$	n^2m^2	Kn
Wu-Manber	$n + m \times \sum_{i=0}^{B-1} \|\Sigma\| \times (2^{bitshift})^i$	m^2	$n^2m^2 \log_{\|\Sigma\|} m^2$	$\frac{n^2 \log_{\|\Sigma\|} m^2}{m}$
SOG	$n + \|\Sigma\|^B + m^2$	m^2	n^2m^2	n^2

To improve the efficiency of the hashing method described, a two-level hashing technique [Muth and Manber 1996] is used. This technique involves creating a 16-bit hash value hs' by using the XOR logical operation between the lower and the upper 16 bits of hs that is stored in an ordered table. For the experiments of this article, SOG was implemented with a block size of $B = 3$.

The SOG variant of the Baker and Bird algorithm uses a search window that slides along each row of the input string instead of a trie, similar to the Wu-Manber variant presented earlier. During the preprocessing phase, a hash value h is assigned to each different B-character block of the patterns using the hash function $h1()$ of the SOG algorithm. Additionally, a hash value hs is computed from each pattern row and a two-level hash value hs' is created from hs and stored in an ordered table. The hash value hs for each pattern row and a pointer to the pattern row it corresponds are stored in a hashmap that is then used to assign a unique index to each different pattern row in $\mathcal{O}(m^2)$ time. To search for the occurrence of the patterns in the input string during the row-matching step, an m-bit variable E is used that is updated based on the hash value h of the B-character substrings of each row j of the input string. If the $(m - B)^{th}$ bit of E is equal to 0, a candidate match occurs that is verified by calculating the two-level hash of the search window and then searching the ordered table for its occurrence using binary search. When the Baker and Bird algorithm is implemented using the SOG algorithm, it has an $\mathcal{O}(n^2m^2)$ theoretical search phase complexity in the worst case and $\mathcal{O}(n^2)$ in the average case.

The SOG variant of the Baeza-Yates and Regnier algorithm can be implemented in a similar way. During the preprocessing phase, the two-dimensional pattern P is reduced to a one-dimensional array P' of indices by calculating the hash value hs of each pattern row and storing it in a hashmap together with a pointer to the pattern row it corresponds. Moreover, a two-level hash value hs' is created from hs and stored in an ordered table. Finally, a hash value h is assigned to each different B-character block of the pattern rows using a hash function $h1()$. The search phase is performed on only the primary rows as per the original Baeza-Yates and Regnier algorithm. For each position k of a primary row j, an m-bit variable E is used that is updated based on the hash value h of each B-character substring of the input string. If the $(m - B)^{th}$ bit of E is equal to 0, a candidate match occurs that is verified by calculating the two-level hash of the search window and then searching the ordered table for its occurrence using binary search. If a match is found on a primary row, then E is updated based on the hash value h of each B-character substring for each of the secondary rows in a similar way. The searching phase complexity of the SOG variant of the Baeza-Yates and Regnier algorithm is the same as of its original implementation; $\mathcal{O}(n^2m^3)$ in the worst case and $\mathcal{O}(\frac{n^2}{m})$ in the average.

Tables II and III summarize the theoretical extra space, preprocessing and worst and average searching time complexity of the presented Baker and Bird and Baeza-Yates

Table III. Known Theoretical Extra Space, Preprocessing, Worst and Average Searching Time Complexity of the Baeza-Yates and Regnier Algorithm Implementations

Implementation	Extra space	Preprocessing	Worst case	Average case
Aho-Corasick	$\|\Sigma\|m^2$	$\|\Sigma\|m^2$	n^2m	$\frac{n^2}{m}$
Set Horspool	$\|\Sigma\|m^2$	$\|\Sigma\|m^2$	n^2m^2	$\frac{Kn}{m}$
Set Backward Oracle Matching	$\|\Sigma\|m^2$	$\|\Sigma\|m^2$	n^2m^3	$\frac{Kn}{m}$
Wu-Manber	$m \times \sum_{i=0}^{B-1} \|\Sigma\| \times (2^{bitshift})^i$	m^2	$n^2m^3 \log_{\|\Sigma\|} m^2$	$\frac{n^2 \log_{\|\Sigma\|} m^2}{m^2}$
SOG	$\|\Sigma\|^B + m^2$	m^2	n^2m^3	$\frac{n^2}{m}$

and Regnier algorithm implementations. The parameter K is defined such that $K < n$ in the average case. The searching time of SOG corresponds to the time of the filtering and verification phases of the algorithm combined.

4. EXPERIMENTAL METHODOLOGY

The parameters that describe the performance of two-dimensional pattern matching algorithms are the length n^2 of the input string, the length m^2 of the pattern, and the size $|\Sigma|$ of the alphabet used.

To evaluate the performance of the original algorithms and the introduced variants, the preprocessing and the searching time were used as a measure. Preprocessing time is the time in seconds an algorithm uses to preprocess the pattern, whereas the searching time is the total time in seconds an algorithm uses to locate all occurrences of the pattern in the input string. Both times were measured using the MPI_Wtime function of the message passing interface because it has a better resolution than the standard *clock*() function of *time.h*.

The data set used was a superset of the sets used in Baeza-Yates and Regnier [1993], Tarhio [1996], and Zhu and Takaoka [1989]. It consisted of a randomly generated input string with a length of $n = 1{,}000$ and $n = 10{,}000$ with three alphabets of size 2,256 and 1,024 to simulate bitmaps with different color depths. The pattern had a length of $m = 4, 8, 16, 32, 64, 128$, and 256 characters.

The experiments were executed locally on an Intel Core 2 Duo E8400 CPU with a 3.00GHz clock speed and 3GB of memory, 64KB L1 cache per core and 6MB L2 shared cache. The Ubuntu Linux operating system was used and during the experiments only the typical background processes ran. To decrease random variation, the time results were averages of 100 runs. All algorithms were implemented using the ANSI C programming language and were compiled using the GCC 4.4.3 compiler with the "-O2" and "-funroll-loops" optimization flags.

5. ANALYSIS

This section discusses the efficiency and presents a performance evaluation in terms of preprocessing and searching time of the original Baker and Bird and Baeza-Yates and Regnier two-dimensional pattern matching algorithms and their Set Horspool, Set Backward Oracle Matching, Wu-Manber, and SOG variants for different problem parameters including the pattern, input string, and alphabet sizes used.

Figure 3 depicts the preprocessing time of the different implementations of the Baker and Bird algorithm. The preprocessing time of the Baeza-Yates and Regnier algorithm was similar and thus is omitted. The searching time of the original Baker and Bird and the Baeza-Yates and Regnier algorithms and their variants is presented in Figures 4 to 7. All figures have a linear horizontal axis and a logarithmic vertical. Since the SOG algorithm uses bitwise operations to create the hash values for the verification

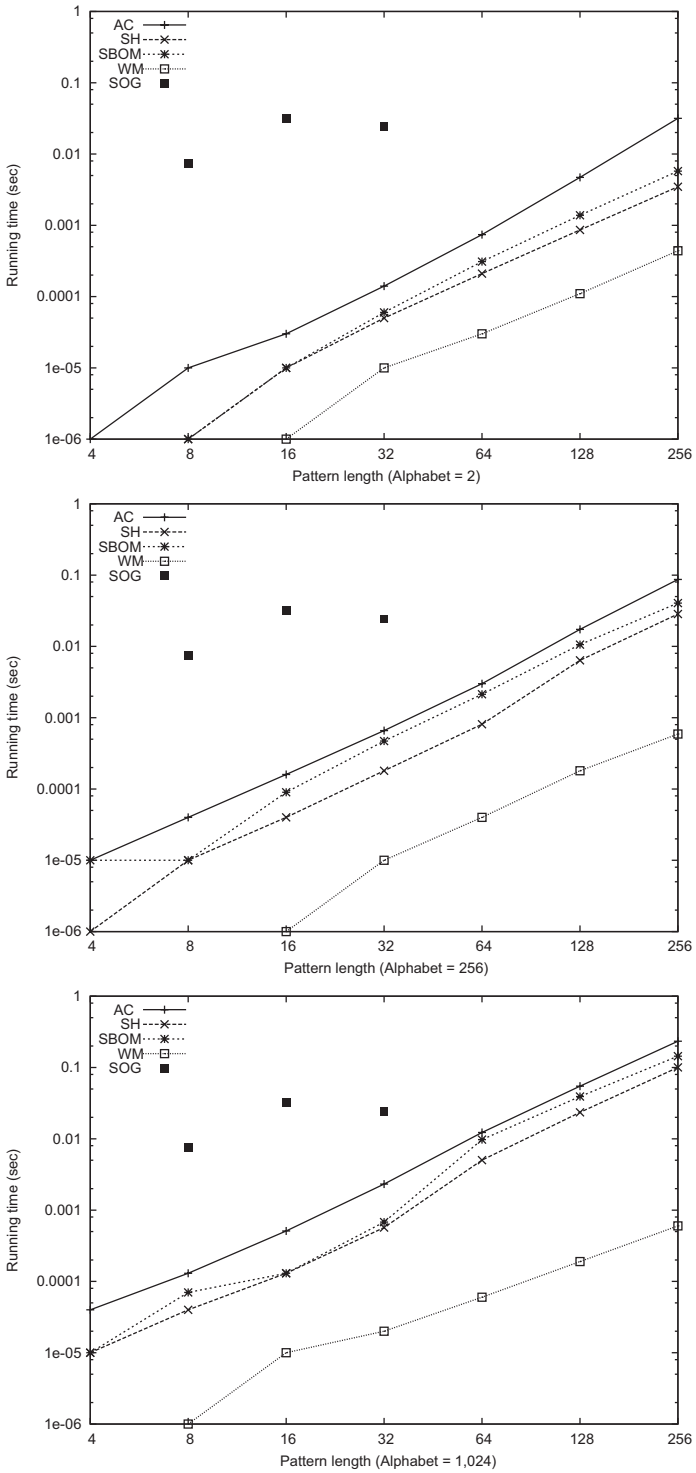

Fig. 3. Preprocessing time of the algorithm implementations.

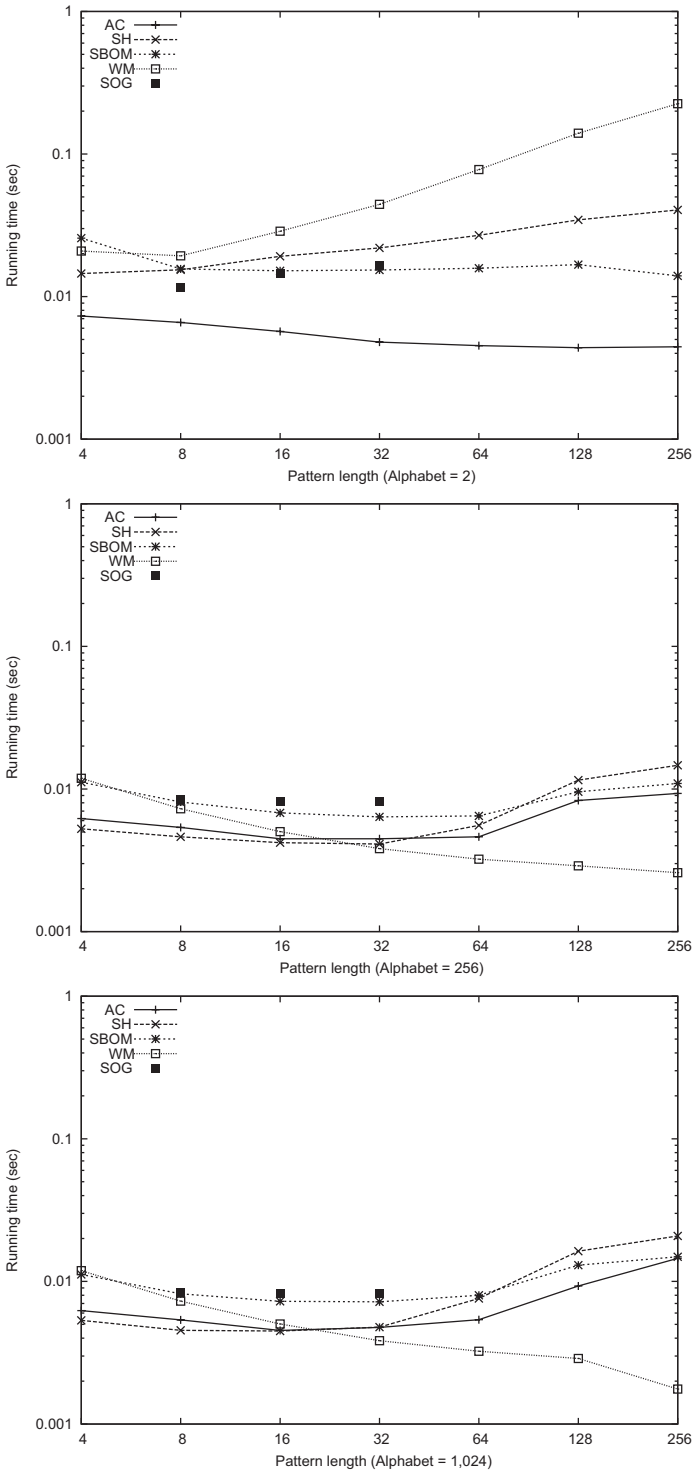

Fig. 4. Searching time of the Baker and Bird implementations for an input string with $n = 1{,}000$.

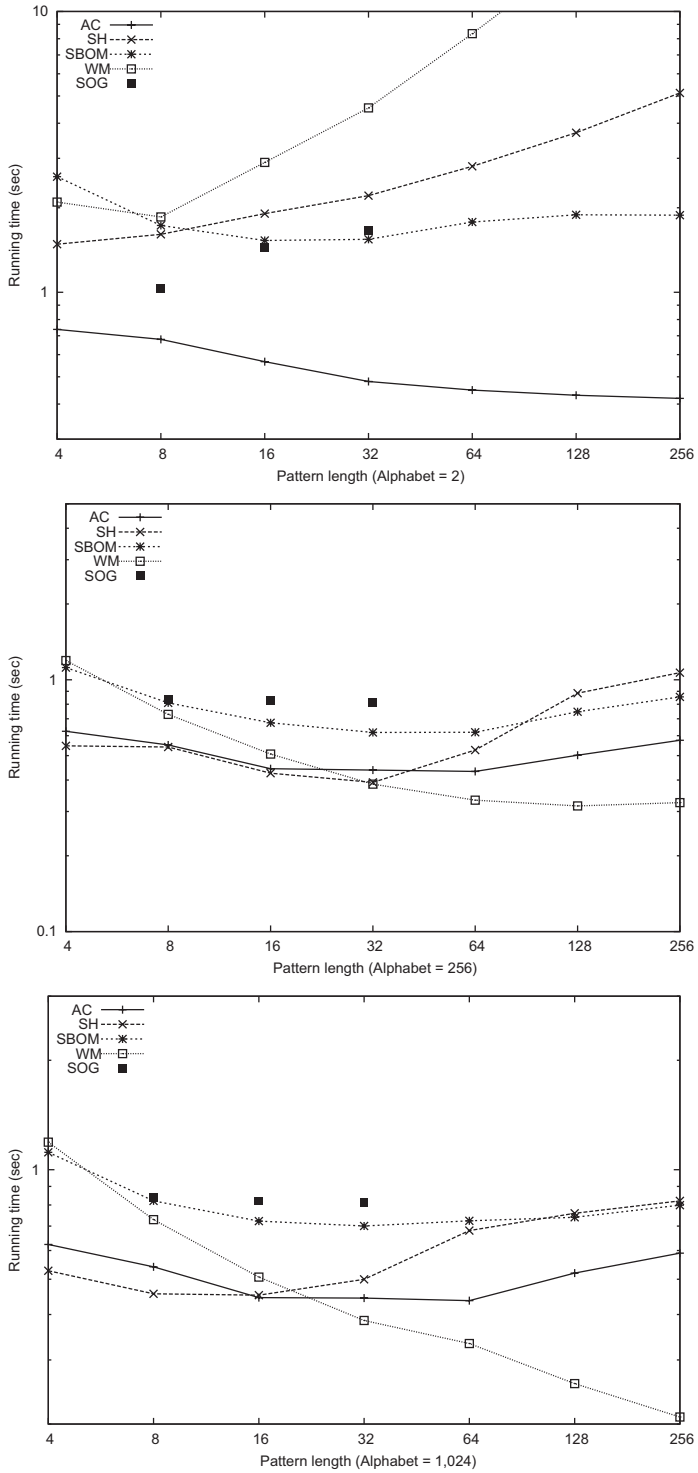

Fig. 5. Searching time of the Baker and Bird implementations for an input string with $n = 10{,}000$.

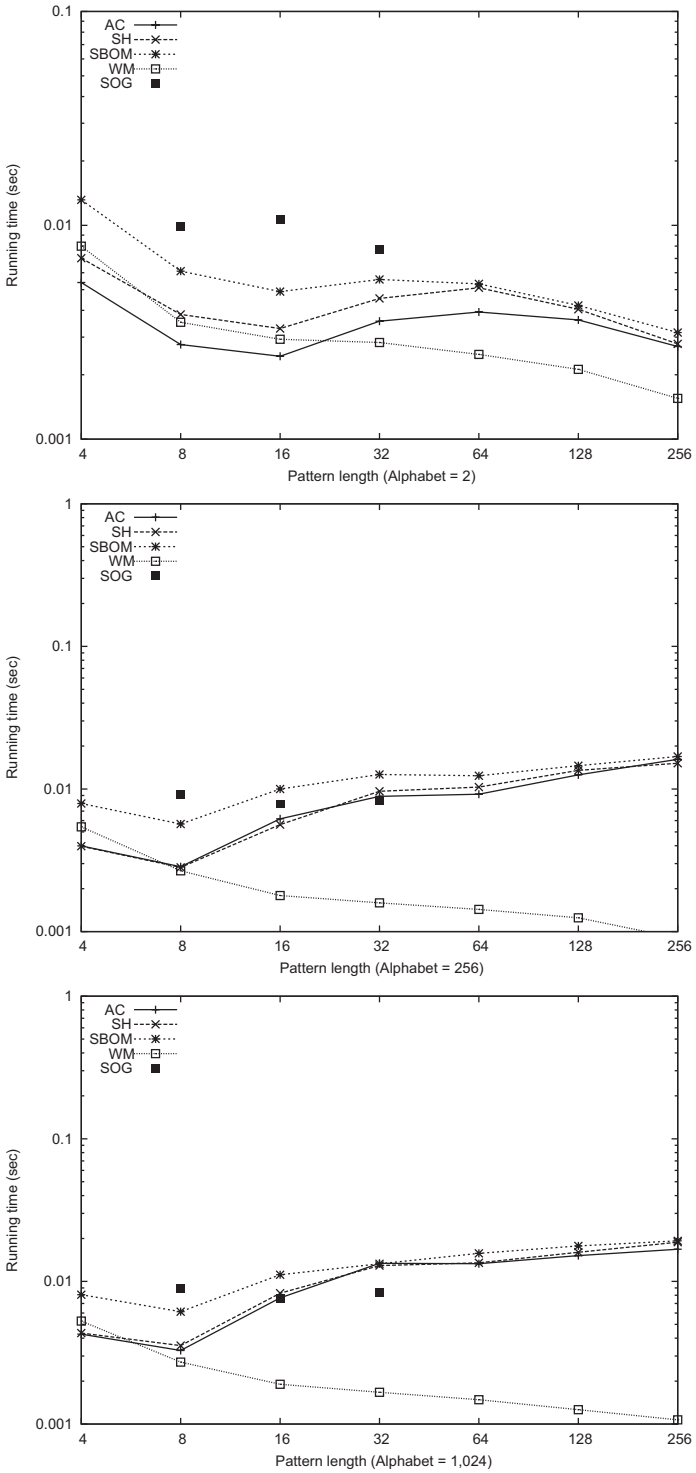

Fig. 6. Searching time of the Baeza-Yates and Regnier implementations for an input string with $n = 1{,}000$.

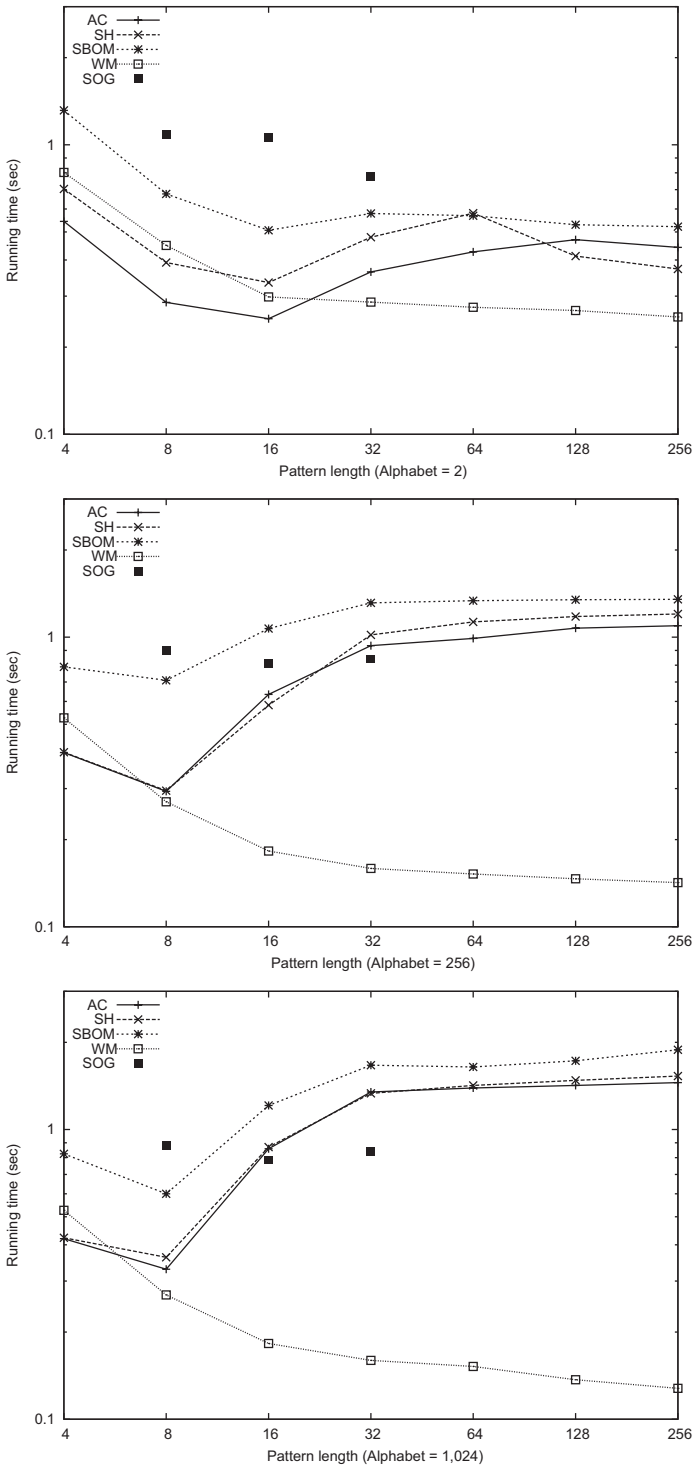

Fig. 7. Searching time of the Baeza-Yates and Regnier implementations for an input string with $n = 10{,}000$.

phase, it is not trivial to implement it on a 32-bit hardware for a pattern where the length m of each dimension is larger than 32 characters, although it could potentially be used as a filter to search for the occurrence of subpatterns with a length of up to 32 characters. For this reason, the searching time of the Baker and Bird and the Baeza-Yates and Regnier algorithms is given only for patterns of size $m = 8, 16$, and 32 characters when implemented using the SOG algorithm, similar to the methodology presented in Salmela et al. [2006]. The experimental results with only a few exceptions are consistent with the theoretical time complexity, as presented in Tables I through III.

5.1. Performance Evaluation

The Baker and Bird and the Baeza-Yates and Regnier implementations presented in this article use the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber, and the SOG multiple pattern matching algorithms to preprocess the pattern as discussed in previous Sections. As shown in Table I, the theoretical time to construct the trie of Aho-Corasick and Set Horspool and the factor oracle of Set Backward Oracle Matching is $\mathcal{O}(|\Sigma|m^2)$, whereas the theoretical preprocessing time of the Wu-Manber and SOG algorithms is $\mathcal{O}(m^2)$. If the theoretical time complexity of an algorithm can be used as a means to predict its performance, the preprocessing time of the implementations was expected to increase linearly in m^2. The experimental results, as presented in Figure 3, confirm that the preprocessing time of the original implementation of the Baker and Bird and the Baeza-Yates and Regnier algorithms and of their Set Horspool, Set Backward Oracle Matching and Wu-Manber variants increased linearly in the size of the pattern. The preprocessing time of the SOG variant of both the Baker and Bird and the Baeza-Yates and Regnier algorithms increased proportional to the length of the pattern when used on a pattern with $m = 16$ as opposed to $m = 8$ characters, but it was constant in m^2 as the length increased from 16 to 32 characters. This irregularity was caused by the aggressive optimization of the code by the GCC compiler when the "O2" flag was used and was not observed with the "O0" flag.

Figure 3 also shows that the preprocessing time of all algorithm implementations, regarding the alphabet size, was in line with their theoretical complexity. More specifically, the time of the Aho-Corasick, Set Horspool, and Set Backward Oracle Matching implementations of the Baker and Bird and the Baeza-Yates and Regnier algorithms to preprocess the pattern increased linearly in $|\Sigma|$, whereas the time of the Wu-Manber and SOG variants of both two-dimensional algorithms was constant in $|\Sigma|$.

The experiments confirm in practice that the time of the Aho-Corasick implementation of the Baker and Bird and the Baeza-Yates and Regnier algorithms to locate all occurrences of the pattern in the input string increased linearly in the length n^2 of the input string. The pattern length and the alphabet size are two parameters that are closely correlated as to the manner in which they affect the performance of the algorithm implementations. The searching time of the original Baker and Bird algorithm was roughly constant in m^2, as expected by its average case theoretical time, although an increase in the searching time of the implementation is apparent for large pattern sizes, especially when data with alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$ were used. A similar behavior of the original Baeza-Yates and Regnier algorithm can be observed to a much greater extent in Figures 6 and 7. In that case and although the searching time of the implementation had a tendency to decline, in accordance to its theoretical complexity, a sharp increase can be observed, especially on data with alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$. The increase in the searching time of the implementation is related to the rate of memory cache misses as trie nodes are visited during the searching phase. The CPU used for the experiments of this article had a 64KB L1 cache per core and a 6MB L2 cache. All data read or written by the CPU are

stored in the L1 cache in the form of 64-byte cache lines, namely blocks of contiguous data words. The limited size of the cache results in the eviction of the cache lines to the L2 cache and from there to main memory to make room in the caches for new cache lines. Each trie node uses a table of size $|\Sigma|$ to store the transitions to other nodes and since in principle the frequency of cache misses is determined by the size of the data, increasing the size of the alphabet resulted in the degradation of the implementations performance. To measure the effect of the memory caches on the performance of the algorithm implementations, Callgrind, an execution-driven cache simulator was used from the Valgrind suite [Nethercote and Seward 2007]. A detailed analysis on the memory architecture of modern processors can be found in Drepper [2007]. Finally, it can be seen that the searching time of the original Baker and Bird and Baeza-Yates and Regnier algorithms was constant in $|\Sigma|$, if the sharp increase in time due to the cache misses is taken into consideration.

The performance in terms of searching time of the Set Horspool and Set Backward Oracle Matching variants of the Baker and Bird and the Baeza-Yates and Regnier algorithms has similar characteristics to the original implementations. The time to locate all occurrences of the pattern in the input string increased linearly in n^2 while a sharp increase in m^2 of their searching time is evident when data with alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$ were used due to cache misses. Contrary to the original Baker and Bird algorithm, the searching time of its Set Horspool and the Set Backward Oracle Matching variants actually decreased in the size of the alphabet, although it was expected to be constant in $|\Sigma|$ in the average case. Recall that when a mismatch occurs at a position i of the input string, Set Horspool uses the bad character shift of the Horspool algorithm to shift the trie, while the oracle of the Set Backward Oracle Matching algorithm is shifted past that position. As the maximum possible shift of the trie and the factor oracle, respectively, is greatly reduced on data with a small $|\Sigma|$, it can explain the reduction of the efficiency of the specific algorithm implementations when used on data with a binary alphabet. The searching time of the Set Horspool and the Set Backward Oracle Matching variants of the Baeza-Yates and Regnier algorithm, on the other hand, was roughly constant in $|\Sigma|$. Although the maximum shift was reduced in that case as well, both the trie and the factor oracle were only shifted along the primary rows of the input string, thereby not affecting significantly the performance of the implementations.

The searching time of the Wu-Manber variant of the Baker and Bird and the Baeza-Yates and Regnier algorithms also increased linearly in n^2. The performance in terms of searching time of the Wu-Manber implementation of the Baker and Bird algorithm was influenced by the length of the pattern in different ways, based on the alphabet size used; the time to scan the input string increased in m^2 for data with a binary alphabet while decreased when alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$ were used. Wu-Manber, similar to the Set Horspool algorithm, uses the bad character shift of the Horspool algorithm to shift the search window when a mismatch or a complete match occurs, and thus, for the reasons described earlier, the algorithm is not efficient when data with a small alphabet size are used. In practice, for data with a binary alphabet together with patterns where $m \geq 16$, the Wu-Manber variant of Baker and Bird performed only single-character shifts during the row-matching step of the algorithm. As with the Set Horspool and the Set Backward Oracle Matching variants of the Baeza-Yates and Regnier algorithm, the searching time of the Wu-Manber variant was not significantly affected by the reduction of the maximum shift when binary alphabet data were used and thus it was decreased in m^2 for all alphabet sizes, as expected by the average theoretical complexity of the algorithm. The time of the Wu-Manber variant of both the Baker and Bird and the Baeza-Yates and Regnier algorithms to locate all occurrences of the pattern in the input string decreased in $|\Sigma|$.

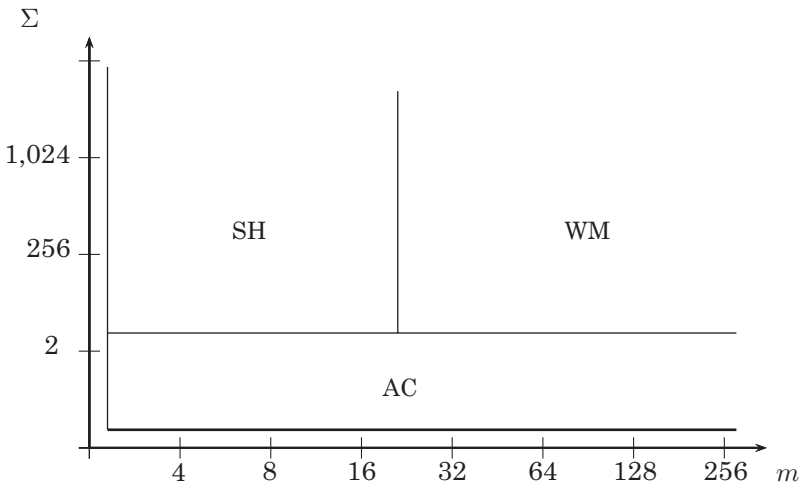

Fig. 8. Experimental map of the implementations of the Baker and Bird algorithm for $n = 1{,}000$ and $n = 10{,}000$.

Finally, the searching time of the SOG variant of both the Baker and Bird and the Baeza-Yates and Regnier algorithms increased linearly in n^2, whereas it was roughly constant in m^2 and $|\Sigma|$, as expected by the average theoretical searching time of the algorithm implementations.

5.2. Algorithm Comparison

As shown in Figures 4 through 7, the searching time of the two-dimensional algorithm implementations was affected in various ways by different problem parameters, and as such, different implementations are preferred for different types of data. The implementations of the presented two-dimensional pattern matching algorithms had similar characteristics for input strings with sizes of $n = 1{,}000$ and $n = 10{,}000$.

When comparing the implementations of the Baker and Bird algorithm in terms of searching time, it is clear that the original algorithm was faster than the presented variants for data with a binary alphabet size. The Set Horspool variant of Baker and Bird outperformed the rest of the implementations when data with alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$ were used together with patterns with a length m between 4 and 16 characters. Finally, the Wu-Manber variant had the fastest searching time for patterns with a length of $m \geq 32$ characters when the same alphabet sizes were used. Although the Set Backward Oracle Matching and SOG variants of the Baker and Bird algorithm had a comparable performance in terms of searching time to the rest of the implementations, they were consistently slower for all types of data.

The original implementation of the Baeza-Yates and Regnier algorithm had the fastest searching time among the presented implementations for patterns with a length between $m = 4$ and $m = 16$ when data with a binary alphabet were used. Moreover, the Aho-Corasick together with the Set Horspool implementation of Baeza-Yates and Regnier were faster than the Set Backward Oracle Matching, Wu-Manber, and SOG variants for data with an alphabet of size $|\Sigma| = 256$ and 1,024 and for patterns with a length of $m = 4$. The Wu-Manber variant outperformed the rest of the algorithm implementations for patterns with a length of $m \geq 32$ and a binary alphabet and when data with an alphabet of size $|\Sigma| = 256$ and 1,024 together with patterns of a length $m \geq 8$ were used.

The presented implementations of the Baeza-Yates and Regnier algorithm had a consistently better performance than the corresponding implementations of the Baker

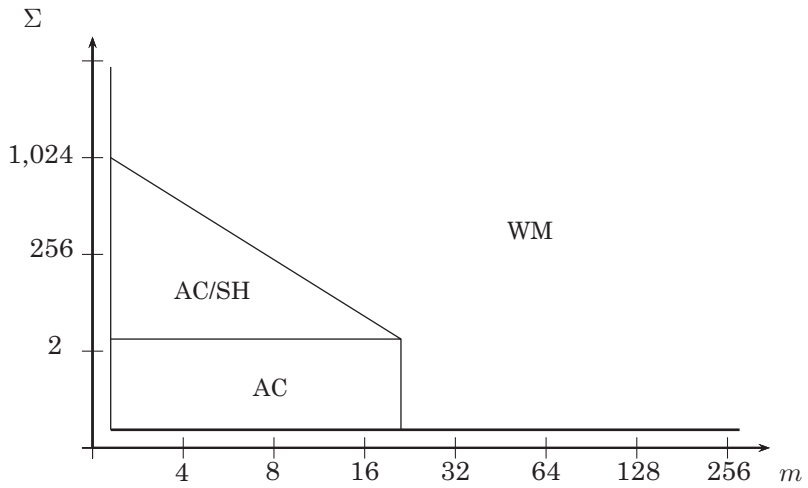

Fig. 9. Experimental map of the implementations of the Baeza-Yates and Regnier algorithm for $n = 1{,}000$ and $n = 10{,}000$.

and Bird algorithm when used on data with a binary alphabet. It is interesting to note that the Wu-Manber variant of the Baeza-Yates and Regnier algorithm was two orders of magnitude faster than the respective variant of the Baker and Bird algorithm for data with a binary alphabet and for all pattern and input string lengths, as shown in Figures 4 through 7. On data with alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$ characters, the Aho-Corasick, Set Horspool, and Set Backward Oracle Matching implementations of the Baker and Bird algorithm were faster than the Aho-Corasick, Set Horspool and Set Backward Oracle Matching implementations of Baeza-Yates and Regnier while the Wu-Manber variant was up to three times slower. For the same types of data, the SOG variant of both Baker and Bird and Baeza-Yates and Regnier algorithms had a roughly equal performance.

The experimental maps that are depicted in Figures 8 and 9 summarize the most efficient among the presented implementations of the Baker and Bird and the Baeza-Yates and Regnier algorithms for different pattern, alphabet, and input string sizes.

6. CONCLUSIONS

This article presented efficient variants of the Baker and Bird and the Baeza-Yates and Regnier algorithms that use the trie of the Set Horspool algorithm, the factor oracle of the Set Backward Oracle Matching algorithm, and the hashing functions of the Wu-Manber and SOG algorithms to preprocess a two-dimensional pattern and then locate all of its occurrences in a two-dimensional input string. The performance of the original algorithms and their variants was compared in terms of preprocessing and searching time, and it was detailed the way it was affected by different problem parameters including the length of the pattern and the input string as well as the size of the alphabet, since these are the parameters that generally affect the performance of two-dimensional pattern matching algorithms.

As discussed in the previous section, using the Set Horspool, Wu-Manber, Set Backward Oracle Matching, and SOG data structures in place of the trie of Aho-Corasick resulted in the improvement of the efficiency in terms of searching time of the Baker and Bird and the Baeza-Yates and Regnier algorithms for some types of data. More specifically, the original implementation of the Baker and Bird outperformed the presented variants only on data with a binary alphabet. When a pattern and an

input string with alphabets of size $|\Sigma| = 256$ and $|\Sigma| = 1{,}024$ were used, the Set Horspool variant outperformed the rest of the Baker and Bird implementations for a pattern of a length up to $m = 16$, while Wu-Manber was the fastest implementation for a bigger pattern length. In a similar fashion, the original implementation of the Baeza-Yates and Regnier algorithm had the fastest searching time among the presented implementations when data with a binary alphabet were used together with a pattern of a length up to $m = 16$ as well as for data with an alphabet of size $|\Sigma| = 256$ and 1,024 when a pattern with a length of $m = 4$ was used, together with the Set Horspool implementation. For a pattern with a length m between 32 and 256 characters, when a binary alphabet was used and for a pattern with a length m between 8 and 256 when an alphabet of size $|\Sigma| = 256$ and 1,024 was used, the Wu-Manber variant outperformed the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, and SOG implementations of the Baeza-Yates and Regnier algorithm.

The work presented in this article could be extended with a performance evaluation of the presented algorithm variants for larger data sizes and for additional types of data, including photo archives and satellite imagery. The way the memory accesses of the different implementations of the Baker and Bird and the Baeza-Yates and Regnier algorithms affect their performance is a very interesting subject that should be studied further. Future research in the area of two-dimensional pattern matching could focus on further speeding up the existing algorithms when parallel processed on traditional parallel architectures including cluster environments and multicore systems as well as on modern parallel systems like GPU architectures, especially when large data are involved (i.e., aerial imaging) or in time-critical applications.

REFERENCES

Aho, A. and Corasick, M. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18,* 6, 333–340.

Allauzen, C., Crochemore, M., and Raffinot, M. 1999. Factor oracle: A new structure for pattern matching. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics.* 758–758.

Allauzen, C. and Raffinot, M. 1999. Factor oracle of a set of words. Tech. rep. 99–11, Institut Gaspard-Monge, Université de Marne-la-Vallée.

Amir, A. 1992. Multidimensional pattern matching: A survey. Tech. rep. GIT-CC-92/29.

Amir, A., Kapah, O., and Tsur, D. 2006. Faster two-dimensional pattern matching with rotations. *Theor. Comput. Sci. 368,* 3, 196–204.

Amir, A., Landau, G., and Vishkin, U. 1992. Efficient pattern matching with scaling. *J. Algorithms 13,* 1, 2–32.

Apostolico, A. and Galil, Z. 1997. *Pattern Matching Algorithms*. Oxford University Press.

Baeza-Yates, R. and Fuentes, L. 1996. A framework to animate string algorithms. *Inf. Process. Lett. 59,* 5, 241–244.

Baeza-Yates, R. and Gonnet, G. 1992. A new approach to text searching. *Commun. ACM 35,* 10, 74–82.

Baeza-Yates, R. and Perleberg, C. 1992. Fast and practical approximate string matching. In *Combinatorial Pattern Matching*. Springer, 185–192.

Baeza-Yates, R. and Regnier, M. 1993. Fast two dimensional pattern matching. *Inf. Process. Lett. 45,* 1, 51–57.

Baker, T. 1978. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput. 7,* 4, 533–541.

Bird, R. 1977. Two dimensional pattern matching. *Inf. Process. Lett. 6,* 5, 168–170.

Boyer, R. and Moore, J. 1977. A fast string searching algorithm. *Commun. ACM 20,* 10, 762–772.

Chen, X., Fang, B., Li, L., and Jiang, Y. 2005. WM+: An optimal multi-pattern string matching algorithm based on the wm algorithm. In *Advanced Parallel Processing Technologies*. Springer, 515–523.

Commentz-Walter, B. 1979. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*. 118–132.

Crochemore, M. and Rytter, W. 1994. *Text Algorithms*. Oxford University Press.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*, 1st Ed. World Scientific.

DONG-HONG, Y., KE, X., AND YONG, C. 2006. An improved Wu-Manber multiple patterns matching algorithm. In *Proceedings of the 25th IEEE International Performance, Computing and Communications Conference*. 675–680.

DORI, S. AND LANDAU, G. 2006. Construction of Aho Corasick automaton in linear time for integer alphabets. *Inf. Process. Lett. 98,* 2, 66–72.

DREPPER, U. 2007. What every programmer should know about memory. www.akkadia.org/drepper/cpumemory.pdf.

FREDRIKSSON, K., NAVARRO, G., AND UKKONEN, E. 2002. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In *Combinatorial Pattern Matching 2373*. Springer, 235–248.

HORSPOOL, R. 1980. Practical fast searching in strings. *Software: Practice and Experience 10,* 6, 501–506.

KARP, R. AND RABIN, M. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev. 31,* 2, 249–260.

KNUTH, D., MORRIS, J., AND PRATT, V. 1977. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2, 323–350.

KOUZINOPOULOS, C. AND MARGARITIS, K. 2009. Parallel implementation of exact two dimensional pattern matching algorithms using MPI and OpenMP. In *Proceedings of the 9th Hellenic European Research on Computer Mathematics and Its Applications Conference*.

KOUZINOPOULOS, C. AND MARGARITIS, K. 2010. Experimental results on algorithms for multiple keyword matching. In *Proceedings of the IADIS International Conference on Informatics*. 129–133.

KOUZINOPOULOS, C. AND MARGARITIS, K. 2011. Experimental Results on multiple pattern matching algorithms for biological sequences. In *Proceedings of the International Conference on Bioinformatics—Models, Methods and Algorithms*. 274–277.

LIU, P., LIU, Y., AND TAN, J. 2005. A partition-based efficient algorithm for large scale multiple-strings matching. In *Proceedings of 12th Symposium on String Processing and Information Retrieval*. Springer, 399–404.

MUTH, R. AND MANBER, U. 1996. Approximate multiple string search. In *Combinatorial Pattern Matching*. Springer, 75–86.

NAVARRO, G. AND FREDRIKSSON, K. 2004. Average complexity of exact and approximate multiple string matching. *Theor. Comput. Sci. 321,* 2–3, 283–290.

NAVARRO, G. AND RAFFINOT, M. 2002. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* 89–100.

POLCAR, T. AND MELICHAR, B. 2004. A two-dimensional online tessellation automata approach to two-dimensional pattern matching. In *Proceedings of the Eindhoven FASTAR Days*. www.fastar.org/publications/EFD2004Proceedings.pdf.

SALMELA, L., TARHIO, J., AND KYTÖJOKI, J. 2006. Multipattern string matching with q-grams. *J. Exp. Algor. 11*, 1–19.

SHEU, T., HUANG, N., AND LEE, H. 2008. Hierarchical multi-pattern matching algorithm for network content inspection. *Inf. Sci. 178*, 2880–2898.

STREAMLINE. 2012. Streamline. http://netstreamline.org/.

TAO, T. 2005. Compressed pattern matching for text and images. Ph.D. thesis, University of Central Florida. AAI3178974.

TARHIO, J. 1996. A sublinear algorithm for two-dimensional string matching. *Pattern Recognit. Lett. 17,* 8, 833–838.

TUCKER, A. 2004. *Computer Science Handbook*. CRC Press.

WATSON, B. 1995. Taxonomies and toolkits of regular language algorithms. Ph.D. thesis, Eindhoven University of Technology.

WU, S. AND MANBER, U. 1994. A fast algorithm for multi-pattern searching. Tech. Rep. TR-94-17.

ZDAREK, J. 2010. Two-dimensional pattern matching using automata approach. Ph.D. thesis, Czech Technical University.

ZHU, R. AND TAKAOKA, T. 1989. A technique for two-dimensional pattern matching. *Commun. ACM 32,* 9, 1110–1120.

Received December 2012; revised March 2013; accepted May 2013

APPENDIX: PREPROCESSING AND SEARCHING TIME RESULTS

Table IV. Preprocessing Time of the Baker and Bird Implementations (sec)

	$\|\Sigma\| = 2$					$\|\Sigma\| = 256$					$\|\Sigma\| = 1{,}024$				
	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG
4	0.0000	0.0000	0.0000	0.0000		0.0000	0.0000	0.0000	0.0000		0.0000	0.0000	0.0112	0.0000	
8	0.0000	0.0000	0.0000	0.0000	0.0075	0.0000	0.0000	0.0000	0.0000	0.0075	0.0001	0.0000	0.0082	0.0000	0.0075
16	0.0000	0.0000	0.0000	0.0000	0.0318	0.0002	0.0000	0.0001	0.0000	0.0318	0.0005	0.0001	0.0073	0.0000	0.0318
32	0.0001	0.0001	0.0001	0.0000	0.0242	0.0007	0.0002	0.0005	0.0000	0.0241	0.0023	0.0006	0.0072	0.0000	0.0242
64	0.0007	0.0002	0.0003	0.0000		0.0030	0.0008	0.0021	0.0000		0.0123	0.0050	0.0080	0.0001	
128	0.0047	0.0009	0.0014	0.0001		0.0173	0.0064	0.0106	0.0002		0.0548	0.0235	0.0130	0.0002	
256	0.0317	0.0035	0.0058	0.0004		0.0869	0.0284	0.0406	0.0006		0.2339	0.1004	0.0149	0.0006	

Table V. Preprocessing Time of the Baeza-Yates and Regnier Implementations (sec)

	$\|\Sigma\| = 2$					$\|\Sigma\| = 256$					$\|\Sigma\| = 1{,}024$				
	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG
4	0.0000	0.0000	0.0000	0.0000		0.0000	0.0000	0.0000	0.0000		0.0000	0.0000	0.0081	0.0000	
8	0.0000	0.0000	0.0000	0.0000	0.0075	0.0000	0.0000	0.0000	0.0000	0.0077	0.0001	0.0000	0.0061	0.0000	0.0076
16	0.0000	0.0000	0.0000	0.0000	0.0318	0.0002	0.0001	0.0001	0.0000	0.0318	0.0005	0.0001	0.0111	0.0000	0.0318
32	0.0001	0.0001	0.0001	0.0000	0.0241	0.0007	0.0002	0.0005	0.0000	0.0241	0.0022	0.0006	0.0133	0.0000	0.0242
64	0.0007	0.0002	0.0003	0.0000		0.0029	0.0008	0.0021	0.0000		0.0122	0.0050	0.0157	0.0001	
128	0.0048	0.0009	0.0014	0.0001		0.0213	0.0064	0.0106	0.0002		0.0547	0.0235	0.0177	0.0002	
256	0.0311	0.0035	0.0058	0.0005		0.0875	0.0285	0.0407	0.0006		0.1340	0.0651	0.0193	0.0006	

Table VI. Searching Time of the Baker and Bird Implementations for an Input String with $n = 1{,}000$ (sec)

	$\|\Sigma\| = 2$					$\|\Sigma\| = 256$					$\|\Sigma\| = 1{,}024$				
	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG
4	0.0073	0.0145	0.0257	0.0209		0.0062	0.0053	0.0112	0.0119		0.0062	0.0053	0.0112	0.0119	
8	0.0066	0.0154	0.0156	0.0193	0.0117	0.0054	0.0046	0.0081	0.0073	0.0084	0.0054	0.0045	0.0082	0.0073	0.0084
16	0.0057	0.0192	0.0152	0.0287	0.0145	0.0045	0.0042	0.0068	0.0050	0.0082	0.0045	0.0045	0.0073	0.0050	0.0082
32	0.0048	0.0220	0.0154	0.0443	0.0166	0.0045	0.0041	0.0064	0.0038	0.0082	0.0048	0.0048	0.0072	0.0038	0.0083
64	0.0045	0.0269	0.0158	0.0778		0.0046	0.0056	0.0065	0.0032		0.0054	0.0076	0.0080	0.0032	
128	0.0044	0.0345	0.0168	0.1399		0.0083	0.0115	0.0096	0.0029		0.0093	0.0163	0.0130	0.0029	
256	0.0044	0.0405	0.0140	0.2255		0.0093	0.0147	0.0109	0.0026		0.0145	0.0209	0.0149	0.0018	

Table VII. Searching Time of the Baker and Bird Implementations for an Input String with $n = 10{,}000$ (sec)

	$\|\Sigma\| = 2$					$\|\Sigma\| = 256$					$\|\Sigma\| = 1{,}024$				
	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG
4	0.7378	1.4830	2.5807	2.0947		0.6243	0.5465	1.1186	1.1928		0.6234	0.5278	1.1170	1.1912	
8	0.6797	1.6082	1.7295	1.8555	1.0351	0.5512	0.5410	0.8097	0.7292	0.8371	0.5410	0.4561	0.8213	0.7291	0.8392
16	0.5661	1.9054	1.5287	2.9011	1.4453	0.4437	0.4258	0.6749	0.5071	0.8309	0.4453	0.4521	0.7226	0.5068	0.8218
32	0.4811	2.2097	1.5446	4.5357	1.6617	0.4380	0.3913	0.6180	0.3851	0.8089	0.4438	0.4996	0.7005	0.3852	0.8117
64	0.4484	2.8085	1.7788	8.3362		0.4326	0.5259	0.6190	0.3327		0.4368	0.6803	0.7241	0.3329	
128	0.4298	3.6996	1.8871	15.8425		0.5015	0.8849	0.7469	0.3155		0.5200	0.7592	0.7400	0.2584	
256	0.4189	5.1251	1.8810	29.5086		0.5750	1.0678	0.8553	0.3254		0.5899	0.8211	0.7992	0.2091	

Table VIII. Searching Time of the Baeza-Yates and Regnier Implementations for an Input String with $n = 1{,}000$ (sec)

	$\|\Sigma\| = 2$					$\|\Sigma\| = 256$					$\|\Sigma\| = 1{,}024$				
	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG
4	0.0054	0.0070	0.0132	0.0080		0.0040	0.0040	0.0079	0.0054		0.0043	0.0043	0.0081	0.0053	
8	0.0028	0.0038	0.0061	0.0035	0.0099	0.0029	0.0028	0.0057	0.0027	0.0091	0.0033	0.0036	0.0061	0.0027	0.0089
16	0.0024	0.0033	0.0049	0.0029	0.0106	0.0062	0.0056	0.0100	0.0018	0.0079	0.0077	0.0083	0.0111	0.0019	0.0076
32	0.0036	0.0046	0.0056	0.0028	0.0077	0.0089	0.0096	0.0126	0.0016	0.0082	0.0134	0.0129	0.0133	0.0017	0.0084
64	0.0039	0.0051	0.0053	0.0025		0.0092	0.0103	0.0124	0.0014		0.0133	0.0135	0.0157	0.0015	
128	0.0036	0.0041	0.0042	0.0021		0.0126	0.0135	0.0146	0.0013		0.0152	0.0160	0.0177	0.0013	
256	0.0027	0.0028	0.0032	0.0016		0.0161	0.0152	0.0169	0.0009		0.0168	0.0188	0.0193	0.0011	

Table IX. Searching Time of the Baeza-Yates and Regnier Implementations for an Input String with $n = 10{,}000$ (sec)

	$\|\Sigma\| = 2$					$\|\Sigma\| = 256$					$\|\Sigma\| = 1{,}024$				
	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG	AC	SH	SBOM	WM	SOG
4	0.5439	0.7037	1.3139	0.8023		0.3997	0.4006	0.7895	0.5264		0.4203	0.4231	0.8242	0.5264	
8	0.2854	0.3920	0.6764	0.4491	1.0829	0.2935	0.2951	0.7097	0.2702	0.8961	0.3295	0.3622	0.6001	0.2686	0.8842
16	0.2508	0.3342	0.5067	0.2982	1.0602	0.6342	0.5824	1.0713	0.1826	0.8140	0.8582	0.8697	1.2107	0.1826	0.7865
32	0.3637	0.4793	0.5790	0.2861	0.7748	0.9349	1.0182	1.3138	0.1590	0.8400	1.3481	1.3334	1.6672	0.1596	0.8366
64	0.4268	0.5805	0.5688	0.2745		0.9903	1.1287	1.3358	0.1522		1.3919	1.4199	1.6447	0.1521	
128	0.4702	0.4128	0.5297	0.2676		1.0743	1.1784	1.3466	0.1464		1.4205	1.4787	1.7271	0.1369	
256	0.4422	0.3725	0.5214	0.2543		1.0951	1.2021	1.3507	0.1420		1.4526	1.5299	1.8860	0.1278	