Neural, Parallel, and Scientific Computations 22 (2014) 563-593

MULTIPLE PATTERN MATCHING: SURVEY AND EXPERIMENTAL RESULTS

CHARALAMPOS S. KOUZINOPOULOS AND KONSTANTINOS G. MARGARITIS

Parallel and Distributed Processing Laboratory
Department of Applied Informatics, University of Macedonia
156 Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece

ABSTRACT. This paper presents a survey of multiple pattern matching algorithms and experimental results of the well-known Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms. The performance of the algorithms is evaluated in terms of preprocessing and searching time for randomly generated data, the genome of Escherichia coli, the FASTA Nucleic Acid (FNA) of the A-thaliana genome, the FASTA Amino Acid (FAA) of the A-thaliana genome, the Swiss Prot. Amino Acid sequence database and English language data for sets of 100 to 100.000 patterns.

Key Words pattern matching, multiple pattern matching, algorithms

1. Introduction

Multiple pattern matching is a variant of string matching that involves the location of all the positions of an input string where one or more patterns from a finite pattern set occur. It is the computationally intensive kernel of many security and network applications including information retrieval, web filtering, intrusion detection systems, virus scanners and spam filters. Moreover, in recent years there is an increased interest in string matching problems as a powerful tool in locating nucleotide or Amino Acid sequence patterns in biological sequence databases. The multiple pattern matching problem can be defined as:

Definition. Given an input string $T = t_0 t_1 \dots t_{n-1}$ *of length* n *and a finite set of* d *patterns* $P = p^0, p^1, \dots, p^{d-1}$*, where each* p^r *is a string* $p^r = p_0^r p_1^r \dots p_{m-1}^r$ *of length* m *over a finite character set* Σ*, the alphabet size is denoted as* $|\Sigma|$ *and the total size of all patterns as* $|P|$*, the task is to find all occurrences of any of the patterns in the input string.*

This paper presents a short survey on multiple pattern matching algorithms, details the Aho-Corasick [1], Set Horspool [24], Set Backward Oracle Matching [2], Wu-Manber [33] and the SOG [26] algorithms and presents experimental results in terms of preprocessing and searching time for different types of data. The algorithms

Received December 1, 2013

were chosen since they are efficient and are frequently encountered in other research papers. Aho-Corasick is a classic multiple pattern matching algorithm with a linear in n search phase in the worst and average case. Set Horspool has a sublinear search phase in the average case. Commentz-Walter [6], the algorithm upon Set Horspool is based, is substantially faster in practice than the Aho-Corasick algorithm, particularly when long patterns are involved [33, 31]. 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 [24]. The Wu-Manber algorithm was chosen as it is considered a very fast algorithm in practice [24]. Finally, Salmela-Tarhio-Kytöjoki is a recently introduced family of algorithms that has a reportedly good performance on specific types of data [17]. SOG has a linear search phase in the average case.

The lack of previously published work with extensive experimental analysis on multiple pattern matching algorithms motivated us to compare the performance of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms in terms of preprocessing and searching time and to identify a suitable and preferably fast algorithm for randomly generated data with a binary alphabet, biological sequence databases containing the building blocks of nucleotides and Amino Acids and natural language data of the English alphabet and for several problem parameters such as the total size of the pattern set, the alphabet size and the length of the input string and the patterns. To the best of our knowledge, this is the first time that the cost of the preprocessing phase is studied based on experimental results and is compared to the searching time of the specific algorithms.

2. **Related Work**

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 [24] for a randomly generated data set. The input string had a size of approximately 10 MB while the pattern set consisted of 5 to 1.000 patterns where each pattern had a length $m = 5$ to 100 characters. There, it was concluded that for relatively small pattern set sizes, Aho-Corasick had the best performance when patterns of size $m = 5$ to 15 characters and an alphabet of size $|\Sigma| = 2$ to 4 were used, Set Backward Oracle Matching was the fastest algorithm for patterns of size $m = 15$ to 100 characters and an alphabet of size $|\Sigma| = 2$ to 8 while Wu-Manber outperformed the other algorithms when the alphabet of the data set used had a size $|\Sigma| = 8$ to 64. For larger pattern sets, the Set Backward Oracle Matching algorithm was more attractive; it was the fastest algorithm when patterns of a size $m = 20$ to 100 were used and for all alphabet

sizes while the Aho-Corasick and Wu-Manber algorithms were faster for patterns of a length $m = 5$ to 20 characters.

A variant of the Wu-Manber algorithm called QWM was presented in [10] and its performance was compared to the Aho-Corasick, Commentz-Walter and the original Wu-Manber algorithm 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. The pattern sets used for the experiments of that paper consisted of 100 to 2.000 patterns. It was shown that Aho-Corasick was the fastest algorithm for data with a binary alphabet while Wu-Manber outperformed the Aho-Corasick and Commentz-Walter algorithms on randomly generated data with an alphabet of size $|\Sigma| = 4$ and on data with an English and a Chinese language alphabet. HMA, a hierarchical multiple pattern matching algorithm was introduced in [28] 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 [25] 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 32 MB. It was shown that for the specific problem parameters, Wu-Manber outperformed the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms. Moreover, the performance of the search phase of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms was evaluated. For $m = 8$ and $|\Sigma| = 256$, Wu-Manber was the fastest algorithm when up to 5.000 patterns were used while SOG outperformed the rest of the algorithms for more than 5.000 patterns. When $m = 32$ and $|\Sigma| = 4$ on the other hand, the Set Backward Oracle Matching algorithm had the best performance when up to 1.000 patterns were used while SOG was the fastest for larger pattern set sizes.

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 [18].

3. **Multiple Pattern Matching Algorithms**

A naive solution to the multiple pattern matching problem is to perform d separate searches in the input string with a string matching algorithm leading to a worst-case complexity of $\mathcal{O}(|P|)$ for the preprocessing phase and $\mathcal{O}(n|P|)$ for the search phase. While frequently used in the past, this technique is not efficient, especially when a large pattern set is involved. The modern multiple pattern matching algorithms can scan the input string in a single pass to locate all occurrences of the patterns. These algorithms are often based on string matching algorithms with some

of their functions generalized to process multiple patterns simultaneously during the preprocessing phase. Based on the way the multiple patterns are represented and the search is performed, the algorithms can generally be classified in to one of the four following approaches.

- **Prefix algorithms** The prefix searching algorithms use a *trie* to store the patterns, a data structure where each node represents a prefix u of one of the patterns. At a given position i of the input string, the algorithms traverse the trie looking for the longest possible suffix u of $t_0...t_i$ that is a prefix of one of the patterns. One of the most well known prefix multiple pattern matching algorithms is Aho-Corasick, an efficient algorithm based on Knuth-Morris-Pratt [16] that preprocesses the pattern in time linear in $|P|$ and searches the input string in time linear in n in the worst case. Multiple Shift-And, a bit-parallel algorithm generalization of the Shift-And algorithm for multiple pattern matching was introduced in [24] but is only useful for a small size of $|P|$ since the pattern set must fit in a few computer words.
- **Suffix algorithms** The suffix algorithms store the patterns backwards in a suffix trie, a rooted directed tree that represents the suffixes of all patterns. At each position i of the input string the algorithms compute the longest suffix u of the input string that is a suffix of one of the patterns. Commentz-Walter [6] combines a suffix trie with the *good suffix* and *bad character* shift functions of the Boyer-Moore [4] algorithm. A simpler variant of Commentz-Walter is Set Horspool [24], an extension of the Horspool [13] algorithm that uses only the *bad character* shift function. Suffix searching is generally considered to be more efficient than prefix searching since on average more input string positions are skipped following each mismatch.
- **Factor algorithms** The factor searching algorithms build a *factor oracle*, a trie with additional transitions that can recognize any substring (or factor) of the patterns. Dawg-Match [8] and MultiBDM [9] were the first two factor algorithms, algorithms complicated and with a poor performance in practice [24]. The Set Backward Oracle Matching and the Set Backward Dawg Matching algorithms [24] are natural extensions of the Backward Oracle Matching and the Backward Dawg Matching [7] algorithms respectively for multiple pattern matching.
- **Hashing algorithms** The algorithms following this approach use hashing to reduce their memory footprint, usually in conjunction with other techniques. Wu-Manber is based on the Horspool algorithm. It reads the input string in blocks to effectively increase the size of the alphabet and then applies a hashing technique to reduce the necessary memory space. Zhou et al. [34] proposed an algorithm called MDH, a variant of Wu-Manber for large-scale pattern sets.

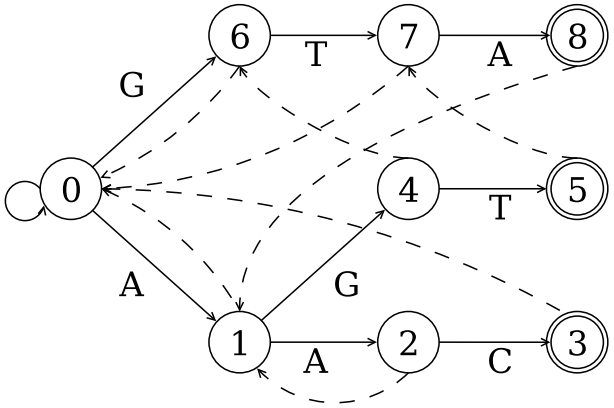

FIGURE 1. The automaton of the Aho-Corasick algorithm for the pattern set "AAC", "AGT" and "GTA"

Kim and Kim introduced in [15] a multiple pattern matching algorithm that also takes the hashing approach. The Salmela-Tarhio-Kytöjoki variants of the Horspool, Shift-Or [3] and BNDM [23] algorithms can locate *candidate* matches by excluding positions of the input string that do not match any of the patterns. They combine hashing and a technique called q-grams to increase the alphabet size, similar to the method used by Wu-Manber.

Many well known tools utilize multiple pattern matching algorithms: Snort [29] uses a variant of the Aho-Corasick algorithm to perform intrusion detection, the Wu-Manber algorithm is utilized by Agrep [32] to perform approximate string searching while Commentz-Walter is used in GNU Grep [12] when searching for the occurrences of multiple patterns in an input string.

3.1. **Aho-Corasick.** Aho-Corasick is an extension of the Knuth-Morris-Pratt 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 one or more child states by σ where σ is a matching character. Each state of the trie is labeled after a single character of a pattern $p^r \in P$. Then $L(q)$ is 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.

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

```
Function AC_Preproc_Goto ( p, m, d, Σ )
create state q_0
forall the α ∈ Σ do
  | g(q_0, α) := fail
end
newState := 0
for i := 0; i < d; 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 l := j; l < m; l := l + 1 do
  |   | create state q_current
  |   | forall the α ∈ Σ do
  |   |   | g(q_current, α) := fail
  |   | end
  |   | newState := q_current
  |   | g(state, p_l^i) := newState
  |   | state := newState
  | end
  | Output(q_current) := {p_0^i ... p_{m-1}^i}
  | Add terminal state on q_current
end
```

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 while 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 is determined that no such state exists; in that case the *supply* state of q is set to the initial state.

Let u 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 $L(f) = u\sigma$ 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 $|u\sigma| = m$. 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 $u = L(f')$. 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 is determined that no such transition exists. The construction of the *goto* function of the Aho-Corasick automaton is given in Algorithm listing 1, the computation of the *supply* function is presented in Algorithm listing 2 while the search phase of the Aho-Corasick algorithm is detailed in Algorithm listing 3. The output function returns $L(q)$ for each terminal state q and is denoted as $Output()$. An external transition that does not point to a state is denoted as $fail$. An in-depth analysis of the Aho-Corasick algorithm is presented in [24].

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(q0, α) = fail then
        g(q0, α) := q0
    else
        Supply(g(q0, α)) := q0
    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
```

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

[24]; a link 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 [11]. The implementation used for the experiments of this paper was based on code from the Streamline system I/O software layer [30]. It uses an array of size $|\Sigma|$ for each state of the automaton and a linked list to represent the transitions of the *goto* function and the *supply* function. The trie of P can then be built for all d patterns in $\mathcal{O}(|\Sigma||P|)$ time, with a total size of $\mathcal{O}(|\Sigma||P|)$. The time to pass through a transition of the *goto* function is $\mathcal{O}(|\Sigma|)$ in the worst and average case while the search phase has a cost of $\mathcal{O}(n)$ in the worst and average case.

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
```

An example of a complete Aho-Corasick automaton for the pattern set "AAC", "AGT" and "GTA" is presented in Figure 1. Assume that the *goto* function of the trie is already constructed and that the *supply* function for states $0-4$ has been computed. The *supply* state of state 5 is determined next. State 4 is the parent state of state 5 since $g(4, \text{“}T\text{”}) = 5$ and $Supply(4) = 6$, therefore the *goto* function of state 6 is considered next. Since $g(6, \text{“}T\text{”}) = 7$ then $Supply(5)$ can be set to 7. If there was no outgoing transition from state 6 by "T" then $Supply(6)$ would be checked next for an outgoing transition to another state by "T" and so on, until one such state is found or is determined that no such state exists.

3.2. **Set Horspool.** The Set Horspool algorithm combines a deterministic finite state pattern matching machine with the *shift* function of the Horspool 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 g, created from each pattern $p^r \in P$ 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.

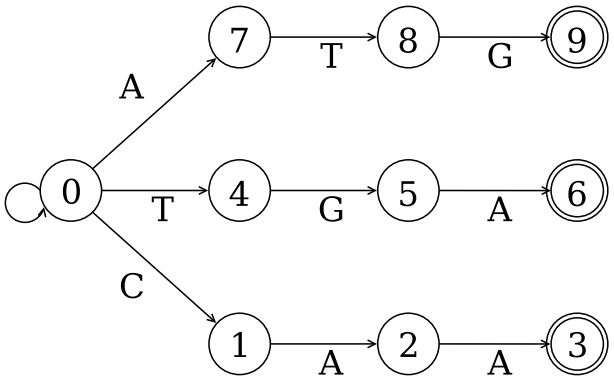

FIGURE 2. The automaton of the Set Horspool algorithm for the reversed pattern set "AAC", "AGT" and "GTA"

The *goto* function and the *shift* function are computed during the preprocessing phase. To build the *goto* function, the trie is depth-first traversed and extended for each character of the patterns $\in P$ while at the same time the outgoing transitions to each state q of the trie are created. The *bad character* shift is denoted as $bc(\sigma)$ and is computed for each different character $\sigma \in \Sigma$ as the maximum $|L(q)|$ where q is a state labeled by σ. If no such character exists in any pattern, the *bad character* shift for σ is set to m.

ALGORITHM 4: The construction of the bad character shift of the Set Horspool algorithm

```
Function SH_Bad_Character_Shift ( p, m, d, |Σ| )
for i := 0; i < |Σ|; i := i + 1 do
    bc[i] := m
end
for i := 0; i < d; 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 of $t_0 \dots t_i$ 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 i 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 computation of the *bad character* shift function of the Set Horspool

algorithm is depicted in Algorithm listing 4 while the search phase is detailed in Algorithm listing 5. The computation of the *goto* function is identical to that of the Aho-Corasick algorithm as already given in Algorithm listing 1. An example of a complete Set Horspool automaton for the reversed pattern set "AAC", "AGT" and "GTA" is presented in Figure 2.

ALGORITHM 5: The search phase of the Set Horspool algorithm

```
Function SH_Search ( t, n )
i := m − 1
while i < n do
    j := 0; state = q0
    while j < m AND (newState := g(state, t_{i−j}) ≠ fail) do
        state := newState; j := j + 1
        if Output(state) is not empty then
            report match at i
        end
    end
    i := i + bc(t_i)
end
```

The implementation of Set Horspool uses an array of size $|\Sigma|$ for each state of the automaton and a linked list to represent the transitions of the *goto* function. The construction of the trie and the shift function requires $\mathcal{O}(|\Sigma||P|)$ time and space while the search phase of the algorithm is $\mathcal{O}(nm)$ worst case time or sublinear on average.

3.3. **Set Backward Oracle Matching.** The Set Backward Oracle Matching algorithm extends the Backward Oracle Matching 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. The automaton consists of a *goto* function that uses at most $|P|$ external transitions. The transitions map a state q and a pattern character into the next state. The oracle is based on the notion of *weak factor recognition*; each state of the Set Backward Oracle Matching automaton can have several incoming links such that *at least* any factor of a pattern can be recognized.

The *goto* function of the Set Backward Oracle Matching algorithm is constructed during the preprocessing phase from the set of the reversed patterns, similar to the automaton of the Set Horspool algorithm. 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 build the *goto* function, a *supply* function is used that associates each state q with a *supply* state $Supply(q)$. Initially, $Supply(q_0)$ is set to *fail*. Assume that the *goto* and *supply* functions for all states up to the parent state q_{parent} of q were already computed and that state q is created and labeled by σ. A pointer k points to $Supply(q_{parent})$. If $k = fail$, then $Supply(q) = q_0$. As long as k is defined and $g(k, \sigma) = fail$ then a transition from state k by σ to the current state is created and k is updated to point to $Supply(k)$. If k is defined and an external transition from state k by σ exists, $Supply(q)$ is set to $g(k, \sigma)$ and the construction of the external transitions for q ends.

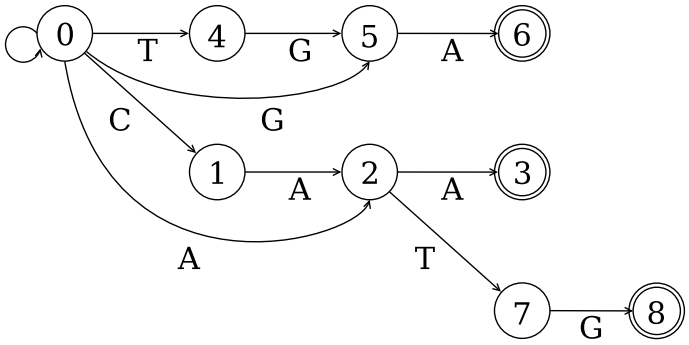

FIGURE 3. The automaton of the Set Backward Oracle Matching algorithm for the reversed pattern set "AAC", "AGT" and "GTA"

ALGORITHM 6: The construction of the factor oracle of the Set Backward Oracle Matching algorithm

```
Function SBOM_Preproc ( p, m, d, Σ )
create state q_0
set Supply(q_0) := fail
forall the α ∈ Σ do
| g(q_0, α) := fail
end
newState := 0
for i := 0; i < d; i := i + 1 do
| j := m − 1; state := q_0
| while newState := g(state, p_j^i) ≠ fail do
| | state := newState; j := j − 1
| end
| for l := j; l ≥ 0; l := l − 1 do
| | create state q_current
| | forall the α ∈ Σ do
| | | g(q_current, α) := fail
| | end
| | newState := q_current
| | g(state, p_l^i) := newState
| | k := Supply(state)
| | while k ≠ fail AND g(k, p_l^i) = fail do
| | | g(k, p_l^i) := newState
| | | k := Supply(k)
| | end
| | if k ≠ fail then
| | | Supply(newState) := g(k, p_l^i)
| | else
| | | Supply(newState) := q_0
| | end
| | state := newState
| end
| if terminal state on q_current does not exist then
| | F(q) := 0
| | Add terminal state on q_current
| else
| | F(q_current) := F(q_current) ∪ {i}
| end
end
```

During the search phase, the algorithm reads backwards the longest suffix u of the input string that is also a suffix of *any* pattern. Assume that a mismatch occurs at position i of the input string when reading character σ. The factor oracle can not determine with certainty that a substring is a factor of one of the patterns but can recognize if a substring is not, therefore σu is not a factor of *any* $p^r \in P$. The oracle can then be safely shifted past i. 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 they correspond. 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 preprocessing phase of the Set Backward Oracle Matching algorithm is detailed in Algorithm listing 6 while the search phase is presented in Algorithm listing 7. An example of a complete Set Backward Oracle Matching oracle for the reversed pattern set "AAC", "AGT" and "GTA" is depicted in Figure 3. The substring "CAT" would be recognized by the oracle although it is not a factor of any pattern.

ALGORITHM 7: The search phase of the Set Backward Oracle Matching algorithm

```
Function SBOM_Search ( p, t, m, n )
i := m − 1
while i < n do
    state := q_0, j := 0
    while j < m AND (newState := g(state, t_{i−j}) ≠ fail) do
        state := newState; j := j + 1
    end
    if F(state) is not empty AND j = m then
        if Verify all patterns in F(state) against the input string then
            report match at i − m + 1
        end
        i := i + 1
    else
        i := i + m − j
    end
end
```

The implementation of Set Backward Oracle Matching uses an array of size $|\Sigma|$ for each state of the oracle and a linked list to represent the transitions of the *goto* and *supply* functions. The set of indices F is stored on each terminal state using an array of size d. n auxiliary table of size d is also maintained to map the patterns to

their corresponding indices The oracle is created during the preprocessing phase in $\mathcal{O}(|\Sigma||P|)$ for all d patterns of the pattern set using a size of $\mathcal{O}(|\Sigma||P|)$. The search phase complexity of the algorithm is $\mathcal{O}(n|P|)$ worst case time or sublinear on average.

3.4. **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 with the size of the pattern set 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 $|\Sigma|^B$.

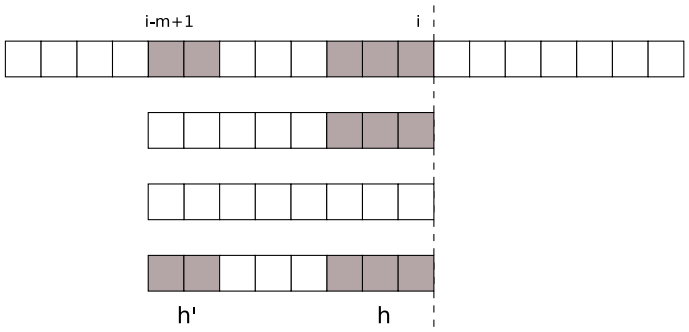

FIGURE 4. Comparing the suffix and prefix of the search window of the Wu-Manber algorithm

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. Since two or more blocks can have the same hash value, the search window can be safely skipped by the minimum shift between them. 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. As recommended in [33], a good value for B is $\log_{|\Sigma|} 2|P|$ although usually B could be set as equal to 2 for a small pattern set size or to 3 otherwise. The original paper also recommends setting $B' = 2$.

ALGORITHM 8: The preprocessing phase of the Wu-Manber algorithm

```
Function WM_Preproc ( p, m, d, B, B' )
Initialize all elements of SHIFT to m − B + 1
for i := 0; i < d; i := i + 1 do
    for q := m; q ≥ B; q := q − 1 do
        h := h1(p^i_{q−B−1} ... p^i_{q−1})
        shiftlen := m − q
        SHIFT[h] := MIN(SHIFT[h], shiftlen)
        if shiftlen = 0 then
            h' := h2(p^i_0 ... p^i_{B'−1})
            HASH[h] := HASH[h] ∪ {i}
            PREFIX[i] := h'
        end
    end
end
```

Assume that the search window is aligned with the input string at position i and that h is the hash value of the B-character suffix of $t_0 \dots t_i$. Then the *SHIFT* table is used to determine the number of safe shift positions. If *SHIFT[h]* > 0 then the search window is shifted by *SHIFT[h]* positions. If, on the other hand, *SHIFT[h]* $= 0$, the suffix of the input string potentially matches the suffix of *some* patterns of the pattern set and thus it must be determined if a complete match occurs at that position. The hash value h' of the B'-character prefix of the input string starting at position $i-m+1$ is then computed. For each pattern p^r with the same hash value h of its B-character suffix, it is checked if *PREFIX[p]* matches with h'. If both the prefix and the suffix of the search window match with the prefix and suffix of some $p^r \in P$, then the corresponding patterns are compared directly with the input string. The preprocessing phase of the Wu-Manber algorithm is detailed in Algorithm listing 8 while the search phase is presented in Algorithm listing 9.

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 [24]. For the experiments of this paper, the algorithm was implemented with a block size of $B = 3$ and $B' = 2$ while hash values h and h' were calculated by bit-shifting the ASCII values of the characters of the patterns and the input string to the left by *bitshift* positions. The value of *bitshift* was set to 2.

ALGORITHM 9: The search phase of the Wu-Manber algorithm

```
Function WM_Search ( p, t, m, n, B, B' )
i = m − 1
while i < n do
    h := h1(t_{i−B} ... t_i)
    if SHIFT[h] > 0 then
        i := i + SHIFT[h]
    else
        h' := h2(t_{i−m+1} ... t_{i−m+1+B'−1})
        forall the pattern indices r stored in HASH[h] do
            if PREFIX[r] = h' then
                Verify the pattern corresponding to r directly against the input
                string
            end
        end
        i := i + 1
    end
end
```

Finally, the verification of the patterns to the input string was performed using the *memcmp()* function of *string.h*. The cost of the implementation is as follows. To calculate the values of the *SHIFT*, *HASH* and *PREFIX* tables during the preprocessing phase, the algorithm requires an $\mathcal{O}(|P|)$ 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 d 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 $d \times \sum_{i=0}^{B-1} |\Sigma| \times (2^{bitshift})^i$ space for a space complexity of $\mathcal{O}(d \times \sum_{i=0}^{B-1} |\Sigma| \times (2^{bitshift})^i)$.

In the worst case for the searching phase of the Wu-Manber algorithm, the input string and $m-1$ characters of all d 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 since *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 d patterns will be compared directly with the input string using the *memcmp()* function. The worst case searching time of Wu-Manber is given in [5] as $\mathcal{O}(n \log_{|\Sigma|}(|P|)d(m-1))$. In [22]

the lower bound for the average time complexity of exact multiple pattern matching algorithm is given as $\Omega(n \log_{|\Sigma|}(|P|)/m)$ and according to [5] the searching phase of the Wu-Manber algorithm is optimal in the average case for a time complexity of $\mathcal{O}(n \log_{|\Sigma|}(|P|)/m)$. In [20] the average time complexity of Wu-Manber was also estimated as $\mathcal{O}(\frac{n}{(m-B+1)\times(1-\frac{(m-B+1)\times d}{2\times|\Sigma|^B})})$.

3.5. **SOG.** The SOG algorithm extends the Shift-Or [3] string matching algorithm to perform multiple pattern matching in linear time on average. SOG is a bit-parallel algorithm simulating a non-deterministic 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. When a candidate match is found at a given position of the input string, 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 big 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 if 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.

ALGORITHM 10: The preprocessing phase of the SOG algorithm

```
Function SOG_Preproc ( p, m, d, B )
for i := 0; i < d; i := i + 1 do
    sv := 0
    Create hs' for pattern p^r
    for j := 0; j < m − B; j := j + 1 do
        h := h_1(p_j^i ... p_{j+B−1}^i)
        V[h] := V[h] ∧ (2^m − (1 ≪ sv))
        sv := sv + 1
    end
end
```

During the preprocessing phase, a hash value h is assigned to each different B-character block of the patterns using a hash function $h_1()$ by bit-shifting the ASCII values of the characters. 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:

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

To improve the efficiency of the hashing method described above, a two-level hashing technique [21] 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. In [25] is stated that the use of two-level hashing significantly improves the performance of the algorithm when less than 100.000 patterns are involved.

ALGORITHM 11: The search phase of the SOG algorithm

```
Function SOG_Search ( text, n, B )
E := 2^m
for i := 0; i < n − B + 1; i := i + 1 do
    E = (E ≪ 1) ∨ V[h_1(t_i ... t_{i+B−1})]
    if E ∧ 2^{m−B} then
        continue
    end
    Verify the patterns using two-level hashing
end
```

To search for the occurrence of the patterns in the input string, an m-bit variable E is used where each bit is initialized to 1. For each position $0 \leq i < n - B + 1$ of the input string, a hash value h is assigned to the character block $t_i \dots t_{i+B-1}$ using function $h_1()$ again. E is then updated with the following formula:

$$E = (E \ll 1) \vee V[h]$$

If the $(m-B)^{th}$ bit of E is equal to 0, then a *candidate* match of one of the patterns in the pattern set occurs starting from position $i - m + B$ of the input string. To verify the candidate match, the ordered table is examined using binary search for the hash value hs' of the input string characters $t_{i-m+B} \dots t_{i+B-1}$. For the experiments of this paper, SOG was implemented using a block size of $B = 3$. The preprocessing phase of the SOG algorithm is presented in Algorithm listing 10 while the search phase is given in Algorithm listing 11.

The preprocessing time of the SOG algorithm is $\mathcal{O}(|P|)$ with an $\mathcal{O}(|\Sigma|^B + |P|)$ 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 $\mathcal{O}(n|P|)$. If the pattern rows have a different hash value, the verification time is $\mathcal{O}(n(\log m + m))$ in the worst case. The filtering phase is linear in n in the worst and average case. The combined filtering and verification phase is then $\mathcal{O}(n|P|)$ 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.

TABLE 1. Known theoretical space, preprocessing and worst and average searching time complexity of the presented multiple pattern matching algorithms

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

Table 1 summarizes the theoretical space, preprocessing and searching time complexity of the presented multiple pattern matching algorithms.

4. **Experimental Methodology**

The parameters that describe the performance of multiple pattern matching algorithms are the size of input string n, the size of the pattern set d, the length of the patterns m and the size $|\Sigma|$ of the alphabet used.

To evaluate the performance of the multiple pattern matching algorithms, 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 set while the searching time is the total time in seconds an algorithm uses to locate all occurrences of any pattern from the pattern set in the input string. Both times were measured using the MPI_Wtime function of the Message Passing Interface since it has a better resolution than the standard *clock*() function of *time.h*.

The data set was similar to the sets used in [14, 19, 27]. It consisted of randomly generated input strings of a binary alphabet, the genome of Escherichia coli from the Large Canterbury Corpus, the Swiss Prot. Amino Acid sequence database, the FASTA Amino Acid (FAA) and FASTA Nucleic Acid (FNA) sequences of the A-thaliana genome and natural language input strings of the English alphabet:

- Randomly generated input strings of size $n = 4.000.000$ with a binary alphabet. The alphabet used was $\Sigma = \{0, 1\}$.
- The genome of Escherichia coli from the Large Canterbury Corpus with a size of $n = 4.638.690$ characters and the FASTA Nucleic Acid (FNA) of the A-thaliana genome with a size of $n = 116.237.486$ characters. The alphabet $\Sigma = \{a, c, g, t\}$ of both genomes consisted of the four nucleotides used to encode DNA.
- The FASTA Amino Acid (FAA) of the A-thaliana genome with a size of $n = 10.830.882$ characters and the Swiss Prot. Amino Acid sequence database with a

size of $n = 177.660.096$ characters. The alphabet $\Sigma = \{a, c, d, e, f, g, h, i, k, l, m, n, p, q, r, s, t, v, w, y\}$ used by the databases consisted of 20 different characters.

- The CIA World Fact Book from the Large Canterbury Corpus. The input string had a size of $n = 1.914.500$ characters and an alphabet of size $|\Sigma| = 128$ characters.

The pattern set was created from subsequences of the corresponding input string, consisting of 100 to 100.000 patterns with each pattern having a length of $m = 8$ and $m = 32$ characters. The subsequences were chosen for at least $\min\{d, \lfloor \frac{n}{m} \rfloor\}$ matches.

The experiments were executed locally on an Intel Core 2 Duo CPU with a 3.00GHz clock speed and 2 Gb of memory, 64 KB L1 cache and 6 MB L2 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**

In the previous Sections, the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG multiple pattern matching algorithms were presented and the experimental methodology was discussed. In this Section, the performance of the algorithms is evaluated in terms of preprocessing and searching time for different sets of data.

Figures 5 to 8 present the time used by the algorithms to preprocess the available pattern sets. Each pattern had a length of $m = 8$ and $m = 32$ characters and an alphabet size $|\Sigma|$ of $2, 4, 20$ and 128 characters while the sets consisted of 100 to 100.000 distinct patterns. When a pattern length of $m = 8$ and a binary alphabet were used, there were only $2^8 = 256$ different patterns while for a pattern length of $m = 8$ and an alphabet of $|\Sigma| = 4$ there was a maximum of $4^8 = 65536$ different patterns. The time of the algorithms to preprocess the pattern set of the FASTA Nucleic Acid and the FASTA Amino Acid database was similar to that of the E.coli genome and the Swiss Prot. Amino Acid database respectively and thus the corresponding Figures were omitted.

The time of the multiple pattern matching algorithms to locate all the occurrences of any pattern from the pattern set in randomly generated binary alphabet input strings, the E.coli genome, the FASTA Nucleic Acid of the A-thaliana genome, the Swiss Prot. Amino Acid sequence database, the FASTA Amino Acid of the A-thaliana genome as well as in input strings with an English alphabet is depicted in Figures 9 to 14. All Figures have a logarithmic horizontal axis and a linear vertical. As can

be seen from the Figures, varying parameters such as the size of the pattern set, the length of the patterns and the input string as well as the alphabet size can affect the performance of the preprocessing and the searching phase and the overall performance of the presented algorithms in different ways. The experimental study proved that no algorithm is the best for all values of the problem parameters.

The presented multiple pattern matching algorithms can generally be classified into two categories based on their performance, as depicted in Figures 5 to 14 and on their theoretical time complexity as reported in the original papers and summarized in Table 1; the trie-based category of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms and the hash-based category of the Wu-Manber and SOG algorithms. All algorithms within the same category share common characteristics, as detailed below.

5.1. **Preprocessing.** The time of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms to preprocess the pattern set was expected theoretically to increase linearly in the size d of the pattern set, the length m of each pattern and the alphabet size $|\Sigma|$. As can be seen in Figures 5 to 8, all three algorithms had a similar behavior in terms of preprocessing time, albeit Aho-Corasick was slower than the Set Horspool and Set Backward Oracle Matching algorithms to preprocess the available pattern sets. The experimental results also confirmed that the time to preprocess the pattern set increased linearly in d for patterns of a length $m = 8$ and 32 characters with an alphabet of size $|\Sigma|$ of $2, 4, 20$ and 128 characters. It is worth noting the significant increase in the time to construct the *supply* function *Supply* of Aho-Corasick when sets of more than 2.000 patterns were used for all alphabet sizes and pattern lengths, that resulted in a proportional increase of the preprocessing time of the algorithm.

The same Figures also depict the dependent relationship between the preprocessing time of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms and the length m of the patterns. Quadrupling m from 8 to 32 characters resulted in a proportional increase in the time to preprocess the pattern set. The highest increase rate of the preprocessing time of the algorithms was observed when Aho-Corasick, Set Horspool and Set Backward Oracle Matching were used on patterns with a small alphabet size, including randomly generated sets with a binary alphabet and sets constructed from the genome of E.coli. Generally it can be concluded that for all types of pattern sets, the preprocessing time of Aho-Corasick increased in m with a higher rate comparing to the Set Horspool and Set Backward Oracle Matching algorithms.

The time of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms to preprocess the pattern set also increased linearly in $|\Sigma|$ for most types of

pattern sets. It is worth noting that the preprocessing time of Aho-Corasick actually decreased when used on sets with an English alphabet, contradicting the theoretical preprocessing time of the algorithm.

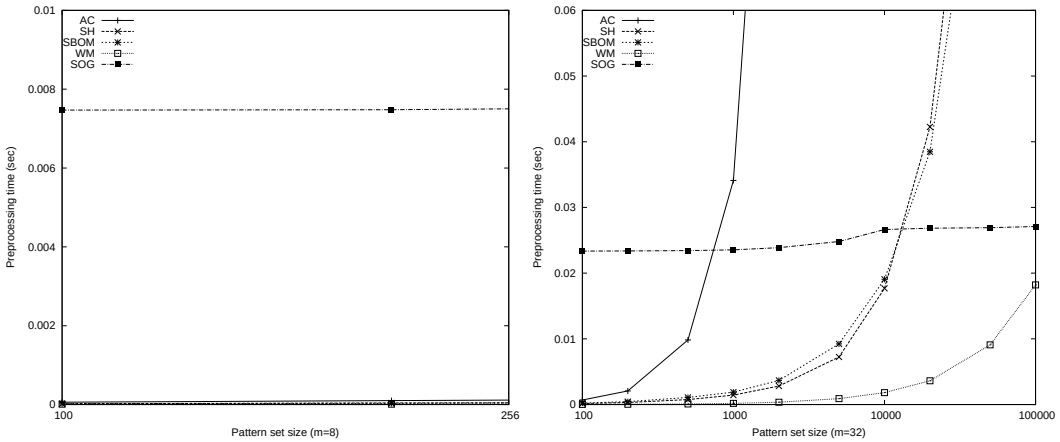

FIGURE 5. Preprocessing time of the algorithms for randomly generated data with $|\Sigma| = 2$

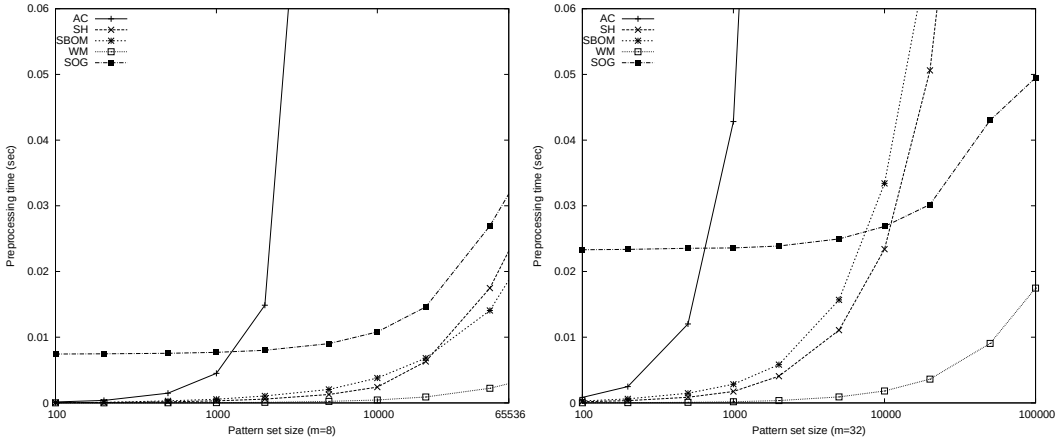

FIGURE 6. Preprocessing time of the algorithms for the E.coli genome with $|\Sigma| = 4$

Wu-Manber had the fastest preprocessing phase between the presented algorithms, as depicted in Figures 5 to 8. The time spent during the preprocessing phase of the Wu-Manber and SOG algorithms was expected to increase linearly in the size d of the pattern set and the length m of the patterns, since both have a theoretical preprocessing time of $\mathcal{O}(|P|)$. The experimental results confirmed that the time of the Wu-Manber algorithm to preprocess the pattern set increased linearly in d for all types of data. The preprocessing phase of SOG appears in the Figures to be roughly constant in d when used on sets of up to 2.000 patterns and linear in d for larger pattern set sizes. This can be explained by the relatively high cost in terms of preprocessing time to setup the necessary data structures of the algorithm.

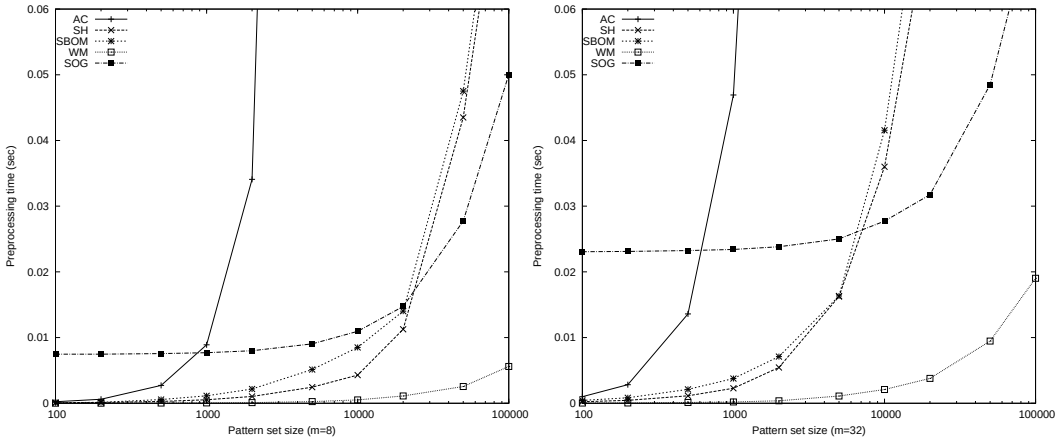

FIGURE 7. Preprocessing time of the algorithms for the Swiss Prot. Amino Acid database with $|\Sigma| = 20$

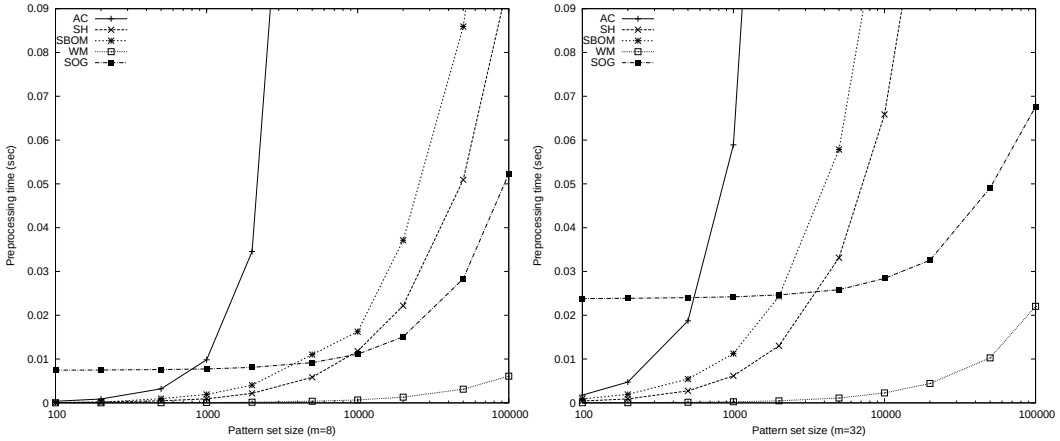

FIGURE 8. Preprocessing time of the algorithms for English alphabet data with $|\Sigma| = 128$

The performance of the preprocessing phase of the hash-based algorithms was also affected by the length m of the patterns used. The preprocessing time of both Wu-Manber and SOG algorithms increased linearly in m for all types of data. Finally, the time of Wu-Manber and SOG to preprocess the pattern set was generally constant in $|\Sigma|$, confirming in practice the theoretical preprocessing time of the algorithms.

5.2. **Searching.** Table 1 summarizes the theoretical time of the presented multiple pattern matching algorithms to scan the input string; $\mathcal{O}(n)$ in the worst and average case for Aho-Corasick, $\mathcal{O}(nm)$ and sublinear in the worst and average case respectively for Set Horspool and $\mathcal{O}(n|P|)$ and sublinear for Set Backward Oracle Matching. As depicted in Figures 9 to 14, the time spent during the search phase of the three presented trie-based algorithms to scan the input string increased linearly in the size of the pattern set, although for Aho-Corasick and Set Horspool was expected to be constant in d.

The time spent during the search phase of Aho-Corasick and Set Horspool increased linearly in m. The higher increase rate for both algorithms was observed for data with a binary alphabet and an alphabet of size $|\Sigma| = 4$ as well as for English alphabet data, especially when sets of more than 10.000 patterns were used. The performance in terms of searching time of the Set Backward Oracle Matching algorithm on the other hand improved when used on patterns with a size $m = 32$ as opposed to $m = 8$ for most types of data. It is interesting that on English alphabet data and similar to the Aho-Corasick and Set Horspool algorithms, the searching time of Set Backward Oracle Matching actually increased linearly in m.

The time of the Aho-Corasick algorithm to search the input string for all occurrences of any pattern from the pattern set was roughly constant in the size $|\Sigma|$ of the alphabet for most types of data. The performance of Set Horspool and Set Backward Oracle Matching actually improved when used to search data sets with a larger alphabet size, while not expected by their theoretical average searching time complexity. This is clearly shown in Figures 11 and 12; although the input string of the Swiss Prot. Amino Acid database was significantly larger than the input string of the FASTA Nucleic Acid database, the searching time of the Set Horspool algorithm decreased for a set of 20.000 patterns with $m = 32$ from over 12 to 7 seconds while the searching time of the Set Backward Oracle Matching decreased for the same pattern set from 1.14 to 0.58 seconds.

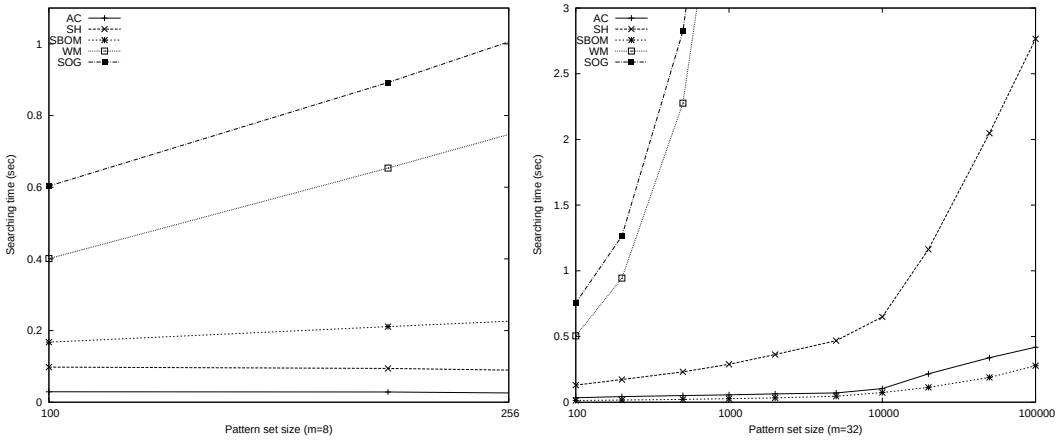

FIGURE 9. Searching time of the algorithms for randomly generated input strings with $|\Sigma| = 2$

The theoretical time complexity of the Wu-Manber algorithm to locate all the occurrences of the patterns in the input string is $\mathcal{O}(n|P|\log_{|\Sigma|}|P|)$ in the worst and $\mathcal{O}(\frac{n\log_{|\Sigma|}|P|}{m})$ in the average case while the theoretical searching time of SOG is $\mathcal{O}(n|P|)$ in the worst and $\mathcal{O}(n)$ in the average case. Based on the theoretical time of Wu-Manber, it was expected that the time of the algorithm to scan the input string would increase linearly in the size n of the input string and the size d of the pattern

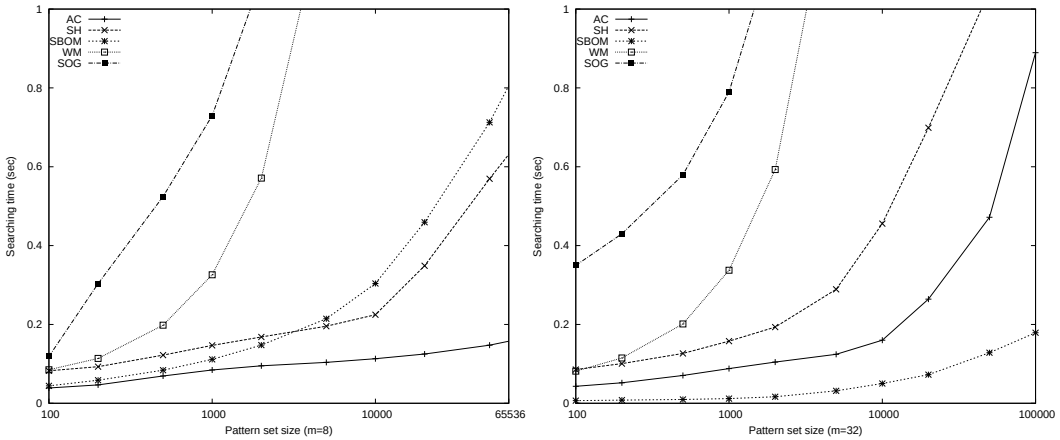

FIGURE 10. Searching time of the algorithms for the E.coli genome with $|\Sigma| = 4$

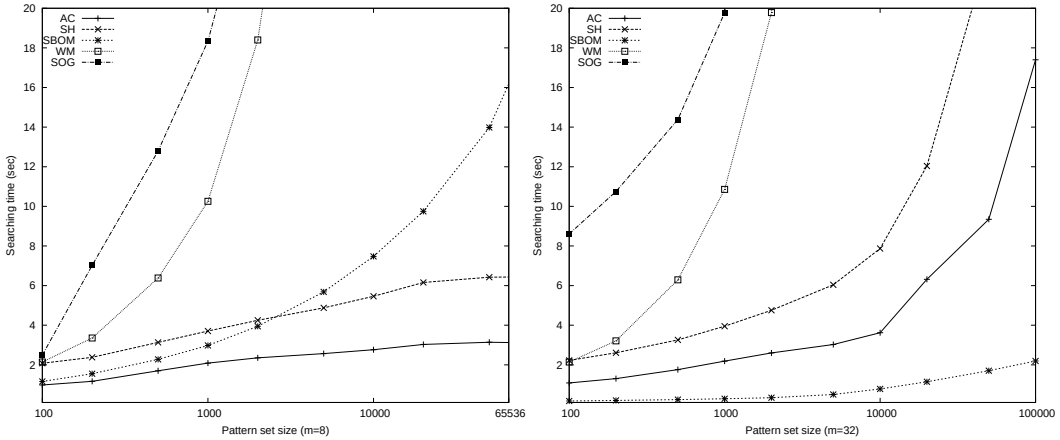

FIGURE 11. Searching time of the algorithms for the FASTA Nucleic Acid database with $|\Sigma| = 4$

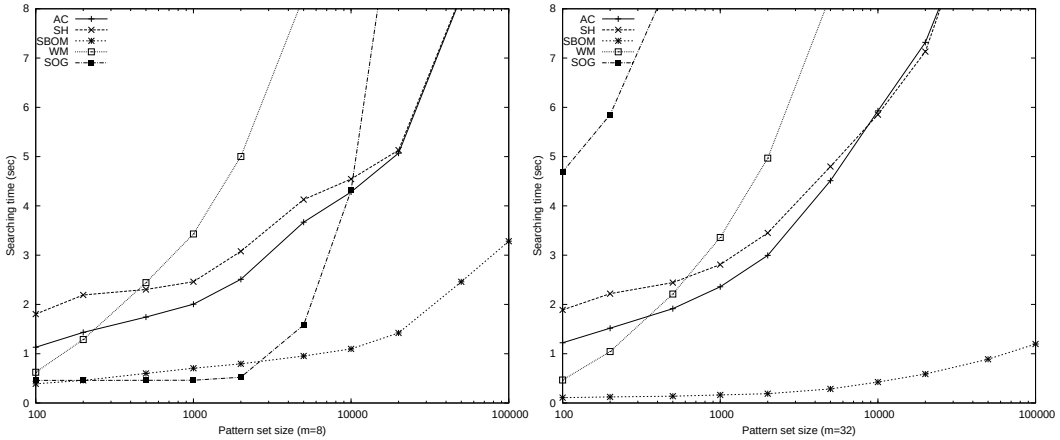

FIGURE 12. Searching time of the algorithms for the Swiss Prot. Amino Acid database with $|\Sigma| = 20$

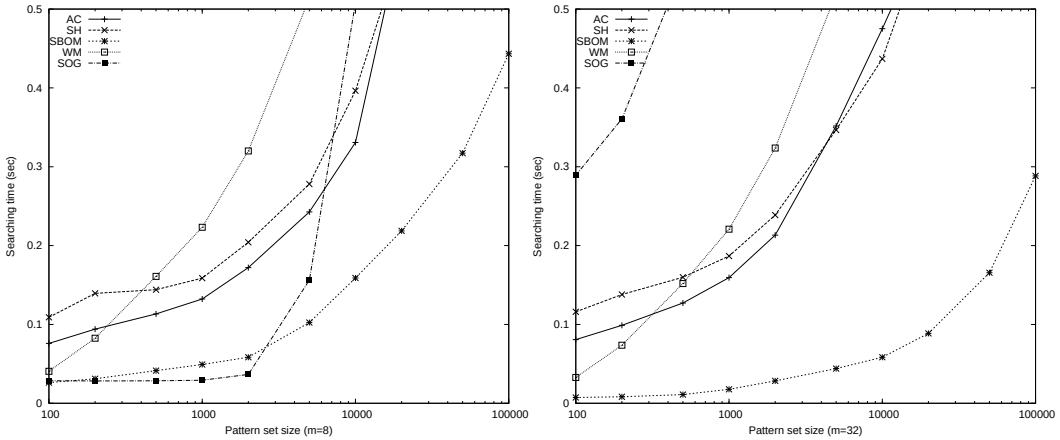

FIGURE 13. Searching time of the algorithms for the FASTA Amino Acid database with $|\Sigma| = 20$

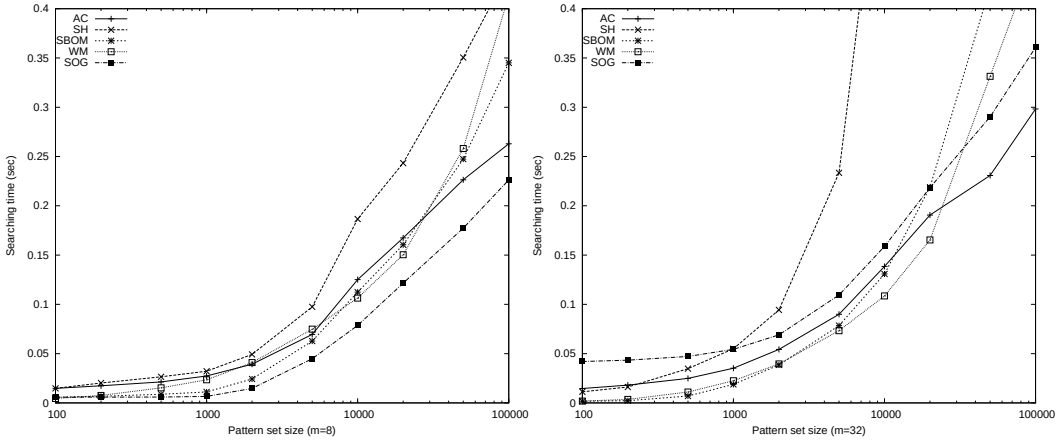

FIGURE 14. Searching time of the algorithms for English alphabet input strings with $|\Sigma| = 128$

set and decrease exponentially in the size m of the patterns and the size $|\Sigma|$ of the alphabet in the average case. The searching time of the SOG algorithm on the other hand was expected to increase linearly in n and be constant in d and $|\Sigma|$ in the average case.

Figures 9 to 14 confirm that the searching time of the Wu-Manber algorithm increased linearly in the size d of the pattern set for all alphabet and pattern sizes used. The searching time of the SOG algorithm also increased linearly in d, close to the worst case searching complexity of the algorithm. As a general remark, it can be observed that the rate of increase of the searching time of the hash-based algorithms in the size d of the pattern set was slightly higher than the corresponding rate of increase of the trie-based algorithms for all types of data.

Although not expected by the theoretical average time of the Wu-Manber algorithm, its searching time was roughly constant in the size m of the patterns when data with an alphabet of size $|\Sigma| = 20$ and 128 were used while actually increased

when used on randomly generated binary alphabet data, the E.coli genome and the FASTA Nucleic Acid database. The time of the SOG algorithm to search the input string for all occurrences of any pattern from the pattern set increased significantly when patterns of a size $m = 32$ were used as opposed to patterns of a size $m = 8$, also close to its worst case theoretical searching complexity. The increase in the searching time was higher when data with a larger alphabet size were used, including the Swiss Prot. Amino Acid database, the FASTA Amino Acid database as well as English alphabet data.

Finally, similar to Set Horspool and Set Backward Oracle Matching, the searching time of the Wu-Manber and SOG algorithms decreased in the alphabet size $|\Sigma|$. This observation confirms the theoretical average time complexity of the Wu-Manber algorithm but was not expected from the theoretical complexity of the SOG algorithm.

5.3. **Overall Comparison.** Aho-Corasick was the fastest algorithm to scan the input string when data sets with an alphabet of a size $|\Sigma| = 2$ and 4 and patterns of a length $m = 8$ were used; randomly generated input strings with a binary alphabet, the E.coli genome and the FASTA Nucleic Acid database. The inadequate performance of Aho-Corasick on data with a large alphabet size or on patterns with a length of $m = 32$ characters was generally due to the slow preprocessing phase of the algorithm on these types of data, as already discussed. The fast running time of Aho-Corasick on the other hand on data sets with a small alphabet size and a pattern length of $m = 8$ was expected by the experimental maps presented in [24] where Aho-Corasick was faster than the Wu-Manber and the Set Backward Oracle Matching algorithms when data with a binary alphabet size were used. Similar results for multiple pattern matching were reported in [10], where for a binary alphabet data set the Aho-Corasick outperformed the Commentz-Walter and Wu-Manber algorithms.

The Set Horspool algorithm had a moderate performance on most types of data. It had a similar searching time to Aho-Corasick albeit was slightly slower on each type of data. Set Backward Oracle Matching on the other hand was one of the fastest algorithms in terms of searching time. It outperformed the Aho-Corasick, Set Horspool, Wu-Manber and SOG algorithms on most types of data where patterns of a length $m = 32$ were used. It was also the fastest algorithm when patterns of a length $m = 8$ were used on the Swiss Prot. and FASTA Amino Acid databases for sets of more than 5.000 patterns. In the experiments presented in [24], Set Backward Oracle Matching was the fastest algorithm for 1.000 patterns and for a pattern size m of more than 20 characters which also holds true for the experiments presented in this paper.

Although Wu-Manber had the fastest preprocessing phase, it was one of the slowest algorithms to search the input string for the occurrences of any pattern from

the pattern set on most types of data. It was faster than the rest of the algorithms in terms of searching time only when English alphabet data sets were used together with patterns of a length $m = 32$. Based on the average searching time complexity of the algorithms as given in Table 1, it was expected for Wu-Manber to outperform the SOG algorithm on most types of data, with SOG being faster only when patterns of a length $m = 8$ and an alphabet size of $|\Sigma| = 2$ and 4 were used. In practice though, Wu-Manber was faster than SOG for data with an alphabet of size $|\Sigma| = 2$ and 4 or when patterns of a length $m = 32$ were used. Finally, SOG outperformed the Aho-Corasick, Set Horspool, Set Backward Oracle Matching and the Wu-Manber algorithms on the Swiss Prot. and FASTA Amino Acid databases and on data sets with an English alphabet when sets of up to 2.000 patterns were used with a length of $m = 8$ characters.

6. **Conclusions**

This paper presented a survey of multiple pattern matching algorithms and experimental results of the well-known Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms. The performance of the algorithms was evaluated in terms of preprocessing and searching time for randomly generated data, the genome of Escherichia coli, the FASTA Nucleic Acid (FNA) of the A-thaliana genome, the FASTA Amino Acid (FAA) of the A-thaliana genome, the Swiss Prot. Amino Acid sequence database and English language data for sets of 100 to 100.000 patterns.

In the previous Sections it was confirmed that the performance of the preprocessing phase of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms depended on the total size $|P|$ of the pattern set and the size $|\Sigma|$ of the alphabet used while the performance of the preprocessing phase of the Wu-Manber and SOG algorithms depended on $|P|$, as expected by their theoretical time complexity.

Based on the experimental results it was also concluded that the searching time of the trie-based algorithms increased linearly in the size d of the pattern set. The searching time of Aho-Corasick and Set Horspool also increased linearly in m while that of Set Backward Oracle Matching decreased when used on patterns with a length $m = 32$ as opposed to $m = 8$ for most types of data. When used on English alphabet pattern sets and input strings, the searching time of Set Backward Oracle Matching actually increased in m. Moreover it was shown that the time of the Aho-Corasick algorithm to search the input string for all occurrences of any pattern from the pattern set was roughly constant in the size of the alphabet while the performance of Set Horspool and Set Backward Oracle Matching improved when used to search data sets with a larger alphabet size $|\Sigma|$.

The searching time of the hash-based algorithms increased linearly in the size d of the pattern set. For Wu-Manber and based on the average theoretical complexity of the algorithm as presented in Table 1, it was expected for the searching time to decrease exponentially in m. In practice though it was observed that the searching time of the algorithm increased in the length m of the patterns when data with a small alphabet size $|\Sigma|$ were used and was constant for data with an alphabet size of at least 20 characters. While not expected by its theoretical average searching time and similar to its preprocessing phase, the time of the SOG algorithm to search the input string for all occurrences of any pattern from the pattern set increased significantly in m. Finally, the time of the hash-based algorithms to scan the input string for the occurrences of all patterns decreased as the alphabet size $|\Sigma|$ increased, although for the SOG algorithm it was expected by its theoretical time to be constant in $|\Sigma|$.

It was generally discussed that for randomly generated data with a binary alphabet, the E.coli genome and the FASTA Nucleic Acid database, the trie-based algorithms outperformed the hash-based algorithms in terms of searching time for all pattern set sizes and pattern lengths. For these types of data, Aho-Corasick was the fastest algorithm when patterns of a length $m = 8$ were used while Set Backward Oracle Matching outperformed the rest of the algorithms for patterns of a length $m = 32$. For the Swiss Prot. and the FASTA Amino Acid databases, SOG was the fastest algorithm when sets of up to 2.000 patterns were used, with the patterns having a length of $m = 8$ characters. For the same data and for sets of more than 2.000 patterns or when patterns with a length of $m = 32$ characters were used, Set Backward Oracle Matching was the fastest among the presented algorithms. Finally, when English alphabet data were used, the performance in terms of searching time of the algorithms converged, with the SOG algorithm being slightly faster when patterns of a length $m = 8$ were used and the Set Backward Oracle Matching, Wu-Manber and the Aho-Corasick algorithms outperforming the rest of the algorithms when patterns of a length $m = 32$ were used.

REFERENCES

[1] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. 1725:758–758, 1999.

[3] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

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

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

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

[7] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994.

[8] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3-4):107 – 113, 1999.

[9] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, Inc., 1994.

[10] Y. Dong-hong, X. Ke, and C. Yong. An improved wu-manber multiple patterns matching algorithm. In *Proceedings of the 25th IEEE International Performance, Computing and Communications Conference*, pages 675–680, 2006.

[11] S. Dori and G.M. Landau. Construction of aho corasick automaton in linear time for integer alphabets. *Information Processing Letters*, 98(2):66–72, 2006.

[12] GNU Grep. Webpage containing information about the gnu grep search utility. Website, 2012. http://www.gnu.org/software/grep/.

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

[14] P. Kalsi, H. Peltola, and J. Tarhio. Comparison of exact string matching algorithms for biological sequences. *Communications in Computer and Information Science*, 13:417–426, 2008.

[15] S. Kim and Y. Kim. A fast multiple string-pattern matching algorithm. *Proceedings of the 17th AoM/IAoM Inernational Conference on Computer Science*, pages 1–6, 1999.

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

[17] C.S. Kouzinopoulos and K.G. Margaritis. Experimental results on algorithms for multiple keyword matching. In *Proceedings of the IADIS International Conference on Informatics*, pages 129–133, 2010.

[18] C.S. Kouzinopoulos and K.G. Margaritis. Experimental Results On Multiple Pattern Matching Algorithms For Biological Sequences. In *Proceedings of the International Conference on Bioinformatics - Models, Methods and Algorithms*, pages 274–277, 2011.

[19] T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.

[20] P. Liu, Y. Liu, and J. Tan. A partition-based efficient algorithm for large scale multiple-strings matching. In *SPIRE*, pages 399–404. Springer, 2005.

[21] R. Muth and U. Manber. Approximate multiple string search. In *Combinatorial Pattern Matching*, pages 75–86. Springer, 1996.

[22] G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science*, 321(2-3):283–290, 2004.

[23] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. *Lecture Notes in Computer Science*, 1448:14–33, 1998.

[24] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical online search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[25] L. Salmela. *Improved Algorithms for String Searching Problems*. PhD thesis, Helsinki University of Technology, 2009.

[26] L. Salmela, J. Tarhio, and J. Kytöjoki. Multipattern string matching with q -grams. *Journal of Experimental Algorithmics*, 11:1–19, 2006.

[27] S.S. Sheik, S.K. Aggarwal, A. Poddar, B. Sathiyabhama, N. Balakrishna, and K. Sekar. Analysis of string-searching algorithms on biological sequence databases. *Current Science*, 89(2):368–374, 2005.

[28] T.F. Sheu, N.F. Huang, and H.P. Lee. Hierarchical multi-pattern matching algorithm for network content inspection. *Information Sciences*, 178:2880–2898, 2008.

[29] Snort. Webpage containing information on the snort intrusion prevention and detection system. Website, 2010. http://www.snort.org/.

[30] Streamline. The official webpage of the streamline project. Website, 2011. http://netstreamline.org/.

[31] B.W. Watson. *Taxonomies and toolkits of regular language algorithms*. PhD thesis, Eindhoven University of Technology, 1995.

[32] S. Wu and U. Manber. Agrep - A Fast Approximate Pattern-Matching Tool. *In Proceedings of USENIX Technical Conference*, pages 153–162, 1992.

[33] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. pages 1–11, 1994. Technical report TR-94-17.

[34] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li. Mdh: A high speed multi-phase dynamic hash string matching algorithm for large-scale pattern set. *Information and Communications Security*, 4861:201–215, 2007.