Linear Pattern Matching with Swaps for Short Patterns

Tomáš Flouri

Xhevi Qafmolla

Abstract—The Pattern Matching problem with swaps is a variation of the classical pattern matching problem. It consists of finding all the occurrences of a pattern \( P \) in a text \( T \), when an unrestricted number of disjoint local swaps is allowed. In this paper, we present a new, efficient method for the Swap Matching problem with short patterns. In particular, we present an algorithm constructing a non-deterministic finite automaton for a given pattern \( P \) which, when transformed to a deterministic finite automaton, serves as a pattern matcher running in time \( O(n) \), where \( n \) is the length of the input text \( T \).

I. INTRODUCTION

Finding all the occurrences of a given pattern in a text, i.e. the classical pattern matching, is one of the basic and most well-studied problems in computer science with many practical appliances in many areas such as computational biology, communications, data mining and multimedia. For example the Boyer-Moore algorithm is implemented in the emacs’ “s” command, or in UNIX’s “grep”. UNIX’s “diff” command uses the longest common subsequence algorithm [9] presented by Chvatal et al. since 1972.

The tremendous and continuous expansion of these fields, however, implied the need of a more generalized theoretical foundation of the pattern matching concept. Research has emerged in two directions: generalized matching and approximate matching. In generalized matching one seeks exact occurrences of the pattern in the text, but matching doesn’t mean equality. Instead, matching is done with “don’t cares”, less-than matching, or matching relation defined by a graph on the alphabet. In approximate matching one seeks to find approximate matches of the pattern. The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. This number is called the edit distance, also called the Levenshtein distance, between the string and the pattern.

In our paper we focus on the problem of Pattern Matching with Swaps, also known as the Swap Matching problem. In Swap Matching context, we say that the pattern \( P \) of length \( m \) matches the given text \( T \) of length \( n \) at location \( i \), when an unrestricted number of adjacent characters from the pattern can be swapped in order to become identical with a substring of \( T \) starting or ending at \( i \), given that all swaps are disjoint, i.e. no one character is involved in more than one swap. Both \( P \) and \( T \) are sequences of characters drawn from the same finite character set \( \Sigma \) of size \( \sigma \). To provide just a few applications of this definition, we could name mistyping in text pattern search, transmission noise adjusting in communications or finding of close mutations in biology. For example in gene mutation phenomenon we observe swaps in a disease called Spinal Muscular Atrophy [14]. Such cases serve as a convincing pointer to further theoretical study of swaps in computer science.

The Swap Matching problem was introduced in 1995, as one of the open problems in nonstandard string matching, by Muthukrishnan [16]. Amir et al. have since then done extensive research in this area producing many interesting results. They first provided an algorithm of \( O(nm^2 \log m \log \sigma) \) time complexity for an alphabet set of size two (see [2]). They also showed that alphabets of larger sizes could be reduced to the size of two having an \( O((\log^2 \sigma) \text{ time overhead}) \). Later in 1998, Amir et al. also studied some restrictive cases [5] for which they could obtain an algorithm of \( O(n \log^2 m) \) time complexity. Back in the year 2000, again Amir et al. tried to reduce the overhead of their 1998 algorithm, with the method of alphabet size reduction [3], introducing now an overhead of only \( O(\log \sigma) \). More recently, in another paper in 2003, Amir et al. found a new solution of \( O(n \log m \log \sigma) \) time, using overlap matching [4]. It is important to mention that all the above streams of research are based on the Fast Fourier Transformation (FFT).

The first efficient solution without using FFT was introduced in 2008 by Iliopoulos and Rahman [13]. Their approach consisted in introducing graph theory for initially modeling the problem and then, using bit parallelism, they developed an efficient algorithm running at \( O((n+m) \log m) \) time complexity. The constraint given was that the pattern size must be of a comparable size with the word size in the target machine, thus limiting their algorithm for small patterns.

More recently, in 2009, Cantone et al. continued in bit parallelism approach to introduce an algorithm named CROSS-SAMPLING [7]. The algorithm was characterized by a worst-case time complexity of \( O(nm) \) having a \( O(\sigma) \) space complexity for short patterns fitting in a few machine words. In the same year, Campanelli et al. presented an efficient way [6] for solving the Swap Matching problem with small patterns at \( O(nm^2) \) time complexity in general. Their algorithm was named BACKWARDS-CROSS-SAMPLING and inherited many properties of the original CROSS-SAMPLING algorithm, but was based on a right-to-left scan of the
text. Albeit having a worse time complexity, BACKWARDS-CROSS-SAMPLING proved to have better results in practice (for small patterns) than the other algorithms.

In this paper, we introduce an algorithm that runs in linear time. Our method uses finite automata (see [10], [15]) and is based on preprocessing the pattern, an operation we carry out only once at the beginning. Additionally, once the preprocessing is done, we can search in arbitrary many texts for the pattern without the need of preprocessing the pattern again each time.

The rest of this article is organized as follows. In section 2 we evoke some of the preliminary definitions needed for the purpose of our paper. In section 3 we present our algorithm along with the necessary proofs. In section 4 we demonstrate the implementation of our solution with an example. Section 5 serves as an overview of the time and space complexities. Finally, in section 5 we draw some conclusions and discuss further future work in our research.

II. PRELIMINARIES

An alphabet $\Sigma$ is a non-empty, finite set of symbols. A string $x$ over a given alphabet is a finite sequence of symbols. $\Sigma^*$ denotes the set of all strings over alphabet $\Sigma$ including the empty string, denoted by $\varepsilon$. A string of length $m \geq 0$ can be represented as a finite array $x[1 \ldots m]$. The length of the string can also be presented as $|x| = m$. A string $y$ is a substring of $x$ if and only if $x = yuv$, where $x, y, u, v \in \Sigma^*$. A substring $y$ of a string $x$ can be represented as a finite array $x[i \ldots j]$, $i$ and $j$ denoting the starting and the ending position of $y$ in $x$, respectively. We define the concatenation operation on the set of strings in the usual way: if $x$ and $y$ are strings over alphabet $\Sigma$, then the concatenation of these strings is $xy$. In particular, for $m = 0$ we obtain the empty string, denoted by $\varepsilon$. For any set $A$ we use $P(A)$ to denote the set of all subsets of $A$. $P(A)$ is called the powerset of $A$. A function $P : X \rightarrow \{\text{true, false}\}$ is called a predicate on $X$.

Since our algorithms are based on finite automata, we give brief definitions to related concepts below. A non-deterministic finite automaton $M$ is a quintuple $(Q, \Sigma, \delta, I, F)$, where: $Q$ is a finite set of states, $\Sigma$ is an input alphabet, $\delta$ is a mapping $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$ called a state transition function, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. A deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is a special case of non-deterministic finite automaton such that the transition mapping is a function $\delta : Q \times \Sigma \rightarrow Q$ and there is only one initial state $q_0 \in Q$.

The extended transition function $\delta^*$ of a non-deterministic finite automaton is defined inductively as follows:

1) $\delta^*(q, \varepsilon) = \{q\}$,
2) $\delta^*(q, us) = \bigcup_{p \in \delta^*(q, u)} \delta(p, s)$.

The left language of state $q$ of a non-deterministic automaton $M = (Q, \Sigma, \delta, q_0, F)$ is defined as $L_M(q) = \{u \mid q \in \delta^*(q_0, u)\}$. The language accepted by a non-deterministic finite automaton $M = (Q, \Sigma, \delta, I, F)$ is defined as $L_M = \{u \mid p \in \delta^*(q_0, u), q \in I \land p \in F\}$. A configuration of a non-deterministic finite automaton is the relation $\Gamma_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$. For example, if $p \in \delta(q, a)$ then $(q, aw) \in \Gamma_M (p, w)$, for arbitrary $w \in \Sigma^*$.

The extended transition function $\delta^*$ of a deterministic finite automaton is defined inductively as follows:

1) $\delta^*(q, \varepsilon) = q$,
2) $\delta^*(q, us) = \delta(\delta^*(q, u), s)$.

The language accepted by a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is defined as $\mathcal{L}_M = \{u \mid p \in \delta^*(q_0, u) \land p \in F\}$.

Finite automata $M_1$ and $M_2$ are said to be equivalent if they accept the same language, that is $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

Subset construction is a process transforming a non-deterministic finite automaton into an equivalent deterministic finite automaton. If $M = (Q, \Sigma, \delta, I, F)$ is a non-deterministic finite automaton and $M'$ is the deterministic finite automaton obtained by subset construction from $M$, then $M'$ is of the form $M' = (\mathcal{P}(Q), \Sigma, \delta', I, F')$ and it holds:

1) $\delta'(B, s) = \bigcup_{q \in B} \delta(q, s), \forall B \in \mathcal{P}(Q)$,
2) $F' = \{B \mid B \in \mathcal{P}(Q) \land B \cap F \neq \emptyset\}$.

Transition diagram of a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is a directed graph such that:

- for each state $q \in Q$, there exists exactly one node labeled by $q$ drawn as circle or oval,
- the graph has an arc from node $q$ to node $p$ labeled by $s$ if and only if $M$ has a transition labeled by $s$ leading from state $q$ to state $p$,
- the initial state has an-intransition with no label,
- final states are drawn as two concentric circles or ovals.

The transition table of a finite automaton $M = (Q, \Sigma, \delta, I, F)$ is a table consisting of $|Q|+1$ rows and $|\Sigma|+1$ columns with the first row and first column indexed by 0. A cell of the table is indicated by the pair $(i, j)$ where $i$ denotes the row and $j$ the column. Cells $(0, 1)$ up to $(0, |\Sigma|)$ contain each a unique $x \in \Sigma$. Cells $(1, 0)$ up to $(|Q|, 0)$ contain a unique $q \in Q$. The content of cells $(i, j)$ where $i \neq 0$ and $j \neq 0$ is the mapping $\delta([i, 0], [0, j])$.

A swap permutation for a string $x$, where $|x| = m$, is a permutation $\pi : \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m-1\}$ such that:

1) if $\pi(i) = j$ then $\pi(j) = i$ (characters are swapped),
2) for all $i$, $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters are swapped),
3) if $\pi(i) \neq i$ then $x[\pi(i)] \neq x[i]$ (identical characters are not swapped).

For a given string $x$ and a swap permutation $\pi$ we denote $x[\pi] = x[\pi(0)], x[\pi(1)], \ldots, x[\pi(m-1)]$ the swapped version of $x$.

For a given string $T$ representing the text and string $P$ representing the pattern, where $|T| = n$ and $|P| = m$, we say that $P$ has a swapped match at location $i$, if there exists a swapped version $P'$ of $P$, such that $P'$ has an exact match with $T$ starting at location $i$, i.e. $\pi(P) = T[i-m+1 \ldots i]$. 


III. ALGORITHM

In this section, we present an algorithm for solving the swap matching problem. The algorithm constructs a non-deterministic finite automaton which can be transformed to an equivalent deterministic finite automaton serving as a pattern matcher. We first present an algorithm which, given a pattern $P$, constructs a deterministic automaton accepting the language $L_1$.

We then extend the first algorithm so that given a pattern $P$, constructs a so-called searching non-deterministic automaton which accepts the language $L = \{ x. \pi(P) \}$ for all swap permutations $\pi$.

**Lemma 1:** Given a pattern $P$, Algorithm 1 constructs a deterministic finite automaton $M = (Q, \Sigma, \delta, 0, F)$ accepting language $L = \{ \pi(P) \}$ for all swap permutations $\pi$.

**Proof:** By strong induction. Let $R(n)$ be a predicate defined over all integers $n$. Predicate $R(n)$ is true, if the automaton $M = (Q, \Sigma, \delta, 0, F)$ constructed by Algorithm 1 accepts the language $L = \{ \pi(P[1 \ldots n]) \}$ for all swap permutations $\pi$. We define the base case and the inductive step in the following manner:

1. Base case: $R(2)$ is true.
2. Inductive step: $R(2), \ldots, R(n) \Rightarrow R(n + 1)$

Given an alphabet $\Sigma = \{x_1, x_2\}$ and a string $x = x_1x_2$, the two possible swap versions of the string are $x_1x_2$ and $x_2x_1$. The automaton $M$ constructed by Algorithm 1 can have the following configurations:

$$(0, x_1x_2w) \rightarrow_M (1, x_2w) \rightarrow_M (2, w)$$

where $w \in \Sigma^*$. Thus, the language accepted by automaton $M$ is $L = \{x_1x_2, x_2x_1\}$ and the base case holds.

Suppose we have a pattern $P$, where $|P| = n + 1$. By definition, symbol $P[n+1]$ can only be swapped with the adjacent symbol $P[n]$. Thus, the set of all swapped versions of $P$ is $\{ \pi(P[1 \ldots n]), P[n+1] \}$, where $\pi$ is a permutation of $\{0, 1, \ldots, n\}$ and $\pi(n) = n + 1$. We will prove this statement.

We now construct a pattern $\pi(P[i \ldots n + 1])$ such that it makes the automaton $M_{n+1} = \{Q_n, \Sigma, \delta_n, 0, F_n\}$ and $M_{n+1} = \{Q_n, \Sigma, \delta_n, 0, F_n\}$ such that $M_{n+1}$ accepts the language $L_{M_{n+1}} = \{ \pi(P[i \ldots n + 1]) \}$, which is a subset of $L$.

**Algorithm 1:** Construction of a deterministic finite automaton accepting language $L = \{ \pi(x) \}$

**Input:** $x = x_1x_2 \ldots x_n$ - input string over alphabet $\Sigma$ representing the pattern

**Output:** $M$ - deterministic finite automaton with swaps, accepting language $L = \{ \pi(x) \}$ for all swap permutations $\pi$

**Algorithm 2:** Given a pattern $P$ of length $n$, Algorithm 2 constructs a non-deterministic finite automaton $M = (Q, \Sigma, \delta, 0, \{ F \})$ accepting language $L = \{ w. \pi(P) \}$ for all $w \in \Sigma^*$.

**Proof:** We only provide a sketch of the proof: Since $0 \in \delta(0, x)$ for all $x \in \Sigma$, it holds that $0 \in \delta^*(0, w)$ for all $w \in \Sigma^*$. In other words, $L_M = \{ w \mid w \in \Sigma^* \}$. Using Lemma 1 we can prove that $L_M = \{ \pi(P[i \ldots n]) \}$ and $L_M = \{ \pi(P[i \ldots n + 1]) \}$.

**Theorem 3:** Given a pattern $P$, Algorithm 1 constructs a deterministic automaton $M_1 = (Q, \Sigma, \delta, I, F)$, having at most $2|P|$ states, 1 initial state $(0)$, 1 final state $(F = \{ n \})$ and $|P| - 2$ transitions. Automaton $M_1$ accepts language $L_{M_1} = \{ \pi(P) \}$.

**Theorem 4:** Given a pattern $P$, Algorithm 2 constructs a non-deterministic automaton $M_2 = (Q, \Sigma, \delta, I, F)$, having
Algorithm 2: Construction of a searching non-deterministic finite automaton accepting language $L = \{ w.\pi(x) \}$

**Input:** $x = x_1x_2 \ldots x_m$ - input string over alphabet $\Sigma$ representing the pattern

**Output:** $M$ - searching non-deterministic finite automaton with swaps

1. $Q \leftarrow \{0\}$
2. $I \leftarrow \{0\}$
3. $\delta \leftarrow \emptyset$
4. $F \leftarrow \{m\}$
5. for $i \leftarrow 1$ to $m$ do
   $Q \leftarrow Q \cup \{i,i'\}$
6. for $i \leftarrow 1$ to $m-1$ do
51. $\delta(i-1,x[i]) = \{i\}$
52. if $x[i] \neq x[i+1]$ then
53. $\delta(i-1,x[i+1]) = \{i'\}$
54. $\delta(i',x[i]) = \{i+1\}$
7. end
8. end
9. $\delta(m-1,x[m]) = \{m\}$
10. for each $x \in \Sigma$ do
11. $\delta(0,x) \leftarrow \delta(0,x) \cup \{0\}$
12. $M \leftarrow (Q, \Sigma, \delta, I, F)$

at most $2|P|$ states, 1 initial state ($I = \{0\}$), 1 final state ($F = \{n\}$) and $3|P| - 2 + |\Sigma|$ transitions. Automaton $M_2$ accepts language $L_{M_2} = \{ x.\pi(P) \}$ for all $x \in \Sigma^*$.

We present Theorem 3 and 4 without proof, as the results can be trivially calculated from Fig. 1–3 and Lemma 1–2.

IV. EXAMPLE

In this section we demonstrate Algorithm 1 and 2 with a short example.

The transition diagram of the automaton $M_1$ created by Algorithm 1 given the pattern $P = abed$ is depicted in Fig. 4. Automaton $M_1$ accepts the language $L_{M_1} = \{ abcd, abd, acbd, bacd, bade \}$. In this case, $M$ is a deterministic finite automaton but in general, the automaton obtained by Algorithm 1 is non-deterministic (when $P[i + 1] = P[i]$ for $1 \leq i < |P|$).

To transform $M_1$ obtained from Algorithm 1 to a searching automaton we modify the transition function $\delta$ to $\delta(q,x) = \delta(q,x) \cup \{q\}$ for all $q \in I$ and all $x \in \Sigma$. The whole process is described in Algorithm 2. The automaton $M_2$, constructed by Algorithm 2, accepts the language $L_{M_2} = \{ x.\pi(abed) \}$ for all swap permutations $\pi$ and all $x \in \Sigma^*$. Again, for a given pattern $P = abcd$, automaton $M_2$ created by Algorithm 2 is depicted in Fig. 5. $M_2$ accepts the language $L_{M_2} = \{ x.abed, x.abde, x.acbd, x.bacd, x.bade \}$ for all $x \in \Sigma^*$.

From the theory of finite automata it holds that, for every non-deterministic finite automaton exists an equivalent deterministic finite automaton [17], [12]. The transformation (non-deterministic to deterministic) can be done using the method of subset construction.

We obtain the deterministic finite automaton $M = \{Q, \{a,b,c,d\}, \delta, \{0\}, \{0,4\} \}$, with its states and transition function presented by the transition table in Table I. The transition diagram of $M$ is depicted in Fig. 6.

The preprocessing phase is now complete and we can search for swap matches of pattern $P$ in arbitrary text. As an example, suppose a string $x = aabeddbadca$. The trace of the deterministic finite automaton $M$ is:

\[
\begin{align*}
&\{\{0\}, aabeddbadca\} &\vdash_M \{\{0,1\}, abeddbadca\} \\
&\vdash_M \{\{0,1\}, beddbadca\} \\
&\vdash_M \{\{0,1\}, cddbadca\} \\
&\vdash_M \{\{0,3\}, ddbadca\} \\
&\vdash_M \{\{0,4\}, adca\} &\text{Match} \\
&\vdash_M \{\{0,3\}, ca\} \\
&\vdash_M \{\{0,4\}, a\} &\text{Match} \\
&\vdash_M \{\{0,1\}, \varepsilon\} 
\end{align*}
\]

The trace locates 2 matches of the swap versions of pattern $P$. The first occurrence ends at position 5 of pattern $P$ (substring $abcd$) and the second ends at position 10 (substring $bade$). The occurrences are detected (accepted) by final state.

We also note that each symbol of the input text $x$ was read only once (linear search phase).

V. COMPLEXITIES

In this section, we present the resulting space and time complexities of our algorithm. But first, we present a proof on the number of all possible swapped versions of a
TABLE I
TRANSITION TABLE OF DETERMINISTIC AUTOMATON M ACCEPTING LANGUAGE $L_M = \{ \pi(abcd) \}$ FOR ALL SWAP PERMUTATIONS $\pi$

<table>
<thead>
<tr>
<th>$\pi$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,1,2)</td>
<td>(0,1,2)</td>
<td>(0,1,2)</td>
<td>(0,1,2)</td>
<td>(0,1,2)</td>
</tr>
<tr>
<td>(0,2)</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
</tr>
<tr>
<td>(0,1,2)</td>
<td>(0,1)</td>
<td>(0,1,2)</td>
<td>(0,1)</td>
<td>(0,1,2)</td>
</tr>
<tr>
<td>(0,3)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,2,3)</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
</tr>
<tr>
<td>(0,4)</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
</tr>
</tbody>
</table>

Fig. 4. Finite automaton $M$ accepting language $L_M = \{ \pi(abcd) \}$ for all swap permutations $\pi$.

Fig. 5. Finite automaton $M$ accepting language $L_M = \{ x.\pi(abcd) \}$ for all swap permutations $\pi$ and $x \in \hat{\Sigma}$.

pattern $P$ which will aid us on proving the space complexity.

Lemma 5: Given a string $x$ of size $n \geq 2$, the number of distinct swapped versions of $x$ is exactly $\frac{(1+\sqrt{5})^{n+1}-(1-\sqrt{5})^{n+1}}{2^{n+1}\sqrt{5}}$.

Proof: Suppose we have a pattern $P[1\ldots n+1]$ and 3 automata, $M_{n+1}$, $M_n$ and $M_{n-1}$, constructed by Algorithm 1. Automaton $M_{n+1}$ is constructed over pattern $P[1\ldots n+1]$, $M_n$ over pattern $P[1\ldots n]$ and $M_{n-1}$ over $P[1\ldots n-1]$. From the proof of Lemma 1 it holds that the languages accepted by $M_{n+1}$, $M_n$, $M_{n-1}$ are $L_{M_{n+1}} = \{ \pi(P[1\ldots n],[P[n+1],P[1\ldots n]], P[n+1],P[n]\}$, $L_{M_n} = \{ \pi(P[1\ldots n]) \}$ and $L_{M_{n-1}} = \{ \pi(P[1\ldots n-1]) \}$ respectively.

This means that $|L_{M_{n+1}}| = |L_{M_n}| + |L_{M_{n-1}}|$, which forms a recurrent formula for generating a Fibonacci sequence (see [8]). The $n$-th element of a Fibonacci sequence can be calculated using Binet’s formula [19], which is $F(n) = \frac{(1+\sqrt{5})^n-(1-\sqrt{5})^n}{2^n\sqrt{5}}$.

For $n = 2, 3, 4$, Binet’s formula yields the following results: $F(2) = 1$, $F(3) = 2$ and $F(4) = 3$. The number of swapped versions of a pattern $P$ of size 2 and 3 is 2 and 3 respectively. The sequence of the number of swapped versions of patterns of size 2, 3, ..., $n$ is the Fibonacci sequence shifted by one element and thus the recurrent formula $F_s(n) = F(n+1)$.

Theorem 6: Given a pattern $P$ of size $m$, Algorithm 1 constructs a non-deterministic finite automaton $M$, accepting language $L_M = \{ \pi(P) \}$ for all swap permutations $\pi$, in time $O(m)$.

Theorem 7: Given a pattern $P$ of size $m$ and alphabet $\Sigma$, Algorithm 2 constructs a non-deterministic finite automaton $M$, accepting language $L_M = \{ w.\pi(P) \}$ for all $w \in \Sigma^*$ and all swap permutations $\pi$, in time $O(m)$.

Theorems 6–8 are trivial to prove. Algorithms 1–2 construct $2m$ states and define at most two transitions for each state by reading the pattern from left to right.

Theorem 8: The space complexity of the deterministic automaton $M_d$, obtained by subset construction on automaton $M_{nd}$ constructed by Algorithm 2 over pattern $P$ of size $m$, is $O(2^m)$.

Proof: The language accepted by automaton $M_{nd}$ consists of $k = \frac{(1+\sqrt{5})^m-(1-\sqrt{5})^{m+1}}{2^{m+1}\sqrt{5}}$ strings (Lemma 1 and 5). Automaton $M_{nd}$ accepts the same language as an Aho-Corasick finite automaton (see [1], [18]) constructed over a finite set of strings $S = \{ p_1, p_2, \ldots, p_k \}$, where $p_i, 1 \leq i \leq k$, are all possible, distinct swapped versions of $P$. The space complexity of the deterministic Aho-Corasick finite automaton is $\Theta(\alpha\beta)$ (see [18]), where $\alpha$ is the size of the alphabet and $\beta$ the sum of lengths of all strings in set $S$. In our case, $\beta = km$, which indicates exponential size.

Theorem 9: The searching phase of the deterministic automaton $M_d$, obtained by subset construction on automaton $M_{nd}$ constructed by Algorithm 2 over pattern $P$ of size $n$, is $O(n)$.

Proof: This is a property of deterministic automata serving as pattern matchers. The input text is read from left to right, symbol by symbol. For each symbol $a$, a transition from some state $q_1$ to a state $q_2$ is taken, according to the transition function $\delta(q_1, a) = q_2$. The automaton detects occurrences of swapped versions of the pattern inside the input text by a transition to the final state.

VI. CONCLUSIONS AND FUTURE WORKS
In this paper we have presented a new, efficient algorithm for the Swap Matching problem on short patterns with a searching phase running in linear time. The algorithm constructs a non-deterministic finite automaton which can be transformed to a deterministic one, serving as a pattern matcher. Our method is based on preprocessing the pattern, an operation carried out only once at the beginning.

The main advantage of the method is that the preprocessing is done only once at the beginning and the constructed automaton can be used as a pattern-matcher for arbitrary many texts without the need of preprocessing the pattern again. The drawback of this method is the high (exponential)
space complexity, which limits this method only for short patterns.

REFERENCES


