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 $\mathcal{O}(n)$, 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. Primitive operations can be insertion, deletion, substitution and transposition, or swapping.

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 Σ of size σ. 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 excessive research in this area producing many interesting results. They first provided an algorithm of $\mathcal{O}(nm^{\frac{1}{3}}\log m\log\sigma)$ 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 $\mathcal{O}(\log^2\sigma)$ time overhead. Later in 1998, Amir et al. also studied some restrictive cases [5] for which they could obtain an algorithm of $\mathcal{O}(n\log^2 m)$ 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 $\mathcal{O}(\log\sigma)$. More recently, in another paper in 2003, Amir et al. found a new solution of $\mathcal{O}(n\log m\log\sigma)$ 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 $\mathcal{O}((n+m)\log m)$ 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 $\mathcal{O}(nm)$ having a $\mathcal{O}(\sigma)$ 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 $\mathcal{O}(nm^2)$ 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

This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSM 6840770014, and by the Czech Science Foundation as project No. 201/09/0807.

T. Flouri and X. Qafmolla are with the Faculty of Electrical Engineering, Department of Computer Science and Engineering, Czech Technical University in Prague, Czech Republic {flourt1,qafmox1}@fel.cvut.cz

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* Σ is a non-empty, finite set of symbols. A *string* x over a given alphabet is a finite sequence of symbols. Σ^* denotes the set of all strings over alphabet Σ including the *empty* string, denoted by ε. A string of length $m \geq 0$ can be represented as a finite array $x[1 \dots m]$. The length of the string can also be presented as $|x| = m$. A string y is a *substring* of x if and only if $x = uyv$, where $x, y, u, v \in \Sigma^*$. A substring y of a string x can be represented as a finite array $x[i \dots j]$, 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 Σ, then the concatenation of these strings is xy. In particular, for $m = 0$ we obtain the empty string, denoted by ε. For any set A we use $\mathcal{P}(A)$ to denote the set of all subsets of A. $\mathcal{P}(A)$ is called the *powerset* of A. A function $P : X \to \{\textbf{true}, \textbf{false}\}$ 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, Σ is an input alphabet, δ is a mapping $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$ 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 $\delta : Q \times \Sigma \mapsto Q$ and there is only one initial state $q_0 \in Q$.

The *extended transition function* δ^* 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 $\overleftarrow{\mathcal{L}}_M(q) = \{u \mid q \in \delta^*(q_0, u)\}$. The language accepted by a non-deterministic finite automaton $M = (Q, \Sigma, \delta, I, F)$ is defined as $\mathcal{L}_M = \{\ u \mid p \in \delta^*(q, u),\ q \in I \ \wedge \ p \in F\ \}$. A *configuration* of a non-deterministic finite automaton is the relation $\vdash_M \subset (Q \times \Sigma^*) \times (Q \times \Sigma^*)$. For example, if $p \in \delta(q, a)$ then $(q, aw) \vdash_M (p, w)$, for arbitrary $w \in \Sigma^*$.

The extended transition function δ^* 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 $\mathcal{L}_M = \{\ u \mid p \in \delta^*(q_0, u) \ \wedge \ p \in F\ \}$.

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) $F' = \{\ B \mid B \in \mathcal{P}(Q) \ \wedge \ B \cap F \neq \emptyset\ \}$.

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 source,
- 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 $(0, 1)$ upto $(0, |\Sigma|)$ contain each a unique $x \in \Sigma$. Cells $(1, 0)$ upto $(|Q|, 0)$ 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 $\pi : \{0, \dots, m-1\} \mapsto \{0, \dots, m-1\}$ 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 π we denote $\pi(x) = x[\pi(0)].x[\pi(1)].\cdots.x[\pi(m-1)]$ 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. $\pi(P) = T[i - m + 1 \dots i]$.

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 $\mathcal{L} = \{ \pi(P) \}$ for all swap permutations π. We then extend the first algorithm so that given a pattern P, constructs a so-called *searching non-deterministic automaton* which accepts the language $\mathcal{L} = \{ x.\pi(P) \}$ for all $x \in \Sigma$, where Σ represents the alphabet over which pattern P was constructed.

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

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 $\mathcal{L} = \{ \pi(P[1 \dots n) \}$ for all swap permutations π. We define the base case and the inductive step in the following manner:

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

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:

$$(0, x_1x_2w) \vdash_M (1, x_2w) \vdash_M (2, w)$$
$$(0, x_2x_1w) \vdash_M (1', x_1w) \vdash_M (2, w)$$

where $w \in \Sigma^*$. Thus, the language accepted by automaton M is $\mathcal{L} = \{x_1x_2, x_2x_1\}$ 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 $\{ \pi(P[1 \dots n]).P[n+1] \} \cup \{ \pi(P[1 \dots n-1]).P[n+1].P[n] \}$, and we will prove this statement.

Suppose we have three automata, $M_{n-1} = (Q_{n-1}, \Sigma, \delta_{n-1}, 0, F_{n-1})$, $M_n = (Q_n, \Sigma, \delta_n, 0, F_n)$ and $M_{n+1} = (Q_{n+1}, \Sigma, \delta_{n+1}, 0, F_{n+1})$ constructed over the patterns $P[1 \dots n-1]$, $P[1 \dots n]$ and $P[1 \dots n+1]$ respectively, where $F_{n-1} = \{n-1\}$, $F_n = \{n\}$ and $F_{n+1} = \{n+1\}$. Their transition diagrams are depicted in Fig. 1–3. According to the assumption in the inductive step, automata M_{n-1} and M_n accept languages $L_{M_{n-1}} = \{\pi(P[1 \dots n-1]\}$ and $L_{M_n} = \{\pi(P[1 \dots n]\}$ respectively, for all swap permutations π. Trivially from Algorithm 1 it holds that the transition function of an automaton constructed over pattern $P[1 \dots n+1]$ is a superset of the transition function of an automaton constructed over pattern $P[1 \dots n]$. Specifically, it holds that $\delta_{n-1} \subseteq \delta_n \subseteq \delta_{n+1}$. Moreover, from Algorithm 1 and Fig. 2–3, we can deduce that $\delta_n(q, x) = \delta_{n+1}(q, x)$ for all $q \in Q_n \setminus \{n-1, n\}$ and all $x \in \Sigma$ and thus $\overleftarrow{\mathcal{L}}_{M_n}(q) = \overleftarrow{\mathcal{L}}_{M_{n+1}}(q)$ for all $q \in Q_n$. In a similar way we deduce that $\overleftarrow{\mathcal{L}}_{M_{n-1}}(q) = \overleftarrow{\mathcal{L}}_{M_{n+1}}(q)$ for all $q \in Q_{n-1}$.

In addition, automaton M_{n+1} has the following configurations over automaton M_n:

(1) $(n, x_{n+1}w) \vdash_{M_{n+1}} (n+1, w)$
(2) $(n-1, x_{n+1}x_nw) \vdash_{M_{n+1}} (n', x_nw) \vdash_{M_{n+1}} (n+1, w)$

From (1),(2), $\overleftarrow{\mathcal{L}}_{M_n}(n) = \overleftarrow{\mathcal{L}}_{M_{n+1}}(n) = \mathcal{L}_{M_n}$ and $\overleftarrow{\mathcal{L}}_{M_{n-1}}(n-1) = \overleftarrow{\mathcal{L}}_{M_{n+1}}(n-1) = \mathcal{L}_{M_{n-1}}$, it holds that $L_{M_{n+1}} = \{ x.P[n+1] \mid \forall x \in \mathcal{L}_{M_n} \} \cup \{ x.P[n+1].P[n] \mid \forall x \in \mathcal{L}_{M_{n-1}} \}$. From the inductive step assumption we have $\mathcal{L}_{M_{n-1}} = \{ \pi(P[1 \dots n-1]) \}$ and $\mathcal{L}_{M_n} = \{ \pi(P[1 \dots n]) \}$ and thus $L_{M_{n+1}} = \{ \pi[P[1 \dots n].P[n+1], \pi(P[1 \dots n-1]).P[n+1].P[n] \}$.

■

Lemma 2: Given a pattern P of length m, Algorithm 2 constructs a non-deterministic finite automaton $M = (Q, \Sigma, \delta, \{0\}, \{m\})$ accepting language $\mathcal{L} = \{ w.\pi(P) \}$ 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, $\overleftarrow{\mathcal{L}}_M = \{ w \mid w \in \Sigma^* \}$. Using Lemma 1 we can prove that $\mathcal{L}_M = \{ w.\pi(P) \}$ for all $w \in \Sigma^*$. ■

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

input : $x = x_1x_2 \dots x_m$ - input string over alphabet Σ representing the pattern
output : M - deterministic finite automaton with swaps, accepting language $\mathcal{L} = \{\pi(x)\}$ for all swap permutations π

```
Q ← {0}
δ ← ∅
F ← {m}
for i ← 1 to m do Q ← Q ∪ {i, i'}
for i ← 1 to m − 1 do
    δ(i − 1, x[i]) = {i}
    if x[i] ≠ x[i + 1] then
        δ(i − 1, x[i + 1]) = {i'}
        δ(i', x[i]) = {i + 1}
    end
end
δ(m − 1, x[m]) = {m}
M ← (Q, Σ, δ, 0, F)
```

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 $3|P| - 2$ transitions. Automaton M_1 accepts language $\mathcal{L}_{M_1} = \{ \pi(P) \}$.

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 $\mathcal{L} = \{ w.\pi(x) \}$

input : $x = x_1 x_2 \dots x_m$ - input string over alphabet Σ representing the pattern
output : M - searching non-deterministic finite automaton with swaps

$Q \leftarrow \{0\}$
$I \leftarrow \{0\}$
$\delta \leftarrow \emptyset$
$F \leftarrow \{m\}$
for $i \leftarrow 1$ **to** m **do** $Q \leftarrow Q \cup \{i, i'\}$
for $i \leftarrow 1$ **to** $m-1$ **do**
    $\delta(i-1, x[i]) = \{i\}$
    if $x[i] \neq x[i+1]$ **then**
        $\delta(i-1, x[i+1]) = \{i'\}$
        $\delta(i', x[i]) = \{i+1\}$
    end
end
$\delta(m-1, x[m]) = \{m\}$
for each $x \in \Sigma$ **do** $\delta(0, x) \leftarrow \delta(0, x) \cup \{0\}$
$M \leftarrow (Q, \Sigma, \delta, I, F)$

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 $\mathcal{L}_{M_2} = \{ x.\pi(P) \}$ 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.

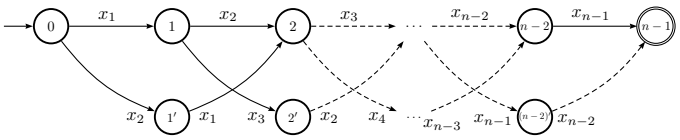

Fig. 1. Transition diagram of automaton M_{n-1} from Lemma 1

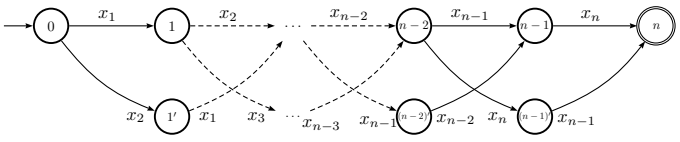

Fig. 2. Transition diagram of automaton M_n from Lemma 1

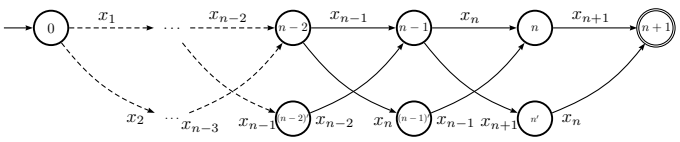

Fig. 3. Transition diagram of automaton M_{n+1} from Lemma 1

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 $P = abcd$ is depicted in Fig. 4. Automaton M_1 accepts the language $\mathcal{L}_{M_1} = \{ abcd, abdc, acbd, bacd, badc \}$. 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 δ 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 $\mathcal{L}_{M_2} = \{ x.\pi(abcd) \}$ for all swap permutations π 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 $\mathcal{L}_{M_2} = \{ x.abcd, x.abdc, x.acbd, x.bacd, x.badc \}$ 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 $x = aabcddbadca$. The trace of the deterministic finite automaton M is:

$$
\begin{array}{lll}
(\{0\}, aabcddbadca) & \vdash_M & (\{0, 1\}, abcddbadca) \\
& \vdash_M & (\{0, 1\}, bcddbadca) \\
& \vdash_M & (\{0, 1', 2\}, cddbadca) \\
& \vdash_M & (\{0, 3\}, ddbadca) \\
& \vdash_M & (\{0, 4\}, dbadca) \quad \text{Match} \\
& \vdash_M & (\{0\}, badca) \\
& \vdash_M & (\{0, 1'\}, adca) \\
& \vdash_M & (\{0, 1, 2\}, dca) \\
& \vdash_M & (\{0, 3'\}, ca) \\
& \vdash_M & (\{0, 4\}, a) \quad \text{Match} \\
& \vdash_M & (\{0, 1\}, \varepsilon)
\end{array}
$$

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 $badc$). 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 $\mathcal{L}_M = \{\ \pi(abcd)\ \}$ FOR ALL SWAP PERMUTATIONS π

	a	b	c	d
$\{0\}$	$\{0,1\}$	$\{0,1'\}$	$\{0\}$	$\{0\}$
$\{0,1\}$	$\{0,1\}$	$\{0,1',2\}$	$\{0,2'\}$	$\{0\}$
$\{0,1'\}$	$\{0,1,2\}$	$\{0,1'\}$	$\{0\}$	$\{0\}$
$\{0,1',2\}$	$\{0,1,2\}$	$\{0,1'\}$	$\{0,3\}$	$\{0,3'\}$
$\{0,2'\}$	$\{0,1\}$	$\{0,1',3\}$	$\{0\}$	$\{0\}$
$\{0,1,2\}$	$\{0,1\}$	$\{0,1',2\}$	$\{0,2',3\}$	$\{0,3'\}$
$\{0,3\}$	$\{0,1\}$	$\{0,1'\}$	$\{0\}$	$\{0,4\}$
$\{0,3'\}$	$\{0,1\}$	$\{0,1'\}$	$\{0,4\}$	$\{0\}$
$\{0,1',3\}$	$\{0,1,2\}$	$\{0,1'\}$	$\{0\}$	$\{0,4\}$
$\{0,2',3\}$	$\{0,1\}$	$\{0,1',3\}$	$\{0\}$	$\{0,4\}$
$\{0,4\}$	$\{0,1\}$	$\{0,1'\}$	$\{0\}$	$\{0\}$

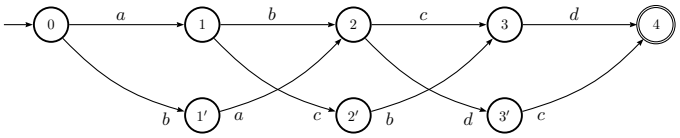

Fig. 4. Finite automaton M accepting language $\mathcal{L}_M = \{\ \pi(abcd)\ \}$ for all swap permutations π.

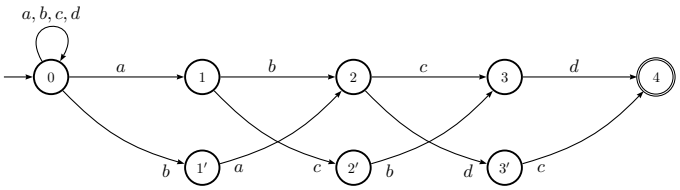

Fig. 5. Finite automaton M accepting language $\mathcal{L}_M = \{\ x.\pi(abcd)\ \}$ for all swap permutations π and all $x \in \Sigma^*$.

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 $P[1 \dots n+1]$ and 3 automata, M_{n+1}, M_n and M_{n-1}, constructed by Algorithm 1. Automaton M_{n+1} is constructed over pattern $P[1 \dots n+1]$, M_n over pattern $P[1 \dots n]$ and M_{n-1} over $P[1 \dots n-1]$. From the proof of Lemma 1 it holds that the languages accepted by M_{n+1}, M_n, M_{n-1} are $L_{M_{n+1}} = \{\ \pi[P[1 \dots n].P[n+1], \pi(P[1 \dots n-1]).P[n+1].P[n]\ \}$, $L_{M_n} = \{\ \pi[P[1 \dots n]\ \}$ and $L_{M_{n-1}} = \{\pi[P[1 \dots n-1]\ \}$, 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 $2, 3, \dots n$ is the Fibonacci sequence shifted by one element and thus the recurrent formula F_s, for finding the swapped versions, is $F_s(n) = F(n+1)$.

■

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 π, in time $\mathcal{O}(m)$.

Theorem 7: Given a pattern P of size m and alphabet Σ, 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 π, in time $\mathcal{O}(m)$.

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 $\mathcal{O}(2^m)$.

Proof: The language accepted by automaton M_{nd} consists of $k = \frac{(1+\sqrt{5})^{m+1}-(1-\sqrt{5})^{m+1}}{2^{m+1}\sqrt{5}}$ 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 $S = \{p_1, p_2, \dots, p_k\}$, 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 α is the size of the alphabet and β 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 $\mathcal{O}(n)$.

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)

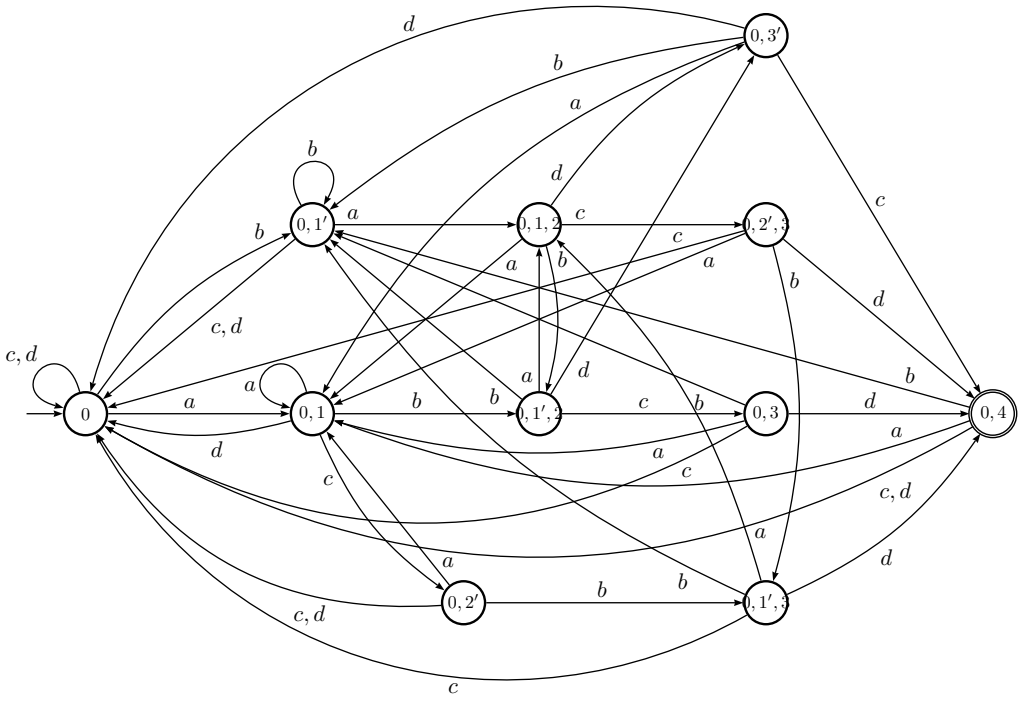

Fig. 6. Transition diagram of deterministic finite automaton M accepting language $L_M = \{\ abcd, abdc, acbd, bacd, badc\ \}$.

space complexity, which limits this method only for short patterns.

REFERENCES

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

[2] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein and N. Lewenstein. Pattern Matching with Swaps. *IEEE Symposium on Foundation of Computer Science*, pages 144–153, 1997

[3] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein and N. Lewenstein. Pattern Matching with Swaps. *Journal of algorithms*, volume 37 (2), pages 247–266, 2000

[4] A. Amir, R. Cole, R. Harihan, M. Lewenstein, and E. Porat. Overlap Matching. *Information and Computation*, volume 181 (1), pages 57–74, 2003

[5] A. Amir, G. M. Landau, M. Lewenstein and N. Lewenstein. Efficient special cases of pattern matching with swaps. *Information Processing Letters*, volume 68 (3), pages 125–132, 1998

[6] M. Campanelli, D. Cantone and S. Faro. A New Algorithm for Efficient Pattern Matching with Swaps. *Proceedings of the 20th International Workshop on Combinatorial Algorithms*, 2009

[7] D. Cantone and S. Faro. Pattern matching with swaps for short patterns in linear time. *Software Seminar Conference 2009*, volume 5404 of Lecture Notes in Computer Science, pages 255–266, 2009

[8] Chandra, Pravin and Weisstein, Eric W. Fibonacci Number. *From MathWorld–A Wolfram Web Resource*. http://mathworld.wolfram.com/FibonacciNumber.html

[9] V. Chvatal, D. A. Klarner and D. E. Knuth. Selected combinatorial research problems. *Technical Report STAN-CS-72-292*, Stanford University, 1972

[10] Crochemore, M., Hancart, Ch. Automata for Matching Patterns, In: *Vol 2: Linear Modeling: Backgroung and Application. Handbook of Formal Languages.*

[11] Crochemore, M., Rytter, W. *Jewels of Stringology*. World Scientific, New Jersey, 1994.

[12] J. E. Hopcroft, R. Motwani, J. D. Ullman. Introduction to automata theory, languages and computation, 2nd Edition. *ACM SIGACT News*, Volume 32, 2001

[13] C. S. Iliopoulos and M. Sohel Rahman. A new model to solve swap matching problem and efficient algorithms for short patterns. *Software Seminar Conference 2008*, volume 4910 of Lecture Notes in Computer Science, pages 316–327, 2008

[14] B. Lewin, Genes for sma: Multum in parvo. *Cell*, volume 8, pages 1–5, 1995

[15] Melichar, B., Holub, J., Polcar, J. *Text Searching Algorithms*. Available on: `http://stringology.org/athens/`, release November 2005, 2005.

[16] S. Muthukrishnan. New results and open problems related to non-standard stringology. *Combinatorial Pattern Matching*, volume 937 of Lecture Notes in Computer Science, pages 298–317, 1995

[17] M.O. Rabin, D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development 3*, Volume 3, pages 114–125, 1959

[18] B. Smyth. Computing Patterns in Strings. *Addison-Wesley*, 2003

[19] Weisstein, Eric W. Binet's Fibonacci Number Formula. *From MathWorld–A Wolfram Web Resource*. http://mathworld.wolfram.com/BinetsFibonacciNumberFormula.html