

On a compact encoding of the swap automaton

Kimmo Fredriksson¹ and Emanuele Giaquinta²

¹ School of Computing, University of Eastern Finland kimmo.fredriksson@uef.fi

² Department of Computer Science, University of Helsinki, Finland
emanuele.giaquinta@cs.helsinki.fi

Abstract. Given a string P of length m over an alphabet Σ of size σ , a swapped version of P is a string derived from P by a series of local swaps, i.e., swaps of adjacent symbols, such that each symbol can participate in at most one swap. We present a theoretical analysis of the nondeterministic finite automaton for the language $\bigcup_{P' \in \Pi_P} \Sigma^* P'$ (swap automaton for short), where Π_P is the set of swapped versions of P . Our study is based on the bit-parallel simulation of the same automaton due to Fredriksson, and reveals an interesting combinatorial property that links the automaton to the one for the language $\Sigma^* P$. By exploiting this property and the method presented by Cantone et al. (2010), we obtain a bit-parallel encoding of the swap automaton which takes $O(\sigma^2 \lceil k/w \rceil)$ space and allows one to simulate the automaton on a string of length n in time $O(n \lceil k/w \rceil)$, where $\lceil m/\sigma \rceil \leq k \leq m$.

1 Introduction

The *Pattern Matching with Swaps* problem (Swap Matching problem, for short) is a well-studied variant of the classic Pattern Matching problem. It consists in finding all occurrences, up to character swaps, of a pattern P of length m in a text T of length n , with P and T sequences of characters over a common finite alphabet Σ of size σ . More precisely, the pattern is said to match the text at a given location j if adjacent pattern characters can be swapped, if necessary, so as to make it identical to the substring of the text ending (or, equivalently, starting) at location j . All swaps are constrained to be disjoint, i.e., each character can be involved at most in one swap.

The Swap Matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [9]. The first result that improved over the naive $O(nm)$ -time bound is due to Amir et al. [1], who presented an $O(nm^{\frac{1}{3}} \log m)$ -time algorithm for binary alphabets and described how to reduce the case of a general alphabet to that of a binary one with a $O(\log \sigma)$ -time overhead. The best theoretical result to date is due to Amir *et al.* [2]. Their algorithm runs in time $O(n \log m)$ for binary alphabets and can also solve the case of general alphabets in time $O(n \log m \log \sigma)$ by using again the alphabet reduction technique of Amir *et al.* [1]. Both solutions are based on reducing the problem to convolutions. Note that this problem can also be solved using more general algorithms for Approximate String Matching [10], albeit with worse bounds.

There also exist different practical solutions, based on word-level parallelism. To our knowledge, the first one is due to Fredriksson [7], who presented a generalization of the nondeterministic finite automaton (NFA) for the language Σ^*P (prefix automaton) for the Swap Matching problem and a fast method to simulate it using bit-parallelism [3]. The resulting algorithm runs in $O(n\lceil m/w \rceil)$ -time and uses $O(\sigma\lceil m/w \rceil)$ space, where w is the machine word size in bits. In the same paper Fredriksson also presented a variant of the BNDM algorithm [11], based on the generalization of the NFA for the language of the suffixes of P (suffix automaton), which achieves sublinear time on average and runs in $O(nm\lceil m/w \rceil)$ -time in the worst-case. In 2008 Iliopoulos and Rahman presented a variant of Shift-Or for this problem, based on a Graph-Theoretic model [8]. Their algorithm runs in time $O(n\lceil m/w \rceil \log m)$ and uses $O(m\lceil m/w \rceil)$ space (the $\log m$ term can be removed at the price of $O(\sigma^2\lceil m/w \rceil)$ space). The improvement over the algorithm by Fredriksson is that the resulting bit-parallel simulation is simpler, in that it requires fewer bitwise operations. Later, Cantone and Faro presented an algorithm based on dynamic programming that runs in time $O(n\lceil m/w \rceil)$ and requires $O(\sigma\lceil m/w \rceil)$ space [5]. Subsequently Campanelli et al. presented a variant of the BNDM algorithm based on the same approach which runs in $O(nm\lceil m/w \rceil)$ -time in the worst-case [4].

In [6] Cantone et al. presented a technique to encode the prefix automaton in $O(\sigma^2\lceil k/w \rceil)$ space and simulate it on a string of length n in $O(n\lceil k/w \rceil)$ time, where $\lceil m/\sigma \rceil \leq k \leq m$. In this paper we extend this result to the NFA described in [7]. First, we present a theoretical analysis of this NFA, from which the correctness of the bit-parallel simulation presented in the same paper follows. We then show that, by exploiting the properties of this NFA that we reveal in the following, we can solve the Swap Matching problem in time $O(n\lceil k/w \rceil)$ and space $O(\sigma^2\lceil k/w \rceil)$, where $\lceil m/\sigma \rceil \leq k \leq m$, using the method presented in [6]. Our result also applies, with small changes, to the case of the generalized suffix automaton for the Swap Matching problem.

2 Notions and Basic Definitions

Given a finite alphabet Σ of size σ , we denote by Σ^m , with $m \geq 0$, the collection of strings of length m over Σ and put $\Sigma^* = \bigcup_{m \in \mathbb{N}} \Sigma^m$. We represent a string $P \in \Sigma^m$ as an array $P[0..m-1]$ of characters of Σ and write $|P| = m$ (in particular, for $m = 0$ we obtain the empty string ε). Thus, $P[i]$ is the $(i+1)$ -st character of P , for $0 \leq i < m$, and $P[i..j]$ is the substring of P contained between its $(i+1)$ -st and $(j+1)$ -st characters, inclusive, for $0 \leq i \leq j < m$. For any two strings P and P' , we write PP' to denote the concatenation of P and P' .

Given a string $P \in \Sigma^m$, we indicate with $\mathcal{A}(P) = (Q, \Sigma, \delta, q_0, F)$ the non-deterministic finite automaton (NFA) for the language Σ^*P of all words in Σ^* ending with an occurrence of P (prefix automaton for short), where:

$$- Q = \{q_0, q_1, \dots, q_m\} \quad (q_0 \text{ is the initial state})$$

– the transition function $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is defined by:

$$\delta(q_i, c) =_{\text{Def}} \begin{cases} \{q_0, q_1\} & \text{if } i = 0 \text{ and } c = P[0] \\ \{q_0\} & \text{if } i = 0 \text{ and } c \neq P[0] \\ \{q_{i+1}\} & \text{if } 1 \leq i < m \text{ and } c = P[i] \\ \emptyset & \text{otherwise} \end{cases}$$

– $F = \{q_m\}$ (F is the set of final states).

The valid configurations $\delta^*(q_0, S)$ which are reachable by the automaton $\mathcal{A}(P)$ on input $S \in \Sigma^*$ are defined recursively as follows:

$$\delta^*(q_0, S) =_{\text{Def}} \begin{cases} \{q_0\} & \text{if } S = \varepsilon, \\ \bigcup_{q' \in \delta^*(q_0, S')} \delta(q', c) & \text{if } S = S'c, \text{ for some } c \in \Sigma \text{ and } S' \in \Sigma^*. \end{cases}$$

Definition 1. A swap permutation for a string P of length m is a permutation $\pi : \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$ such that:

- (a) if $\pi(i) = j$ then $\pi(j) = i$ (characters are swapped);
- (b) for all i , $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters are swapped);
- (c) if $\pi(i) \neq i$ then $P[\pi(i)] \neq P[i]$ (identical characters are not swapped).

For a given string P and a swap permutation π for P , we write $\pi(P)$ to denote the swapped version of P , namely $\pi(P) = P[\pi(0)]P[\pi(1)] \dots P[\pi(m-1)]$.

Definition 2 (Pattern Matching with Swaps Problem). Given a text T of length n and a pattern P of length m , find all locations $j \in \{m-1, \dots, n-1\}$ for which there exists a swap permutation π of P such that $\pi(P)$ matches T at location j , i.e. $P[\pi(i)] = T[j-m+i+1]$, for $i = 0 \dots m-1$.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise **and** “&”, the bitwise **or** “|”, the **left shift** “ \ll ” operator (which shifts to the left its first argument by a number of bits equal to its second argument), and the unary bitwise **not** operator “ \sim ”.

2.1 1-factorization encoding of the prefix automaton

A 1-factorization \mathbf{u} of size k of a string P is a sequence $\langle u_1, u_2, \dots, u_k \rangle$ of nonempty substrings of P such that:

- (a) $P = u_1 u_2 \dots u_k$;
- (b) each factor u_j in \mathbf{u} contains at most one occurrence of any of the characters in the alphabet Σ , for $j = 1, \dots, k$.

The following result was presented in [6]:

Theorem 1 (cf. [6]). Given a string P of length m and a 1-factorization of P of length $\lceil m/\sigma \rceil \leq k \leq m$, we can encode the automaton $\mathcal{A}(P)$ in $O(\sigma^2 \lceil k/w \rceil)$ space and simulate it in time $O(n \lceil k/w \rceil)$ on a string of length n .

We briefly recall how the encoding of Theorem 1 works. A 1-factorization $\langle u_1, u_2, \dots, u_k \rangle$ of P induces a partition $\{Q_1, \dots, Q_k\}$ of the set $Q \setminus \{q_0\}$ of states of the automaton $\mathcal{A}(P)$, where

$$Q_i \stackrel{\text{Def}}{=} \{q_{r_i+1}, \dots, q_{r_{i+1}}\}, \text{ for } i = 1, \dots, k,$$

and $r_j = |u_1 u_2 \dots u_{j-1}|$, for $j = 1, \dots, k+1$. We denote with $q_{i,a}$ the unique state in Q_i with an incoming transition labeled by a , if such a state exists; otherwise $q_{i,a}$ is undefined. The configuration $\delta^*(q_0, Sa)$ of $\mathcal{A}(P)$ on input Sa can then be encoded by the pair (D, a) , where D is the bit-vector of size k such that $D[i]$ is set iff $q_{i,a} \in \delta^*(q_0, Sa)$. We also denote with $\text{id}(i, u_i[j]) = r_i + j$ the position of symbol $u_i[j]$ in P , for $0 \leq j \leq |u_i| - 1$. Equivalently, $\text{id}(i, u_i[j])$ is the index of state $q_{i,u_i[j]}$ in the original automaton.

3 An analysis of the swap automaton

Let P be a pattern of length m and let Π_P be the set including all the swapped versions of P . The swap automaton of P is the nondeterministic finite automaton that recognizes all the words in Σ^* ending with a swapped version of P . Formally, it is the NFA $\mathcal{A}_\pi(P) = (Q, \Sigma, \delta, q_0, F)$, where:

- $Q = \{q_0, q_1 \dots, q_{2m-1}\}$
- the transition function $\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$ is defined by:

$$\delta(q_i, c) \stackrel{\text{Def}}{=} \begin{cases} \{q_0, q_1\} & \text{if } i = 0 \text{ and } c = P[0] \\ \{q_0, q_{m+1}\} & \text{if } i = 0 \text{ and } c = P[1] \\ \{q_0\} & \text{if } i = 0 \text{ and } c \neq P[0] \text{ and } c \neq P[1] \\ \{q_{i+1}\} & \text{if } 1 \leq i < m \text{ and } c = P[i] \\ \{q_{i+m+1}\} & \text{if } 1 \leq i < m - 1 \text{ and } c = P[i + 1] \\ \{q_{i-m+1}\} & \text{if } m + 1 \leq i < 2m \text{ and } c = P[i - m - 1] \\ \emptyset & \text{otherwise} \end{cases}$$

- $F = \{q_m\}$

The language accepted by $\mathcal{A}_\pi(P)$ is $\mathcal{L}(\mathcal{A}_\pi(P)) = \bigcup_{P' \in \Pi_P} \Sigma^* P'$. An example of this automaton for the string *cagca* is depicted in Fig. 1. Compared to the NFA $\mathcal{A}(P)$ for the language $\Sigma^* P$, this automaton has $m - 1$ additional states and $2m - 2$ additional transitions. To our knowledge, this automaton was described for the first time by Fredriksson in [7]. In the same paper, Fredriksson presented an efficient simulation of this automaton based on word-level parallelism. Let $\phi(S) = S'$ be the string of length $|S|$ defined as follows:

$$S'[i] = \begin{cases} S[i] & \text{if } i \geq \lfloor m/2 \rfloor 2 \\ S[i - 1] & \text{if } i \bmod 2 = 1 \\ S[i + 1] & \text{if } i \bmod 2 = 0 \end{cases}$$

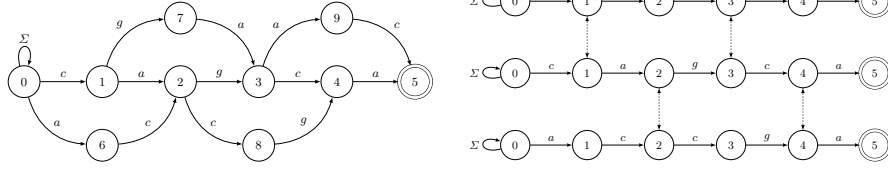


Fig. 1. (a) The swap automaton for the pattern $cagca$; (b) The decomposition of the swap automaton for the pattern $cagca$.

The method is based on the decomposition of the swap automaton for P into the three automata $\mathcal{A}(P)$, $\mathcal{A}(P_e)$ and $\mathcal{A}(P_o)$, where $P_e = P[0]\phi(P[1 \dots m-1])$ and $P_o = \phi(P)$. In the case of the string $cagca$ we have that $P_e = cgaac$ and $P_o = accga$. The corresponding automata are depicted in Fig. 1. Observe that all the automata have exactly $m + 1$ states. We denote with q_i^1 , q_i^2 and q_i^3 the i -th state of the automata $\mathcal{A}(P)$, $\mathcal{A}(P_e)$ and $\mathcal{A}(P_o)$, respectively. Likewise for the corresponding transition functions. Given a string S , let $D_j^i(S)$ be the set recursively defined as

$$D_j^i(S) = \begin{cases} \bigcup_{q \in D_{j-1}^i(S) \cup C_{j-1}^i(S)} \delta_i(q, S[j]) & \text{if } 1 \leq j \leq |S| - 1 \\ \delta_i(q_0^i, S[0]) & \text{if } j = 0. \end{cases}$$

for $i = 1, \dots, 3$, where

$$C_j^1(S) = \{q_i^1 \mid (i \bmod 2 = 1 \wedge q_i^2 \in D_j^2(S)) \vee (i \bmod 2 = 0 \wedge q_i^3 \in D_j^3(S))\},$$

$$C_j^2(S) = \{q_i^2 \mid (i \bmod 2 = 1 \wedge q_i^1 \in D_j^1(S))\},$$

$$C_j^3(S) = \{q_i^3 \mid (i \bmod 2 = 0 \wedge q_i^1 \in D_j^1(S))\},$$

for $j = 0, \dots, |S| - 1$. The idea is to simulate the three automata simultaneously on S . However, at each iteration, we also activate some states of each automaton depending on the configuration of the others. More precisely, we activate state q_i^2 if i is odd and state q_i^1 is active, and viceversa. Similarly, we activate state q_i^3 if i is even and state q_i^1 is active, and viceversa. The sets D_j^i encode the described configurations. Now, consider the automaton $\mathcal{A}_\pi(P)$. It is not hard to see that the following Lemma holds:

Lemma 1. *In a simulation of the automaton $\mathcal{A}_\pi(P)$ on a given string S , state q_i is active at the j -th iteration, i.e., $q_i \in \delta^*(q_0, S[0 \dots j])$, iff one of the following three conditions hold:*

1. $1 \leq i \leq m$ and $q_i^1 \in D_j^1(S)$;
2. $m < i < 2m$, $i - m$ is even and $q_{i-m}^2 \in D_j^2(S)$;
3. $m < i < 2m$, $i - m$ is odd and $q_{i-m}^3 \in D_j^3(S)$.

Hence, to simulate $\mathcal{A}_\pi(P)$ it is enough to simulate the automata $\mathcal{A}(P)$, $\mathcal{A}(P_e)$ and $\mathcal{A}(P_o)$, and compute the sets D_j^i . To this end, Fredriksson uses the well known technique of bit-parallelism [3] to encode each automaton in $O(\sigma \lceil m/w \rceil)$ space. For a given string T of length n , the simulation of the three automata on T can be then computed in time $O(n \lceil m/w \rceil)$, since the number of automata is constant. For the details concerning the bit-parallel simulation see [7].

We now show how to exploit Lemma 1 to devise an improved algorithm for the Swap Matching problem. Our result will be a combination of Lemma 1 and Theorem 1. The idea is to encode each automaton using a 1-factorization of the corresponding string. However, for the simulation to work, we must be able to compute the sets $C_j^i(S)$ in constant time (per word), which is not trivial using the 1-factorization encoding. The first prerequisite for a constant time computation is the following property:

Property 1. For any pair of states (q_i^1, q_i^2) or (q_i^1, q_i^3) , the two states in the pair map onto the same bit position in the bit-vector encoding of the corresponding automaton.

For this to hold, given a sequence of factorizations $\mathbf{u}^1, \mathbf{u}^2, \dots, \mathbf{u}^\ell$ we must have that

1. $|\mathbf{u}^i| = |\mathbf{u}^j|$, for any $1 \leq i, j \leq \ell$
2. $|u_l^i| = |u_l^j|$, for any $1 \leq i, j \leq \ell$ and $1 \leq l \leq |\mathbf{u}^i|$

These conditions are not satisfied in general by the minimal 1-factorizations of the strings. For example, the minimal 1-factorizations of *cagca*, *cgaac* and *accga* are $\langle \text{cag}, \text{ca} \rangle$, $\langle \text{cga}, \text{ac} \rangle$ and $\langle \text{ac}, \text{cga} \rangle$, and the last factorization does not satisfy condition 2. Let $\mathbf{1}\text{-len}(S, s) = i$, where i is the length such that all the symbols in $S[s \dots s+i-1]$ are distinct and either $s+i-1 = |S|-1$ or $S[s+i]$ occurs in $S[s \dots s+i-1]$, for $s = 0, \dots, |S|-1$. We introduce the following definition:

Definition 3. Given a sequence \mathcal{S} of strings S_1, S_2, \dots, S_ℓ of the same length, we define the 1-collection of \mathcal{S} as the sequence of 1-factorizations $\mathbf{u}^1, \mathbf{u}^2, \dots, \mathbf{u}^\ell$ of length k , where $\mathbf{u}^i = \langle u_1^i, u_2^i, \dots, u_k^i \rangle$, such that

- (a) $S_i = u_1^i u_2^i \dots u_k^i$;
- (b) $|u_j^i| = \min_{S \in \mathcal{S}} \mathbf{1}\text{-len}(S, \sum_{t=1}^{j-1} |u_t^i|)$.

Observe that the 1-collection of \mathcal{S} satisfies conditions 1 and 2.

For example, the 1-collection of *cagca*, *cgaac* and *accga* is $\langle \text{ca}, \text{g}, \text{ca} \rangle$, $\langle \text{cg}, \text{a}, \text{ac} \rangle$ and $\langle \text{ac}, \text{c}, \text{ga} \rangle$. Indeed, we can encode the automata $\mathcal{A}(P)$, $\mathcal{A}(P_e)$ and $\mathcal{A}(P_o)$ using Theorem 1 and the 1-collection of P, P_e, P_o in space $O(\sigma^2 \lceil k/w \rceil)$, where k is the size of any 1-factorization in the 1-collection of P, P_e, P_o . By definition, the 1-collection of P, P_e, P_o satisfies conditions 1 and 2, and thus Property 1 holds.

Before continuing, we first bound the size k of the factorizations in the 1-collection of P, P_e and P_o .

Lemma 2. *Let k' be the size of a minimal 1-factorization of P and let k be the size of any factorization in the 1-collection of P , P_e and P_o . Then we have $k \leq \min(3k' - 2, m)$.*

Proof. Let $\langle u_1, u_2, \dots, u_{k'} \rangle$ be the (greedy) minimal 1-factorization of P such that $|u_j| = 1\text{-len}(P, \sum_{l=1}^{j-1} |u_l|)$, for $j = 1, \dots, k'$. Let $s = \sum_{l=1}^{j-1} |u_l|$ for a given j , and suppose that $|u_j| = 1\text{-len}(P, s) = i$, so that $P[s+i]$ occurs in $P[s \dots s+i-1]$. If $s+i$ is even, then $P_e[s+i-1] = P[s+i]$ and $P_o[s+i+1] = P[s+i]$; viceversa if $s+i$ is odd. Suppose that $s+i$ is even (the other case is analogous).

If s is even then $P_o[s \dots s+i-1]$ is a permutation of $P[s \dots s+i-1]$, which implies $1\text{-len}(P_o, s) \geq i$. Instead, in the case of P_e , $P_e[s+1 \dots s+i-2]$ is a permutation of $P[s+1 \dots s+i-2]$. This implies that $1\text{-len}(P_e, s+1) \geq i-2$.

If s is odd then $P_e[s \dots s+i-2]$ is a permutation of $P[s \dots s+i-2]$, which implies that $1\text{-len}(P_e, s) \geq i-1$. Instead, in the case of P_o , $P_o[s+1 \dots s+i-1]$ is a permutation of $P[s+1 \dots s+i-1]$. This implies that $1\text{-len}(P_o, s+1) \geq i-1$.

Observe that $1\text{-len}(S, s) \geq i$ implies $1\text{-len}(S, s+1) \geq i-1$. In both cases, we assume pessimistically that $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, s) = 1$, $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, s+1) = i-2$, and $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, s+i-1) = 1$. This arrangement is compatible with the constraints described above.

In this way each factor u_j covers three factors in the 1-collection of P , P_e and P_o . However, a finer analysis reveals that u_1 and u_k can cover two factors only. Indeed, in the case of u_1 we have that $P_e[0 \dots i-2]$ is a permutation of $P[0 \dots i-2]$ and $P_o[0 \dots i-1]$ is a permutation of $P[0 \dots i-1]$, so we can assume $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, 0) = i-1$ and $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, i-1) = 1$. Instead, in the case of u_k we have that $P_o[s \dots m-1]$ is a permutation of $P[s \dots m-1]$ and $P_e[s+1 \dots m-1]$ is a permutation of $P[s+1 \dots m-1]$, if s is even, viceversa if s is odd. So we can assume $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, s) = 1$ and $\min_{S \in \{P, P_e, P_o\}} 1\text{-len}(S, s+1) = m-s-1$. The claim then follows. \square

We now describe a property of the 1-collection of strings P , P_e and P_o that will be the key for the constant time computation of C_j^i :

Lemma 3. *Let \mathbf{u}^1 , \mathbf{u}^2 and \mathbf{u}^3 be the 1-collection of P , P_e and P_o . Then, the following facts hold:*

- $\text{id}^1(i, P[j]) = \text{id}^2(i, P[j-1])$
- $\text{id}^2(i, P_e[j]) = \text{id}^1(i, P_e[j-1])$ if $j \bmod 2 = 0$
- $\text{id}^1(i, P[j]) = \text{id}^3(i, P[j-1])$
- $\text{id}^3(i, P_o[j]) = \text{id}^1(i, P_o[j-1])$ if $j \bmod 2 = 1$

for $1 \leq i \leq k$ and $\max(r_i^1, 1) \leq j \leq r_i^1 + |u_i^1| - 1$.

Proof. By definition, $\text{id}^1(i, P[j]) = j$, for any i, j as above, since \mathbf{u}^1 is a factorization of P . Similarly, $\text{id}^2(i, P_e[j]) = j$, since $r_i^2 = r_i^1$ and $|u_i^2| = |u_i^1|$. If $j \bmod 2 = 0$, then $P_e[j] = P[j-1]$ and $P[j] = P_e[j-1]$ so that $\text{id}^2(i, P_e[j]) = \text{id}^2(i, P[j-1])$ and $\text{id}^1(i, P[j]) = \text{id}^1(i, P_e[j-1])$. The case of $j \bmod 2 = 1$ is analogous with P_o and id^3 in place of P_e and of id^2 , respectively. \square

We now present how to compute the sets C_j^2 and C_j^3 . The case of C_j^1 is analogous. More precisely, we need to compute the 1-factorization encoding of C_j^2 and C_j^3 , given the pair $(D^1, T[j])$ encoding the set $D_j^1(T)$. Let $E(c)$ be a bit-vector of k bits such that bit i is set in $E(c)$ iff $\text{id}^1(i, c)$ is even, for any $c \in \Sigma$. First we compute the bit-vector D' such that bit i is set iff bit i is set in D^1 and $\text{id}^1(i, T[j])$ is even. This can be done in constant time by performing a bitwise and of D^1 with $E(T[j])$. Observe that the pair $(D', T[j])$ encodes the set $\{q_i^1 \mid (i \bmod 2 = 1 \wedge q_i^1 \in D_j^1(T))\}$. We claim that the pair $(D', T[j-1])$ encodes the set C_j^2 . This follows by Lemma 3 by observing that if bit i is set in D' then $\text{id}^1(i, T[j]) = \text{id}^2(i, T[j-1])$. The case of C_j^3 is symmetric, i.e., the pair $(D^1 \& \sim E(T[j]), T[j-1])$ encodes C_j^3 .

Given a string of length n , we can then simulate the swap automaton using Lemma 1 in time $O(n \lceil k/w \rceil)$. Hence, we obtain the following result:

Theorem 2. *Given a string P of length m , we can encode the automaton $\mathcal{A}_\pi(P)$ in $O(\sigma^2 \lceil k/w \rceil)$ space, where $\lceil m/\sigma \rceil \leq k \leq m$, and simulate it in time $O(n \lceil k/w \rceil)$ on a string of length n .*

References

1. Amihood Amir, Yonatan Aumann, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Pattern matching with swaps. *J. Algorithms*, 37(2):247–266, 2000.
2. Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003.
3. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
4. Matteo Campanelli, Domenico Cantone, and Simone Faro. A new algorithm for efficient pattern matching with swaps. In *IWOCA*, volume 5874 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2009.
5. Domenico Cantone and Simone Faro. Pattern matching with swaps for short patterns in linear time. In *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009.
6. Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Inf. Comput.*, 213:3–12, 2012.
7. Kimmo Fredriksson. Fast algorithms for string matching with and without swaps. 2000. Unpublished manuscript, <http://www.cs.uef.fi/~fredriks/pub/papers/sm-w-swaps.pdf>.
8. Costas S. Iliopoulos and M. Sohel Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 2008.
9. S. Muthukrishnan. New results and open problems related to non-standard stringology. In *CPM*, volume 937 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1995.
10. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
11. Gonzalo Navarro and Mathieu Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5:4, 2000.