Swap Matching in Strings by Simulating Reactive Automata

Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
faro@dmi.unict.it

Abstract. The *pattern matching problem with swaps* consists in finding all occurrences of a pattern P in a text T, when disjoint local swaps in the pattern are allowed. In this paper we introduce a new theoretical approach to the problem based on a reactive automaton modeled after the pattern, and provide two efficient non standard simulations of the automaton, based on bit-parallelism. The first simulation can be implemented by at least 7 bitwise operations, while the second one involves only 2 bitwise operations to simulate the automaton behavior, when the input pattern satisfies particular conditions. The resulting algorithms achieve $\mathcal{O}(n)$ worst-case time complexity with patterns whose length is comparable to the word-size of the target machine.

Keywords: pattern matching with swaps, nonstandard pattern matching, combinatorial algorithms on words, bit parallelism

1 Introduction

The *string matching problem with swaps* (swap matching problem, for short) is a well-studied variant of the classic string 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 drawn from a same finite alphabet Σ of size σ. More precisely, the pattern is said to *swap-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 in at most one swap. Moreover, we make the agreement that identical adjacent characters are not allowed to be swapped.

This problem is of relevance in practical applications such as text and music retrieval, data mining, network security, and many others. Following [18], we also mention a particularly important application of the swap matching problem in biological computing, specifically in the process of translation in molecular biology, with the genetic triplets (otherwise called *codons*). In such application one wants to detect the possible positions of the start and stop codons of a mRNA in a biological sequence and find hints as to where the flanking regions are, relative to the translated mRNA region.

In the field of natural language processing the transposition of two adjacent characters in a text is a most common typing error. Thus several algorithms for the spell-checking problem are designed in order to identify swaps of characters in their matching engines.

The swap matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [19]. The first nontrivial result was reported by Amir *et al.* [1], who provided a $\mathcal{O}(nm^{\frac{1}{3}} \log m)$-time algorithm in the case of alphabet sets of size 2, showing also that the case of alphabets of size exceeding 2 can be reduced

Simone Faro: Swap Matching in Strings by Simulating Reactive Automata, pp. 7–20.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0 © Czech Technical University in Prague, Czech Republic

to that of size 2 with a $\mathcal{O}(\log^2 \sigma)$-time overhead, subsequently reduced to $\mathcal{O}(\log \sigma)$ in the journal version [2]. Amir *et al.* [4] studied some rather restrictive cases in which a $\mathcal{O}(m \log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* [3] solved the swap matching problem in $\mathcal{O}(n \log m \log \sigma)$-time. The above solutions are all based on the fast Fourier transform (FFT).

In 2008 the first attempt to provide an efficient solution to the swap matching problem without using the FFT technique has been presented by Iliopoulos and Rahman in [18]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on the bit-parallelism technique [5], which runs in $\mathcal{O}((n+m) \log m)$-time, in the case of short patterns.

In 2009, Cantone and Faro [9,6] presented a new efficient algorithm, named Cross-Sampling (CS), which simulates a non-deterministic automaton with $2m$ states and $3m-2$ transitions. The CS algorithm though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named Bit-Parallel-Cross-Sampling (BPCS), which achieves $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(\sigma)$ space complexity in the case of short patterns fitting in few machine words.

In this paper we present a new theoretical model to solve the swap matching problem in strings, based on reactive automata [16,13]. Specifically the automaton used in our model has only m states and at most $3m-2$ transitions. Moreover it has $8m-12$ reactive links. We propose also two different non-standard simulations of the automaton based on bit parallelism. The first approach works by encoding the transitions of the automaton and leads to an algorithm with linear worst case time complexity and $\mathcal{O}(\sigma)$-space complexity, in the case of short patterns. Our second approach uses a simpler encoding and, under suitable conditions, it turns out to be very efficient in practice, achieving $\mathcal{O}(n)$ worst case time complexity and requiring $\mathcal{O}(\sigma^2)$-extra space. However in the general case it works as an oracle and needs an extra verification phase when a candidate occurrence is found. In this case its worst case time complexity is $\mathcal{O}(nm)$.

The paper is organized as follows. In Section 2 we introduce some notions and definitions. Then in Section 3 we introduce the notion of swap reactive automaton and propose two non standard simulations of it based on bit parallelism. We draw our conclusions in Section 5.

2 Notations and Definitions

Given a string $P = p_0 p_1 \cdots p_{m-1}$ of length $m \geq 0$, we represent it as a finite array $P[0\,..\,m-1]$. In particular, for $m = 0$ we obtain the empty string ε. We denote by p_i (or $P[i]$) the $(i+1)$-st character of P, for $0 \leq i < m$, and by $P[i\,..\,j]$ the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < m$. For any two strings P and P' we say that P' is a prefix of P if $P' = P[0\,..\,i-1]$, for some $0 \leq i \leq m$. We denote by P_i the nonempty prefix $P[0\,..\,i]$ of P of length $i+1$, for $0 \leq i < m$.

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

(a) if $\pi(i) = j$ then $\pi(j) = i$ (characters at positions i and j are swapped);
(b) for all i, $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters can be swapped);
(c) if $\pi(i) \neq i$ then $P[\pi(i)] \neq P[i]$ (identical characters can not be 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 $\pi(P) = P[\pi(0)] \cdot P[\pi(1)] \cdots P[\pi(m-1)]$.

Definition 2. *Given a text T of length n and a pattern P of length m, P is said to* swap-match *(or to have a* swapped occurrence*)* at location $j \geq m-1$ of T *if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j, i.e., $\pi(P) = T[j-m+1\,..\,j]$. In such a case we write $P \propto T_j$.*

It can be proved [7] that if P has a swap occurrence at location j of the text T, then the permutation π such that $\pi(P)$ matches T at location j, is unique.

A finite automaton (FA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the collection of final states, Σ is an alphabet, and $\delta \subseteq (Q \times \Sigma \times Q)$ is the transition relation.

Definition 3 (Switch Reactive Transformation). *Let $\delta \subseteq (Q \times \Sigma \times Q)$ be the transition relation of an automaton A and let $\varphi \subseteq \delta$. Let T^+, T^- be two subsets of $\delta \times \delta$. A transformation $\delta \to \delta^\varphi$, for $\varphi \subseteq \delta$, is defined as follows*

$$\begin{aligned}\delta^\varphi = (\delta \setminus \{\gamma \mid \gamma \in \delta \text{ and } \exists\, \tau \in \varphi \text{ such that } (\tau, \gamma) \in T^-\}) \\ \cup \{\gamma \mid \gamma \in \delta \text{ and } \exists\, \tau \in \varphi \text{ such that } (\tau, \gamma) \in T^+\}\end{aligned}$$

The reactive links are intended to be applied simultaneously.

Definition 4 (Switch Reactive Automaton). *A reactive automaton is an ordinary non-deterministic automaton with a switch reactive transformation, i.e. a triple $R = (A, T^+, T^-)$ which defines the switch reactive transformation above.*

Definition 5 (Nondeterministic Run). *Let $S = s_0 s_1 \cdots s_{n-1}$ be a word on the alphabet Σ and let $R = (A, T^+, T^-)$ be a reactive automaton, where $A = (Q, q_0, \Sigma, F, \delta)$. The nondeterministic run over S is a sequence of pairs (Q_k, δ_k), for $k = 0, \ldots, n$, with $Q_k \subseteq Q$ is the set of active states, and $\delta_k \subseteq \delta$ is the set of active transitions. It can be formally defined as follows:*

$$(Q_k, \delta_k) = \begin{cases} (\{q_0\}, \delta) & \text{if } k = 0 \\ (\{q \mid (r, s_{k-1}, q) \in \delta_{k-1} \text{ with } r \in Q_{k-1}\}, \delta_{k-1}^\varphi) & \text{if } k > 0 \end{cases}$$

where $\varphi = \{(r, s_{k-1}, q) \mid (r, s_{k-1}, q) \in \delta_{k-1} \text{ and } r \in Q_{k-1}\}$.

We say that the string S is accepted by the reactive automaton if the nondeterministic run $\langle (Q_0, \delta_0), (Q_1, \delta_1), \ldots, (Q_n, \delta_n) \rangle$ over S is such that $Q_n \cap F \neq \emptyset$.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", and the `left shift` "$\ll$" operator (which shifts its first argument to the left by a number of bits equal to its second argument). We say that a bit *is set* to indicate that its value is equal to 1.

3 A New Algorithm Based on Reactive Automata

Reactive automata [16,13] are used to reduce dramatically the number of states in both deterministic and the non-deterministic automata. As stated by Definition 3 and Definition 4, a reactive automaton has extra links whose role is to change the behavior of the automaton itself. In this section we present a new solution for the swap matching problem in strings based on reactive automata.

In particular, we show in Section 3.1 how to construct a reactive automaton which recognizes all swap occurrence of a given input pattern and prove its correctness. Then we give two non-standard approach to simulate such automaton in Section 3.2 and in Section 3.3.

3.1 The Swap Reactive Automaton

The automaton which we use in our solution is called *swap reactive automaton*. It is defined as follows.

Definition 6 (Swap Reactive Automaton). *Let P be a pattern of length m over an alphabet Σ. The Swap Reactive Automaton (SRA) for P is a Reactive Automaton $R = (A, T^+, T^-)$, with $A = (Q, \Sigma, \delta, q_0, F)$, such that*

- *$Q = \{q_0, q_1, \ldots, q_m\}$ is the set of states;*
- *q_0 is the initial state;*
- *$F = \{q_m\}$ is the set of final states;*
- *δ is the transition relation defined as*

$$\begin{array}{ll} \delta = \{(q_i, p_i, q_{i+1}) \mid 0 \leq i < m\} \cup & \textit{no swaps} \\ \quad \{(q_i, p_{i+1}, q_{i+1}) \mid 0 \leq i < m-1 \textit{ and } p_i \neq p_{i+1}\} \cup & \textit{start of a swap} \\ \quad \{(q_i, p_{i-1}, q_{i+1}) \mid 1 \leq i < m \textit{ and } p_i \neq p_{i-1}\} \cup & \textit{end of a swap} \\ \quad \{(q_0, \Sigma, q_0)\} & \textit{self loop} \end{array}$$

- *T^+ is the set of (switch on) reactive links defined as*

$$\begin{array}{rl} T^+ = & \{((q_i, p_i, q_{i+1}), (q_i, p_{i-1}, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1, \} \cup \\ & \{((q_i, p_{i+1}, q_{i+1}), (q_i, p_{i-1}, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1\} \cup \\ & \{((q_i, p_{i-1}, q_{i+1}), (q_i, p_i, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1\} \cup \\ & \{((q_i, p_{i-1}, q_{i+1}), (q_i, p_{i+1}, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1\} \end{array}$$

- *T^- is the set of (switch off) reactive links defined as*

$$\begin{array}{rl} T^- = & \{((q_i, p_i, q_{i+1}), (q_{i+1}, p_i, q_{i+2})) \in (\delta \times \delta) \mid 0 \leq i < m-1\} \cup \\ & \{((q_i, p_{i-1}, q_{i+1}), (q_{i+1}, p_i, q_{i+2})) \in (\delta \times \delta) \mid 1 \leq i < m-1\} \cup \\ & \{((q_i, p_{i+1}, q_{i+1}), (q_{i+1}, p_{i+1}, q_{i+2})) \in (\delta \times \delta) \mid 0 \leq i < m-1\} \cup \\ & \{((q_i, p_{i+1}, q_{i+1}), (q_{i+1}, p_{i+2}, q_{i+2})) \in (\delta \times \delta) \mid 0 \leq i < m-2\} \end{array}$$

The swap reactive automaton of a pattern P of length m has exactly $m+1$ states, (at most) $3m-2$ transitions and (at most) $8m-12$ reactive links.

To simplify the notation we will use the symbol $\tau(i, j)$ to indicate the standard transition starting from state q_i and labeled by character p_j, i.e. $\tau(i, j) = (q_i, p_j, q_{i+1})$. Since all standard transitions of the automaton starting from state q_i, reach the state q_{i+1}, the notation defined above is not ambiguous.

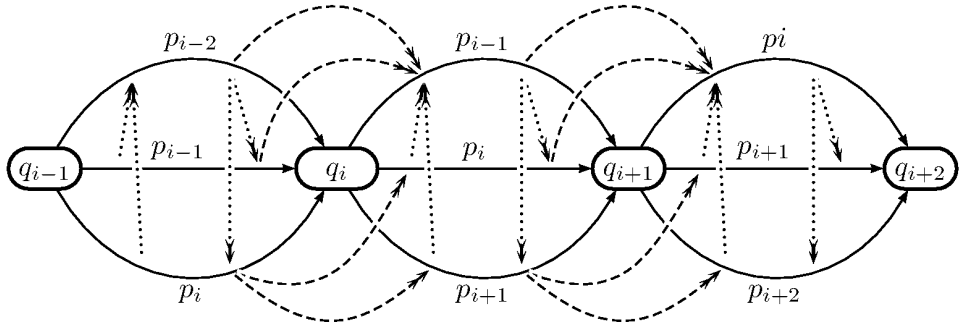

Figure 1. The general structure of a swap reactive automaton. Standard transitions in δ are represented with solid lines, reactive links in T^- are represented with dashed lines, while reactive links in T^+ are represented with dotted lines.

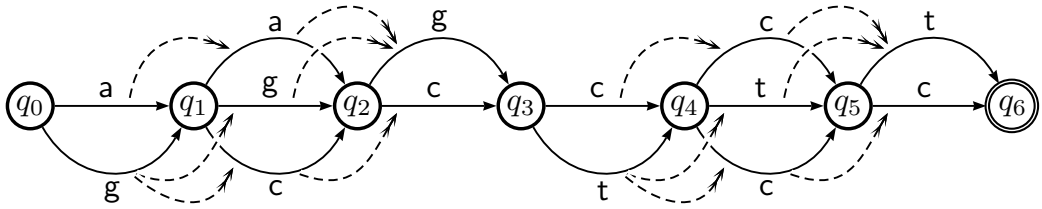

Figure 2. The swap reactive automaton for the pattern $P =$ agcctc. Standard transitions are represented with solid lines while reactive links in T^- are represented with dashed lines. Reactive links in T^+ are not represented.

Figure 1 shows the general structure of a portion (from state q_{i-1} to state q_{i+2}) of the swap reactive automaton, while Figure 2 shows the swap reactive automaton constructed for the pattern $P =$ agcctc. Each state has 3 standard transitions to the next state, with the exception states q_0 and q_{m-1}. Specifically state q_i, for $0 < i < m-1$, has transitions to state q_{i+1} labeled by characters p_{i-1}, p_i and p_{i+1}, respectively. Transitions labeled by character p_i are not involved in any swap, those labeled by character p_{i+1} start a new swap, while transitions labeled by character p_{i-1} end a previously started swap. Due to its external positions, state q_0 has only 2 transitions reaching state q_1. Similarly state q_{m-1} has only two transitions reaching state q_m.

When a new swap starts (with a transition from q_i to q_{i+1} labeled by p_{i+1}) two reactive links switch off the next transitions from state q_{i+1} with the exception of the transition which ends the swap. Otherwise, when a swap ends (following a transition labeled by p_{i-1}) or no swap is involved in the current transition (following a transition labeled by p_i) a reactive link switches off the next transition which ends a swap.

The reactive links in T^+ allows all transitions from q_i to q_{i+1} to be active after any step. The self loop of the initial state allows an occurrence of the pattern to begin at any position of the text.

The two following properties of an SRA trivially follows by Definition 6.

Property 7. There is no state $q_i \in Q$, with $0 \leq i < m$, such that $\tau(i,i)$ and $\tau(i,i+1)$ are both in δ, and $p_i = p_{i+1}$.

Property 8. There is no state $q_i \in Q$, with $0 \leq i < m$, such that $\tau(i,i)$ and $\tau(i,i-1)$ are both in δ, and $p_i = p_{i-1}$.

In what follows we assume that $P = p_0p_1p_2\cdots p_{m-1}$ is a string of length m and $T = t_0t_1t_2\cdots t_{n-1}$ is a string of length n, both over the alphabet Σ. Moreover we assume that $R = (A, T^+, T^-)$ is the SRA of P.

Lemma 9. *Let $\langle (Q_0,\delta_0),(Q_1,\delta_1),\ldots,(Q_n,\delta_n)\rangle$ be a nondeterministic run of the SRA R over the string T. If state $q_i \in Q_j$, with $i > 0$, then only one of the following relations holds*

(a) $\tau(i,i) \notin \delta_j$ and $\tau(i,i+1) \notin \delta_j$, or
(b) $\tau(i,i-1) \notin \delta_j$

Proof. Without loose in generality, we suppose that p_i, p_{i-1} and p_{i-2} are all different characters.

If $q_i \in Q_j$ then it has been reached from state q_{i-1} through one of three transitions starting from q_{i-1}, i.e. $\tau(i-1,i-2)$, $\tau(i-1,i-1)$ or $\tau(i-1,i)$. Since both $(\tau(i-1,i-2),\tau(i,i-1))$ and $(\tau(i-1,i-1),\tau(i,i-1))$ are the only reactive links in T^- starting from $\tau(i-1,i-2)$ and $\tau(i-1,i-1)$, it follows that if q_i is reached through transitions label by p_{i-2} or p_{i-1} we have that $\tau(i-1,i-1) \notin \delta_j$. Moreover $\tau(i,i)$ and $\tau(i,i+1)$ are both in δ.

Similarly, if q_i is reached through transitions label by p_i, since the reactive links $(\tau(i-1,i),\tau(i,i))$ and $(\tau(i-1,i),\tau(i,i+1))$ are the only in T^- starting from $\tau(i-1,i)$, we have that $\tau(i,i) \notin \delta_j$ and $\tau(i,i+1) \notin \delta_j$, while $\tau(i,i-1) \in \delta_j$. □

The following two technical results prove that the swap reactive automaton given in Definition 6 recognizes all and only the strings ending with an occurrence of P.

Lemma 10. *Let $\langle (Q_0,\delta_0),(Q_1,\delta_1),\ldots,(Q_n,\delta_n)\rangle$ be a nondeterministic run of the SRA R over the string T. If state $q_{i+1} \in Q_{j+1}$, for $0 < i < m$ and $0 < j < n$, then one of the following relations holds*

(a) $P_i \propto T_j$, or
(b) $P_{i-1} \propto T_{j-1}$ and $p_{i+1} = t_j$

Proof. We prove this result by induction on the value of i.

When $i = 0$ we have that $q_1 \in Q_1$, i.e. q_1 is active after we read character t_j of the text. Thus it must be $t_j = p_0$ (in which case condition (a) holds) or $t_j = p_1$ (in which case condition (b) holds).

Suppose now that the result holds for values less than i and prove it for $i > 0$. Since q_{i+1} is active after we read t_j it follows that q_i has been active after we read character t_{j-1}. This because q_{i+1} can be reached only from state q_i.

It implies by induction that

(1) $P_{i-1} \propto T_{j-1}$ or
(2) $P_{i-2} \propto T_{j-2}$ and $p_i = t_{j-1}$.

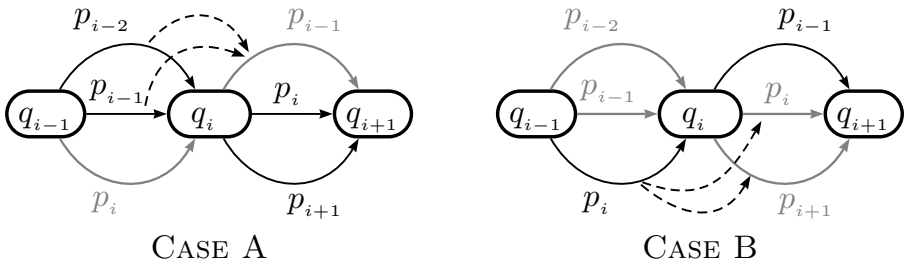

Figure 3. Two different conditions described in Lemma 10. Active transitions are represented in black lines, while switched off transitions are represented in light gray lines.

If (1) holds (see Figure 3 CASE A) then state q_i has been reached through the transition labeled by p_{i-1} or through the transition labeled by p_{i-2}. Since both reactive links $(\tau(i-1,i-2),\tau(i,i-1))$ and $(\tau(i-1,i-1),\tau(i,i-1))$ are in the set T^-, it turns out that transition $\tau(i,i-1) \notin \delta_{j-1}$, before reading t_j. As a consequence, state q_{i+1} can be reached only through the transition labeled by character p_i (in which case condition (a) holds) or through transition labeled by character p_{i+1} (in which case condition (b) holds).

Otherwise, if condition (2) holds (see Figure 3 CASE B), it follows that state q_i has been reached through transition labeled by character p_i. Since both reactive links $(\tau(i-1,i),\tau(i,i))$ and $(\tau(i-1,i),\tau(i,i+1))$ are in T^-, it turns out that transitions $\tau(i,i)$ and $\tau(i,i+1)$ are switched off. As a consequence state q_{i+1} can be reached only through transition $\tau(i,i-1)$, in which case condition (a) holds. □

Corollary 11. *Let $\langle(Q_0,\delta_0),(Q_1,\delta_1),\ldots,(Q_n,\delta_n)\rangle$ be a nondeterministic run of the SRA R over the string T. If state $q_m \in Q_{j+1}$ then $P \propto T_j$.*

Proof. Observe that state q_m can be reached only by state q_{m-1}, through the transition labeled by character p_{m-1} (no swap involved) or through the transition labeled by character p_{m-2} (end of a swap). The result follows by such observation and by Lemma 10. □

Lemma 12. *Let $\langle(Q_0,\delta_0),(Q_1,\delta_1),\ldots,(Q_n,\delta_n)\rangle$ be a nondeterministic run of the SRA R over the string T. If $P_i \propto T_j$ then $q_{i+1} \in Q_{j+1}$.*

Proof. We prove the lemma by induction on the value of i. For the base case, observe that if $P_0 \propto T_j$ then we have $p_0 = t_j$. Since q_0 is always active, due to the self loop, and $(q_0,p_0,q_1) \in \delta$, it follows that q_1 is active after we read t_j.

Suppose now that $i > 0$ and assume that the result holds for values less than i. The condition $P_i \propto T_j$ implies that

(1) $P_{i-2} \propto T_{j-2}$, $p_{i-1} = t_j$ and $p_i = t_{j-1}$, or
(2) $P_{i-1} \propto T_{j-1}$ and $p_i = t_j$.

If condition (1) holds then, by induction, state q_{i-1} is active after we read characters t_{j-2}. This implies that the transition $\tau(i-1,i)$ is switched on. Since $t_{j-1} = p_i$, after we read character t_{j-1} state q_i is active. Finally, observe that the reactive link $(\tau(i-1,i),\tau(i,i-1))$ is not in T^-. Thus the transition $\tau(i,i-1)$ is switched on before

```
BPSRA(P, m, T, n)
 1.  for c ∈ Σ do
 2.      M[c] ← 0
 3.  for i ← 0 to m − 1 do
 4.      M[p_i] ← M[p_i] | (1 ≪ i)
 5.  F ← 1 ≪ (m − 1)
 6.  A ← 0
 7.  B ← 0^{m−1}1 & M[t_0]
 8.  C ← 0^{m−1}1 & M[t_1]
 9.  for i ← 1 to n − 1 do
10.      H ← (A ≪ 1) | (M ≪ 1) | 1
11.      A ← (C ≪ 1) & M[t_j]
12.      B ← H & M[t_j]
13.      C ← H & M[t_{j+1}]
14.      if ((A | B) & F) then
15.          output(i − m + 1)
```

Figure 4. The Bit Parallel Swap Reactive Automaton Matcher for swap matching.

reading character t_j. Since $t_j = p_{i-1}$, we can conclude that after we read character t_j the state q_{i+1} is active.

Suppose now that condition (2) holds. Then we can state, by induction, that q_i is active after we read charactre t_{j-1}. This implies that the transition $\tau(i, i)$ is switched on. Since $t_j = p_i$, after we read character t_j state q_{i+1} is active. □

The following Theorem 13 trivially follows by Corollary 11 and Lemma 12

Theorem 13 (Correctness). *The swap reactive automaton R recognizes all and only the strings S over Σ such that $P \propto S$.* □

3.2 A Bit Parallel Simulation

In this section we show how to simulate the swap reactive automaton of an input pattern P, as given in Definition 6, by using bit-parallelism [5].

The bit-parallelism technique takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor of at most w, where w is the number of bits in the computer word. It has been extensively used in the field of exact string matching [14] for efficiently simulating non-deterministic automata. However it has also been efficiently used in the field field of multiple pattern matching [10,11,12] and approximate string matching [8,17].

In contrast with standard bit-parallel simulation of non-deterministic automata, where states of the automaton are represented by bits in a bit vector, we use bits to represent transitions of the automaton. In this context an active transition which has been just crossed during the last step is represented by a bit set to 1, while all other transitions are represented by a bit set to 0.

Let $P = p_0p_1p_2 \cdots p_{m-1}$ be a pattern of length m and let $T = t_0t_1t_2 \cdots t_{n-1}$ be a text of length n over Σ. Moreover let $R = (A, T^+, T^-)$ be the SRA of P. The

representation of R uses an array M of σ bit-vectors, each of size m, where the i-th bit of $M[c]$ is set if $p_i = c$.

Automaton configurations are then encoded using 3 bit-vectors of m bits. A bit vector A encodes transitions from q_i to q_{i+1} labeled by character p_{i-1}, a bit vector B encodes transitions from q_i to q_{i+1} labeled by character p_i, while another bit vector C encodes transitions from q_i to q_{i+1} labeled by character p_{i+1}. Specifically we have that the i-th bit of A is set iff (q_i, p_{i-1}, q_{i+1}) is switched on and q_{i+1} is active, the i-th bit of B is set iff (q_i, p_i, q_{i+1}) is switched on and q_{i+1} is active and, finally, the i-th bit of C is set iff (q_i, p_{i+1}, q_{i+1}) is switched on and q_{i+1} is active.

When a search starts, the configurations of the 3 bit-vectors are initialized as $A = 0^m$, $B = (0^{m-1}1 \ \& \ M[t_0])$ and $C = (0^{m-1}1 \ \& \ M[t_1])$.

Then, the text T is scanned, character by character, from left to right and the automaton configuration is updated accordingly. Specifically transitions on character t_j are simulated by performing the following bitwise operations

(i) $A = (C \ll 1) \ \& \ M[t_{j-1}]$
(ii) $B = ((A \ll 1) \mid (B \ll 1) \mid 1) \ \& \ M[t_j]$
(iii) $C = ((A \ll 1) \mid (B \ll 1) \mid 1) \ \& \ M[t_{j+1}]$

Operation (i) ends a swap and indicates that transition $\tau(i, i-1)$ can be crossed only if $\tau(i-1, i)$ has been crossed in the previous step and $t_j = p_{i-1}$. Operations (ii) and (iii) indicates that transitions $\tau(i, i)$ and $\tau(i, i+1)$ can be crossed only if $\tau(i-1, i-1)$ or $\tau(i-1, i-2)$ have been crossed at the previous step and, moreover, $t_j = p_i$ or $t_j = p_{i+1}$, respectively.

The simulation showed above uses 7 bitwise operations for each text character scanned during the searching phase.

After we perform transition on character t_j, state q_m is active if and only if the rightmost bit in A, or in B, is active. Specifically if the test $((A \mid B) \ \& \ 10^{m-1}) \neq 0$ is true, then an occurrence has been found ending at position j of the text.

The resulting algorithm is named Bit Parallel Swap Reactive Automaton Matcher (BPSRA). Its pseudocode is shown in Figure 4.

The preprocessing phase of the BPRSA Matcher (lines 1–5) has a $\mathcal{O}(m+\sigma)$-time complexity. Its searching phase (lines 6–15) has a $\mathcal{O}(n)$-time complexity, if $m \leq w$. When $m > w$ we need to represent the whole automaton by using $3\lceil n/m \rceil$ computer words, so that the worst case time complexity is $\mathcal{O}(n\lceil n/m \rceil)$.

3.3 A More Efficient Simulation

In this section we propose a more efficient simulation of the swap reactive automaton of an input pattern P, by using bit parallelism.

As before, let $P = p_0 p_1 p_2 \cdots p_{m-1}$ and $T = t_0 t_1 t_2 \cdots t_{m-1}$ be two strings of length m over the alphabet Σ. Moreover let $R = (A, T^+, T^-)$ be the SRA of P.

Before entering into details it's convenient to give the following definition of a *string with disjoint triplets*, which we will use in the following discussion.

Definition 14 (String With Disjoint Triplets). *A string $S = s_0 s_1 s_2 \cdots s_{m-1}$, of length m, over an alphabet Σ, is a string with disjoint triplets (SDT) if $s_i \neq s_{i+2}$, for $i = 0, \ldots, m-3$.*

The above definition implies that in the SRA of S the standard transitions from state q_i to q_{i+1}, for $i = 0, \ldots, m-1$, are labeled by different characters.

Text	4	8	16	32
Genome Sequence	0.6080	0.2140	0.0170	0.0010
Protein Sequence	0.8420	0.6160	0.3140	0.1170
English Text	0.9380	0.8440	0.6820	0.4380
Italian Text	0.9130	0.7630	0.5100	0.2500
French Text	0.9230	0.7910	0.5930	0.3250
Chinese Text	0.9860	0.9510	0.8990	0.7750

Table 1. Relative Frequency of SDT in different text buffers.

SDT are very common strings in real data, especially in such cases where the size of the alphabet is large. For instances they are common in natural language texts where it's not common to find words with equal characters with a distance of one character. Table 1 shows the relative frequency of SDT in different text buffers and in particular on a genome sequence, on a protein sequence and on four natural language texts. For each text buffer data have been collected by extracting 10.000 random patterns of different length (ranging from 4 to 32) from the text, and computing the corresponding frequency of SDT.

With the exception of biological sequences, where an SDT is generally uncommon due to the small size of the corresponding alphabet, in the case of natural language texts the percentage of SDT are often over 70 %. In particular it turns out that English and Chinese texts have the largest percentage of SDT.

In the following we propose an efficient simulation of the swap reactive automaton of a pattern P which works properly when P is a SDT. Conversely, if P is not a SDT, then the simulation works as an *oracle*, i.e. it may recognize also strings which are not swap occurrences of the pattern. In this last case an additional naive verification must be performed. The actual advantage in using this new simulation is that it can be performed by only 2 bitwise operations for each iteration of the algorithm. This is a significant improvement compared with previous simulations where at least 7 bitwise operations are needed for each iteration of the algorithm.

In the new proposed simulation the representation of R uses an array B of σ^2 bit-vectors, each of size m, where the i-th bit of $B[c_1, c_2]$ (which we indicate as $B[c_1, c_2]_i$) is defined as

$$B[c_1, c_2]_i = \begin{cases} 1 & \text{if } \tau(i,1), \tau(i+1,2) \in \delta \text{ and } (\tau(i,1), \tau(i+1,2)) \notin T^- \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

for $c_1, c_2 \in \Sigma$, and $0 \leqslant i < m$. Roughly speaking, the matrix M encodes the couples of admissible consecutive transitions in R.

Automaton configurations are then encoded as a bit-vector D of m bits (the initial state does not need to be represented), where the i-th bit of D is set if and only if the state q_{i+1} is active.

When a search starts, the configuration D is initialized to $B[t_0, t_1]$. Then, while the string T is read from left to right, the automaton configuration is updated accordingly for each text character.

Suppose the last transition has been performed on character t_{j-1}, with $0 < j < n-1$, leading to a configuration vector D of the SRA. Then a transition on character

t_j can be implemented by the bitwise operations

$$D^{(j)} = \begin{cases} B[t_0, t_1] & \text{if } j = 1 \\ (D^{(j-1)} \ll 1) \;\&\; B[t_{j-1}, t_j] & \text{otherwise} \end{cases} \tag{2}$$

It turns out that, if P is a SDT, then the simulation of the SRA described above works properly, as stated by the following lemma.

Lemma 15. *Let P be a SDT of length m and let T be a text of length n. Suppose the matrix B is initialized according to* (1) *and suppose to scan the string T, from right to left, and to perform transitions according to* (2)*. After we read character t_j of the text the leftmost bit of $D^{(j)}$ is set if and only if $P \propto T_j$.*

Proof. Let R be the SRA of P. By Theorem 13 we know that R recognizes all and only the prefix of T ending with a swap occurrence of P. We prove that, after we read text character t_j, state q_i is active if and only if the i-th bit in $D^{(j)}$ is set. This will prove the thesis.

We prove it by induction on i. The base case is when $i = 1$. If state q_2 is active after we scan the first two characters of T then one of the following relations holds:

(1) $t_0t_1 = p_0p_1$,
(2) $t_0t_1 = p_1p_0$, or
(3) $t_0t_1 = p_0p_2$.

Observe that if $p_0 = p_1$ only transition $\tau(0,0)$ is in δ, otherwise both $\tau(0,0)$ and $\tau(0,1)$ are in δ. Moreover we have also

- $(\tau(0,0), \tau(1,1)) \notin T^-$, thus the 2nd bit of $B[p_0, p_1]$ is set;
- if $\tau(0,1) \in \delta$ then $(\tau(0,1), \tau(1,0)) \notin T^-$, thus the 2nd bit of $B[p_1, p_0]$ is set;
- $(\tau(0,0), \tau(1,2)) \notin T^-$, thus the 2nd bit of $B[p_0, p_2]$ is set;

Thus after the initialization $D^{(1)} = B[t_0, t_1]$, the second bit of $D^{(1)}$ is set, proving the base case.

Conversely, if the second bit of $B[t_0t_1]$ is set then, by equation 1, it follows that both $\tau(0,0)$ and $\tau(1,1)$ are in δ and $(\tau(0,0), \tau(1,1)) \notin T^-$. Thus state q_2 is active after we scan t_0t_1.

Suppose now that the result holds for values less than i and prove it for i. If state q_i is active after we scan character t_j, then state q_{i-1} is active just before reading character t_{j-1} and, by induction, the $(i-1)$-th bit of $D^{(j-1)}$ is active before reading character t_{j-1}. It follows that both $(q_{i-2}, t_{j-2}, q_{i-1})$ and (q_{i-1}, t_{j-1}, q_i) are in δ and that transition (q_{i-1}, t_{j-1}, q_i) is not switched off when q_{i-1} is active. Thus $((q_{i-2}, t_{j-2}, q_{i-1}), (q_{i-1}, t_{j-1}, q_i))$ is not in T^- and we can conclude that the k-th bit of $B[t_{j-2}, t_{j-1}]$ is set.

Conversely, suppose that the i-th bit of $B[t_{j-2}, t_{j-1}]$ is set. Thus just before reading character t_{j-1} the $(i-1)$-th bit of $D^{(j-1)}$ is set and, by induction, state q_{i-1} is active. It follows that the i-th bit of $B[t_{j-2}, t_{j-1}]$ is set, which implies that both $(q_{i-2}, t_{j-2}, q_{i-1})$ and (q_{i-1}, t_{j-1}, q_i) are in δ and the active link $((q_{i-2}, t_{j-2}, q_{i-1}), (q_{i-1}, t_{j-1}, q_i))$ is not in T^-. We can conclude that after we read character t_{j-1}, state q_i is active. □

The resulting algorithm is named Bit Parallel Swap Reactive Oracle (BPSRO). Its pseudocode is shown in Figure 5. The BPSRO algorithm works as the original Shift-And algorithm [5] for the exact string matching problem. During the preprocessing

```
BPSRO(P, m, T, n)
 1.  for c1, c2 ∈ Σ do B[c1, c2] ← 0
 2.  for i = 1 to m − 1 do
 3.      B[p_{i−1}, p_i] ← B[p_{i−1}, p_i] | (1 ≪ i)
 4.      B[p_i, p_{i−1}] ← B[p_i, p_{i−1}] | (1 ≪ i)
 5.      if (i < m − 1) then
 6.          B[p_{i−1}, p_{i+1}] ← B[p_{i−1}, p_{i+1}] | (1 ≪ i)
 7.      if (i > 1) then
 8.          B[p_{i−2}, p_i] ← B[p_{i−2}, p_i] | (1 ≪ i)
 9.          if (i < m − 1) then
10.              B[p_{i−2}, p_{i+1}] ← B[p_{i−2}, p_{i+1}] | (1 ≪ i)
11.  F ← 1 ≪ (m − 1), D ← 0
12.  for i ← 1 to n − 1 do
13.      D ← ((D ≪ 1) | 1) & B[t_{i−1}, t_i]
14.      if (D & F) then
15.          if (P is a SDT) then output(i − m + 1)
16.          else check occurrence at position (i − m + 1)
```

Figure 5. The Bit Parallel Swap Reactive Oracle Matcher for swap matching.

phase the algorithm computes the matrix B of bit masks. The preprocessing phase (lines 1–11) has a $\mathcal{O}(m + \sigma^2)$-time complexity and requires $\mathcal{O}(\sigma^2)$ space.

During the searching phase the algorithm reads characters of the text, one by one, and simulates transitions on the SRA, accordingly. If the leftmost bit of D is set after we read character t_j, i.e. if $(D\&10^{m-1}) \neq 0$, then an occurrence is reported at position $j - m + 1$. The searching phase (lines 12–16) has a $\mathcal{O}(n)$-time complexity, if $m \leq w$ and P is an SDT. Assuming that P is an SDT, when $m > w$ we need to represent the whole automaton by using $3\lceil n/m \rceil$ computer words, so that the worst case time complexity is $\mathcal{O}(n\lceil n/m \rceil)$. In addition, when P is not an SDT, the algorithm works as an oracle and an additional verification phase is needed in order to check all candidate occurrences. Such a naive verification (line 16) takes $\mathcal{O}(m)$-time, so that the overall worst case time complexity of the algorithm is $\mathcal{O}(nm)$.

4 Experimental Results

In this section we briefly present experimental evaluations in order to understand the performances of the newly presented algorithms.

Specifically we compared the Bit Parallel Swap Reactive Automaton algorithm (BPSRA) and the Bit Parallel Swap Reactive Oracle algorithm (BPSRO) against the Bit Parallel Cross Sampling algorithm (BPCS) [9,6], which is one of the most efficient linear algorithm present in literature. Other practical algorithms are known in literature [7] for the swap matching problem, which show a sub-linear behavior in practical cases, however they use a backward scan of the text, an efficient technique which can be applied to almost all automata based algorithm (including ours) and which is out of the scope of the present paper.

All algorithms have been implemented in the C programming language and have been tested using the SMART tool[1], which have been provided for testing exact string

[1] The SMART tool is available online at: `http://www.dmi.unict.it/~faro/smart/`

m	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
BPCS	16.0	15.9	15.9	16.0	15.9	15.9	15.9	16.1	16.1	16.2	16.0	15.9	16.1	16.3	16.0
BPSRA	**15.4**	15.3	15.3	15.2	15.2	15.3	15.8	15.4	15.3	15.3	15.3	15.4	15.3	15.3	15.3
BPSRO	20.4	**13.7**	**11.4**	**11.2**	**11.2**	**12.0**	**11.2**	**11.4**	**11.3**	**11.3**	**12.8**	**11.5**	**11.5**	**11.3**	**11.3**
	(A) genome sequence					(B) protein sequence					(C) natural language text				

Table 2. Experimental results on (A) a genome sequence, (B) a protein sequence and (C) a natural language text.

matching algorithm [14] but allows to be adapted also for approximate string matching algorithms. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. Algorithms have been compared in terms of running times, including any preprocessing time.

For the evaluation we use a genome sequence, a protein sequence and a natural language text (English language), all sequences of 4MB. The sequences are provided by the Smart research tool. In all cases the patterns were randomly extracted from the text and the value m was made ranging from 2 to 32. For each case we reported the mean over the running times of 500 runs.

Table 4 shows experimental results on the three different sequences. Running times are expressed in thousands of seconds. Best times have been boldfaced.

From the experimental results it turns out that the BPSRA algorithm has almost the same performance of the BPCS algorithm, but is slightly better in all practical cases. Both the BPCS and the BPSRA show a linear behavior. The BPSRO shows instead a decreasing trend, which is much evident for the case of short patterns. This behavior is due to the presence of a larger number of verification tests which must be run when the pattern is short and is not an SNR. It turns out moreover that in almost all cases the BPSO is faster than the other algorithms, due to its less complex simulation engine.

5 Conclusions

In this paper we have presented a new theoretical approach to solve the swap matching problem in strings. The new approach uses a reactive automaton with only m states and (at most) $3m-2$ transitions. We propose also two different approaches to simulate the automaton by using bit-parallelism.

As in the case of the Cross Sampling algorithm the new proposed approach can be extended to obtain more efficient solution by scanning the text from right to left. Our future works will consider such improvements.

References

1. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Pattern matching with swaps*, in IEEE Symp. on Foundations of Computer Science, 1997, pp. 144–153.
2. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Pattern matching with swaps*. Journal of Algorithms, 37(2) 2000, pp. 247–266.
3. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat: *Overlap matching*. Inf. Comput., 181(1) 2003, pp. 57–74.
4. A. Amir, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Efficient special cases of pattern matching with swaps*. Inf. Proc. Letters, 68(3) 1998, pp. 125–132.

5. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
6. M. Campanelli, D. Cantone, and S. Faro: *A new algorithm for efficient pattern matching with swaps*, in IWOCA 2009: 20th International Workshop on Combinatorial Algorithms, vol. 5874 of Lecture Notes in Computer Science, Springer, 2009, pp. 230–241.
7. M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta: *Pattern matching with swaps in practice.* International Journal of Foundation of Computer Science, 23(2) 2012, pp. 323–342.
8. D. Cantone, S. Cristofaro, and S. Faro: *Efficient string-matching allowing for non-overlapping inversions.* Theor. Comput. Sci., 483 2013, pp. 85–95.
9. D. Cantone and S. Faro: *Pattern matching with swaps for short patterns in linear time*, in SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5404 of Lecture Notes in Computer Science, Springer, 2009, pp. 255–266.
10. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach*, in Combinatorial Pattern Matching, vol. 6129 of Lecture Notes in Computer Science, 2010, pp. 288–298.
11. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.
12. D. Cantone, S. Faro, and E. Giaquinta: *On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns.* J. Discrete Algorithms, 11 2012, pp. 25–36.
13. M. Crochemore and D. M. Gabbay: *Reactive automata.* Inf. Comput., 209(4) Apr. 2011, pp. 692–704.
14. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, p. 13.
15. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
16. D. M. Gabbay: *Pillars of computer science*, Springer-Verlag, 2008, ch. Introducing reactive Kripke semantics and arc accessibility, pp. 292–341.
17. S. Grabowski, S. Faro, and E. Giaquinta: *String matching with inversions and translocations in linear average time (most of the time).* Inf. Process. Lett., 111(11) 2011, pp. 516–520.
18. C. S. Iliopoulos and M. S. Rahman: *A new model to solve the swap matching problem and efficient algorithms for short patterns*, in SOFSEM 2008, vol. 4910 of Lecture Notes in Computer Science, Springer, 2008, pp. 316–327.
19. S. Muthukrishnan: *New results and open problems related to non-standard stringology*, in Combinatorial Pattern Matching, CPM 95, vol. 937 of Lecture Notes in Computer Science, Springer, 1995, pp. 298–317.