International Journal of Foundations of Computer Science
© World Scientific Publishing Company

PATTERN MATCHING WITH SWAPS IN PRACTICE*

MATTEO CAMPANELLI†
Università di Catania, Scuola Superiore di Catania
Via San Nullo 5/i, I-95123 Catania, Italy

DOMENICO CANTONE‡
SIMONE FARO§
EMANUELE GIAQUINTA¶
Dipartimento di Matematica e Informatica, Università di Catania
Viale A.Doria n.6, I-95125 Catania, Italy

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 the Approximate Pattern Matching problem with Swaps one seeks, for every text location with a swapped match of P, the number of swaps necessary to obtain a match at the location.

In this paper we devise two general algorithms for both Standard and Approximate Pattern Matching with Swaps, named Cross-Sampling and Backward-Cross-Sampling, with a $\mathcal{O}(nm)$ and $\mathcal{O}(nm^2)$ worst-case time complexity, respectively. Then we provide efficient implementations of them, based on bit-parallelism, which achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ worst-case time complexity, with patterns whose length is comparable to the word-size of the target machine.

From an extensive comparison with some of the most effective algorithms for the swap matching problem, it turns out that our algorithms are very flexible and achieve very good results in practice.

Keywords: approximate pattern matching with swaps; nonstandard pattern matching; combinatorial algorithms on words; design and analysis of algorithms.

1. Introduction

The *Pattern Matching problem with Swaps* (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 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

*The paper presents results appeared in a preliminary form in [10], [8], and [9].
†macampanelli@ssc.unict.it
‡cantone@dmi.unict.it
§faro@dmi.unict.it
¶giaquinta@dmi.unict.it

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 [6], 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.

The swap matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [13]. 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 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. We observe that the above solutions are all based on the fast Fourier transform (FFT) technique.

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 [12]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on the bit-parallelism technique [7], which runs in $\mathcal{O}((n + m) \log m)$-time, provided that the pattern size is comparable to the word size in the target machine.

The *Approximate Pattern Matching problem with Swaps* seeks to compute, for each text location j, the number of swaps necessary to convert the pattern to the substring of length m ending at j.

A straightforward solution to the approximate swap matching problem consists in searching for all occurrences (with swap) of the input pattern P, using any algorithm for the standard swap matching problem. Once a swap match is found, to get the number of swaps, it is sufficient to count the number of mismatches between the pattern and its swap occurrence in the text and then divide it by 2.

In [5], Amir *et al.* presented an algorithm that counts in time $\mathcal{O}(\log m \log \sigma)$ the number of swaps at every location containing a swapped matching, thus solving the approximate pattern matching problem with swaps in $\mathcal{O}(n \log m \log \sigma)$-time.

In this paper, we present two algorithms for the Swap Matching problem and the Approximate Swap Matching problem, having linear worst-case time complexity for short patterns. More precisely, the two algorithms presented, named Cross-Sampling and Backward-Cross-Sampling, have a $\mathcal{O}(nm)$ and $\mathcal{O}(nm^2)$ worst-case time complexity, respectively. We also show how to obtain efficient implementations of them, based on bit-parallelism, which achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ worst-case time, respectively, and $\mathcal{O}(\sigma)$-space complexity for patterns having length similar to the word-size of the target machine.

The rest of the paper is organized as follows. In Section 2 we recall some preliminary definitions. In Section 3 we present the Cross-Sampling algorithm for the swap matching and approximate swap matching problem and apply the bit-parallelism technique to obtain efficient implementations. Then in Section 4 we present the Backward-Cross-Sampling algorithm for the swap matching problem and illustrate efficient implementations based on bit-parallelism. In Section 6, we compare our newly proposed algorithms against the most effective algorithms present in literature and, finally, we briefly draw our conclusions in Section 7.

2. Notions and Basic Definitions

Given a string P of length $m \geq 0$, we represent it as a finite array $P[0\,..\,m-1]$ and write $\text{length}(P) = m$. In particular, for $m = 0$ we obtain the empty string ε. We denote by $P[i]$ the $(i+1)$-st character of P, for $0 \leq i < \text{length}(P)$, and by $P[i\,..\,j]$ the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < \text{length}(P)$. A k-substring of a string S is a substring of S of length k. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i\,..\,\text{length}(P)-1]$, for some $0 \leq i < \text{length}(P)$. Similarly, we say that P' is a prefix of P if $P' = P[0\,..\,i-1]$, for some $0 \leq i \leq \text{length}(P)$. We denote by P_i the nonempty prefix $P[0\,..\,i]$ of P of length $i+1$, for $0 \leq i < m$, whereas, if $i < 0$, we agree that P_i is the empty string ε. Moreover, we say that P' is a proper prefix (suffix) of P if P' is a prefix (suffix) of P and $|P'| < |P|$. Finally, we write $P.P'$ to denote the concatenation of P and P'.

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 $\pi(P) = P[\pi(0)].P[\pi(1)].\cdots.P[\pi(m-1)]$.

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 $P \propto T_j$.*

As already observed, if a pattern P of length m has a swap match ending at location j of a text T, then the number k of swaps needed to transform P into its swapped version $\pi(P) = T[j-m+1\,..\,j]$ is equal to half the number of mismatches of P at location j. Thus the value of k lies between 0 and $\lfloor m/2 \rfloor$.

Definition 3. *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 of T with k swaps if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j and $k = |\{i : P[i] \neq P[\pi(i)]\}|/2$. In such a case we write $P \propto_k T_j$.*

Definition 4 (Pattern Matching Problem with Swaps) *Given a text T of length n and a pattern P of length m, find all locations $j \in \{m-1, ..., n-1\}$ such that P swap-matches with T at location j, i.e., $P \propto T_j$.*

Definition 5 (Approximate Pattern Matching Problem with Swaps) *Given a text T of length n and a pattern P of length m, find all pairs (j, k), with $j \in \{m-1...n-1\}$ and $0 \leq k \leq \lfloor m/2 \rfloor$, such that P has a swapped occurrence in T at location j with k swaps, i.e., $P \propto_k T_j$.*

The following elementary result will be used later (its proof is given in [10]).

Lemma 6 ([10]) *Let P and R be strings of length m over an alphabet Σ and suppose that there exists a swap permutation π such that $\pi(P) = R$. Then π is unique.*

Corollary 7. *Given a text T of length n and a pattern P of length m, if $P \propto T_j$, for a given position $j \in \{m-1, \ldots, n-1\}$, then there exists a unique swapped occurrence of P in T ending at position j.* ■

3. Cross-Sampling Algorithms

In this section we present a new algorithm for solving the swap matching problem. Our algorithm is characterized by a $\mathcal{O}(mn)$-time and a $\mathcal{O}(m)$-space complexity, where m and n are the lengths of the pattern and text, respectively. We will also show how to extend our algorithm to the case of the approximate swap matching problem, maintaining the same time and space complexity.

3.1. *Standard Swap Matching Problem*

As above, let T be a text of length n and let P be a pattern of length m. Our algorithm solves the swap matching problem by computing the swap occurrences of all prefixes of the pattern in continuously increasing prefixes of the text using a dynamic programming approach. That is, during its $(j+1)$-st iteration, for $j = 0, 1, \ldots, n-1$, our algorithm establishes whether $P_i \propto T_j$, for each $i = 0, 1, \ldots, m-1$, exploiting information gathered during previous iterations.

Each row of the matrix is labeled with a prefix $\overline{P}_i$ of the pattern, whereas columns are labeled with the locations of the text $\overline{T}$. A value k in row $\overline{P}_i$ and column j means that $\overline{P}_i \propto_k \overline{T}_j$, whereas a symbol - means that $\overline{P}_i \not\propto \overline{T}_j$. Thus, to look for the swap occurrences of $\overline{P}$ in $\overline{T}$ it is enough to inspect the row labelled $\overline{P}_6$, as $\overline{P} = \overline{P}_6$: by doing so, we find that there is a swapped occurrence of $\overline{P}$ at location 9 of $\overline{T}$.

More generally, if we denote by $\mathcal{S}_j$ the set of the integral values in the j-th column of the $m \times n$ matrix of the swap occurrences of the prefixes for our generic pattern P and text T, where $j = 0, 1, \dots, n-1$, we plainly have that P has a swapped occurrence at location j of T if and only if $(m-1) \in \mathcal{S}_j$. In fact, by definition, we have $\mathcal{S}_j = \{0 \le i \le m-1 \mid \; P_i \propto T_j\}$.

The sets $\mathcal{S}_j$ can be computed efficiently by a dynamic programming algorithm, by exploiting the following very elementary property.

Lemma 8. *Let T and P be a text of length n and a pattern of length m, respectively. Then, for each $0 \le j < n$ and $0 \le i < m$, we have that $P_i \propto T_j$ if and only if one of the following two facts holds*

- $P[i] = T[j]$ *and* $P_{i-1} \propto T_{j-1}$*;*
- $P[i] = T[j-1]$*,* $P[i-1] = T[j]$*, and* $P_{i-2} \propto T_{j-2}$*.* ■

To this end, let us denote by $\mathcal{S}'_j$, for $0 \le j < n-1$, the collection of all values i such that the prefix P_{i-1} of P has a swapped occurrence ending at location $j-1$ of the text T and $P[i] = T[j+1]$. Or, more formally, let

$$\mathcal{S}'_j = \{i \mid P_{i-1} \propto T_{j-1} \;\text{ and }\; P[i] = T[j+1]\} \text{ and } \lambda_j = \begin{cases} \{0\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise .} \end{cases}$$

Then, the base case is given by

$$\mathcal{S}_0 = \lambda_0, \quad \text{and} \quad \mathcal{S}'_0 = \lambda_1. \tag{1}$$

Additionally, Lemma 8 justifies the following recursive definitions of the sets $\mathcal{S}_{j+1}$ and $\mathcal{S}'_{j+1}$ in terms of $\mathcal{S}_j$ and $\mathcal{S}'_j$, for $0 \le j < n-1$:

$$\begin{aligned} \mathcal{S}_{j+1} &= \{i \le m-1 \mid \; ((i-1) \in \mathcal{S}_j \;\text{ and }\; P[i] = T[j+1]) \text{ or} \\ &\qquad\qquad\qquad\; ((i-1) \in \mathcal{S}'_j \;\text{ and }\; P[i] = T[j])\,\} \cup \lambda_{j+1} \\ \mathcal{S}'_{j+1} &= \{i < m-1 \mid \; (i-1) \in \mathcal{S}_j \;\text{ and }\; P[i] = T[j+2]\} \cup \lambda_{j+2}\,. \end{aligned} \tag{2}$$

Such relations, coupled with the initial conditions (1), allow one to compute the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$ in an iterative fashion.Observe that $\mathcal{S}_{j+1}$ is computed in terms of both $\mathcal{S}_j$ and $\mathcal{S}'_j$, whereas $\mathcal{S}'_{j+1}$ needs only $\mathcal{S}_j$ for its computation. The resulting dependency graph has a doubly crossed structure, from which the name of the algorithm of Fig. 1, Cross-Sampling, for the swap matching problem.

To compute the worst-case time complexity of the Cross-Sampling algorithm, first observe that the for-cycle of line 4 is executed $\mathcal{O}(n)$ times. During the j-th iteration, the for-cycles of line 6 and line 12 are executed $|\mathcal{S}_j|$ and $|\mathcal{S}'_j|$ times, respectively. However, according to Lemma 6, for each position j of the text we can report only a single swapped occurrence of the prefix P_i in T_j, for each $0 \le i < m$, which implies $|\mathcal{S}_j| \le m$ and $|\mathcal{S}'_j| < m$. Thus the time complexity of the resulting algorithm is $\mathcal{O}(nm)$.

```
(A) Cross-Sampling $(P, m, T, n)$
1.  $\mathcal{S}_0 \leftarrow \mathcal{S}'_0 \leftarrow \emptyset$
2.  if $P[0] = T[0]$ then $\mathcal{S}_0 \leftarrow \{0\}$
3.  if $P[0] = T[1]$ then $\mathcal{S}'_0 \leftarrow \{0\}$
4.  for $j \leftarrow 1$ to $n - 1$ do
5.     $\mathcal{S}_j \leftarrow \mathcal{S}'_j \leftarrow \emptyset$
6.     for $i \in \mathcal{S}_{j-1}$ do
7.        if $i < m - 1$ then
8.           if $P[i+1] = T[j]$
9.              then $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{i+1\}$
10.          if $j < n - 1$ and $P[i+1] = T[j+1]$
11.             then $\mathcal{S}'_j \leftarrow \mathcal{S}'_j \cup \{i+1\}$
12.    for $i \in \mathcal{S}'_{j-1}$ do
13.       if $i < m - 1$ and $P[i+1] = T[j-1]$
14.          then $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{i+1\}$
15.    if $P[0] = T[j]$ then $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{0\}$
16.    if $j < n - 1$ and $P[0] = T[j+1]$
17.       then $\mathcal{S}'_j \leftarrow \mathcal{S}'_j \cup \{0\}$
18.    if $m - 1 \in \mathcal{S}_j$ then Output($j$)
```

```
(B) Approximate-Cross-Sampling $(P, m, T, n)$
1.  $\bar{\mathcal{S}}_0 \leftarrow \bar{\mathcal{S}}'_0 \leftarrow \emptyset$
2.  if $P[0] = T[0]$ then $\bar{\mathcal{S}}_0 \leftarrow \{(0,0)\}$
3.  if $P[0] = T[1]$ then $\bar{\mathcal{S}}'_0 \leftarrow \{(0,0)\}$
4.  for $j \leftarrow 1$ to $n - 1$ do
5.     $\bar{\mathcal{S}}_j \leftarrow \bar{\mathcal{S}}'_j \leftarrow \emptyset$
6.     for $(i,k) \in \bar{\mathcal{S}}_{j-1}$ do
7.        if $i < m - 1$ then
8.           if $P[i+1] = T[j]$
9.              then $\bar{\mathcal{S}}_j \leftarrow \bar{\mathcal{S}}_j \cup \{(i+1,k)\}$
10.          if $j < n - 1$ and $P[i+1] = T[j+1]$
11.             then $\bar{\mathcal{S}}'_j \leftarrow \bar{\mathcal{S}}'_j \cup \{(i+1,k)\}$
12.    for $(i,k) \in \bar{\mathcal{S}}'_{j-1}$ do
13.       if $i < m - 1$ and $P[i+1] = T[j-1]$
14.          then $\bar{\mathcal{S}}_j \leftarrow \bar{\mathcal{S}}_j \cup \{(i+1,k+1)\}$
15.    if $P[0] = T[j]$ then $\bar{\mathcal{S}}_j \leftarrow \bar{\mathcal{S}}_j \cup \{(0,0)\}$
16.    if $j < n - 1$ and $P[0] = T[j+1]$
17.       then $\bar{\mathcal{S}}'_j \leftarrow \bar{\mathcal{S}}'_j \cup \{(0,0)\}$
18.    for $(i,k) \in \bar{\mathcal{S}}_j$ do
19.       if $i = m - 1$ then Output($j, k$)
```

Fig. 1. **(A)** The Cross-Sampling algorithm for solving the swap matching problem. **(B)** The Approximate-Cross-Sampling algorithm for solving the approximate swap matching problem.

3.2. *Approximate Swap Matching Problem*

Next we show how to extend the Cross-Sampling algorithm to solve the approximate swap matching problem. To begin with, we extend Lemma 8 as follows.

Lemma 9. *Let T and P be a text of length n and a pattern of length m, respectively. Then, for each $0 \leq i < n$ and $0 \leq k < m$, we have that $P_i \propto_k T_j$ if and only if one of the following two facts hold*
- $P[i] = T[j]$ and either $(i = 0 \wedge k = 0)$ or $P_{i-1} \propto_k T_{j-1}$;
- $P[i] = T[j-1]$, $P[i-1] = T[j]$, and either $(i = 1 \wedge k = 1)$ or $P_{i-2} \propto_{k-1} T_{j-2}$. ■

Next we define the sets $\overline{\mathcal{S}}_j$ and $\overline{\mathcal{S}}'_j$ which denote, respectively, the collection of all pairs (i,k) such that the prefix P_i of P has a k-swapped occurrence ending at position j of the text and the collection of all pairs (i,k) such that the prefix P_{i-1} of P has a k-swapped occurrence ending at position $j-1$ of the text and $P[i] = T[j+1]$, for $0 \leq j \leq n$. More formally,

$$\bar{\mathcal{S}}_j = \{(i,k) \mid 0 \leq i \leq m-1 \text{ and } P_i \propto_k T_j\}$$
$$\bar{\mathcal{S}}'_j = \{(i,k) \mid 0 \leq i < m-1 \text{ and } (P_{i-1} \propto_k T_{j-1} \vee i = 0) \text{ and } P[i] = T[j+1]\}.$$

In view of such definitions, the approximate swap matching problem translates into the problem of finding all pairs (j,k) such that $(m-1,k) \in \bar{\mathcal{S}}_j$, where $0 \leq k < \lfloor m/2 \rfloor$.

Sets $\bar{\mathcal{S}}_0$ and $\bar{\mathcal{S}}'_0$ can be defined as follows

$$\bar{\lambda}_j = \begin{cases} \{(0,0)\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise} \end{cases}, \quad \bar{\mathcal{S}}_0 = \bar{\lambda}_0, \quad \text{and} \quad \bar{\mathcal{S}}'_0 = \bar{\lambda}_1.$$

Lemma 9 justifies the following recursive definition of the sets $\bar{\mathcal{S}}_{j+1}$ and $\bar{\mathcal{S}}'_{j+1}$ in terms of $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$, for $j < n$:

$$\begin{aligned}
\bar{\mathcal{S}}_{j+1} = \{(i,k) \mid i \leq m-1 \text{ and } & \quad ((i-1,k) \in \bar{\mathcal{S}}_j \text{ and } P[i] = T[j+1]) \text{ or} \\
& \quad ((i-1,k-1) \in \bar{\mathcal{S}}'_j \text{ and } P[i] = T[j]) \ \} \cup \bar{\lambda}_{j+1} \\
\bar{\mathcal{S}}'_{j+1} = \{(i,k) \mid i < m-1 \text{ and } & \quad (i-1,k) \in \bar{\mathcal{S}}_j \text{ and } P[i] = T[j+2]\} \cup \bar{\lambda}_{j+2}.
\end{aligned}$$

Fig. 1(B) shows the APPROXIMATE-CROSS-SAMPLING algorithm for solving the approximate swap matching problem. It can easily be checked that its worst-case time complexity is $\mathcal{O}(nm)$.

3.3. *Bit-Parallel Implementation*

In this section we present two efficient algorithms based on the bit-parallelism techique [7] to search for swapped occurrences of short patterns in texts. 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.

The simulation of the CROSS-SAMPLING algorithm with bit-parallelism is performed by representing the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$ as lists of m bits, D_j and D'_j respectively, where m is the length of the pattern. The i-th bit of D_j is set to 1 if $i \in \mathcal{S}_j$, i.e. if $P_i \propto T_j$, whereas the i-th bit of D'_j is set to 1 if $i \in \mathcal{S}'_j$, i.e. if $P_{i-1} \propto T_{j-1}$ and $P[i] = T[j+1]$. All remaining bits in the bit vectors are set to 0. Note that if $m \leq w$, the entire list fits in a single computer word, whereas if $m > w$ we need $\lceil m/w \rceil$ computer words to represent the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$.

For each character c of the alphabet Σ, the algorithm maintains a bit mask $M[c]$, where the i-th bit is set to 1 if $P[i] = c$.

The bit vectors D_0 and D'_0 are initialized to 0^m. Then the algorithm scans the text from the first character to the last one and, for each position $j \geq 0$, it computes the bit vector D_j in terms of D_{j-1} and D'_{j-1}, by performing the following bitwise operations:

$$\begin{array}{ll}
D_j \leftarrow D_{j-1} \ll 1 & \mathcal{S}_j = \{i : (i-1) \in \mathcal{S}_{j-1}\} \\
D_j \leftarrow D_j \mid 1 & \mathcal{S}_j = \mathcal{S}_j \cup \{0\} \\
D_j \leftarrow D_j \ \&\ M[T[j]] & \mathcal{S}_j = \mathcal{S}_j \setminus \{i : P[i] \neq T[j]\} \\
D_j \leftarrow D_j \mid H^1 & \mathcal{S}_j = \mathcal{S}_j \cup \{i : (i-1) \in \mathcal{S}'_{j-1} \wedge P[i] = T[j-1]\},
\end{array}$$

where $H^1 = \big((D'_{j-1} \ll 1) \ \&\ M[T[j-1]]\big)$.

Similarly, the bit vector D'_j is computed during the j-th iteration of the algorithm in terms of D_{j-1}, by performing the following bitwise operations:

$$\begin{array}{ll}
D'_j \leftarrow D_{j-1} \ll 1 & \mathcal{S}'_j = \{i : (i-1) \in \mathcal{S}_{j-1}\} \\
D'_j \leftarrow D'_j \mid 1 & \mathcal{S}'_j = \mathcal{S}'_j \cup \{0\} \\
D'_j \leftarrow D'_j \ \&\ M[T[j+1]] & \mathcal{S}'_j = \mathcal{S}'_j \setminus \{i : P[i] \neq T[j+1]\}.
\end{array}$$

```
(A) BP-Cross-Sampling $(P, m, T, n)$
$F \leftarrow 0^{m-1}1$
for $c \in \Sigma$ do $M[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m-1$ do
    $M[P[i]] \leftarrow M[P[i]] \mid F$
    $F \leftarrow F \ll 1$
$F \leftarrow 10^{m-1}$
$D \leftarrow D' \leftarrow 0^m$
for $j \leftarrow 0$ to $n-1$ do
    $H \leftarrow (D \ll 1) \mid 1$
    $D \leftarrow (H \ \& \ M[T[j]])$
    $D' \leftarrow (D' \ll 1) \ \& \ M[T[j-1]]$
    $D \leftarrow D \mid D'$
    $D' \leftarrow H \ \& \ M[T[j+1]]$
    if $(D \ \& \ F) \neq 0^m$ then
        Output$(j)$
```

```
(B)BP-Approximate-Cross-Sampling $(P, m, T, n)$
$q \leftarrow \log(\lfloor m/2 \rfloor + 1) + 1$
$F \leftarrow 0^{qm-1}1$
$G \leftarrow 0^{q(m-1)}1^q$
for $c \in \Sigma$ do
    $M[c] \leftarrow 0^{qm}$
    $B[c] \leftarrow 0^{qm}$
for $i \leftarrow 0$ to $m-1$ do
    $M[P[i]] \leftarrow M[P[i]] \mid F$
    $B[P[i]] \leftarrow B[P[i]] \mid G$
    $F \leftarrow (F \ll q)$
    $G \leftarrow (G \ll q)$
$F \leftarrow 0^{q-1}10^{q(m-1)}$
$\bar{D} \leftarrow \bar{D}' \leftarrow 0^{qm}$
for $j \leftarrow 0$ to $n-1$ do
    $H^0 \leftarrow (\bar{D} \ll q) \mid 1$
    $H^1 \leftarrow (\bar{D}' \ll q) \ \& \ B[T[j-1]$
    $H^2 \leftarrow (\bar{D}' \ll q) \ \& \ M[T[j-1]$
    $\bar{D} \leftarrow (H^0 \ \& \ B[T[j]]) \mid H^1$
    $\bar{D} \leftarrow \bar{D} + (H^2 \ll 1)$
    $\bar{D}' \leftarrow (H^0 \ \& \ B[T[j+1]]) \ \& \ \sim \bar{D}$
    if $(\bar{D} \ \& \ F) \neq 0^{qm}$ then
        $k \leftarrow (\bar{D} \gg (q(m-1)+1))$
        Output$(j, k)$
```

Fig. 2. **(A)** The BP-Cross-Sampling algorithm, which solves the swap matching problem in linear time by using bit-parallelism. **(B)** The BP-Approximate-Cross-Sampling algorithm, which solves the approximate swap matching problem in linear time by using bit-parallelism.

During the j-th iteration, we report a swap match at position j, provided that the leftmost bit of D_j is set to 1, i.e., if $(D_j \ \& \ 10^{m-1}) \neq 0^m$.

In practice, we can use only two vectors to maintain D_j and D'_j, for $j = 0, \ldots, n-1$. Thus during iteration j of the algorithm, vector D_{j-1} is transformed into vector D_j, whereas vector D'_{j-1} is transformed into vector D'_j. The resulting BP-Cross-Sampling algorithm is shown in Fig. 2(A). It achieves a $\mathcal{O}(\lceil mn/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m/w \rceil)$ extra-space. If $m \leq w$, then the algorithm reports all swapped matches in $\mathcal{O}(n)$-time and $\mathcal{O}(\sigma)$ extra space.

The simulation of the Approximate-Cross-Sampling algorithm can be performed by representing the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ as a list of q bits, $\bar{D}_j$ and $\bar{D}'_j$ respectively, where $q = \log(\lfloor m/2 \rfloor + 1) + 1$ and m is the length of the pattern. If the pair $(i, k) \in \bar{\mathcal{S}}_j$, for $0 \leq i < m$ and $0 \leq k \leq \lfloor m/2 \rfloor$, then the rightmost bit of the i-th block of $\bar{D}_j$ is set to 1 and the leftmost $q-1$ bits of the i-th block are set so as to contain the value k.[a] Otherwise, if the pair (i, k) does not belong to $\bar{\mathcal{S}}_j$, then the rightmost bit of the i-th block of $\bar{D}_j$ is set to 0. In a similar way we can maintain the current configuration of the set $\bar{\mathcal{S}}'_j$. If $m \log(\lfloor m/2 \rfloor + 1) + m \leq w$, then the entire list fits in a single computer word, otherwise we need $\lceil m(\log(\lfloor m/2 \rfloor + 1)/w \rceil$ computer words to represent the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$.

[a]We need exactly $\log(\lfloor m/2 \rfloor + 1)$ bits to represent a value between 0 and $\lfloor m/2 \rfloor$.

For each character c of the alphabet Σ the algorithm maintains a bit mask $M[c]$, where the rightmost bit of the i-th block is set to $\mathsf{1}$ if $P[i] = c$. Moreover, for each character $c \in \Sigma$, the algorithm maintains, a bit mask $B[c]$ whose i-th block have all bits set to $\mathsf{1}$ if $P[i] = c$, whereas all remaining bits are set to $\mathsf{0}$.

Then each block is made up by q bits, with $q = \log(\lfloor 7/2 \rfloor + 1) + 1 = 3$. The leftmost two bits of each block contain the number of swaps k, where $0 \leq k \leq 3$.

Before entering into details, we observe that if $(i,k) \in \bar{\mathcal{S}}_j$ and $(i,k) \in \bar{\mathcal{S}}'_j$ then we can conclude that $T[j] = T[j+1]$. Moreover, if $T[j+1] = P[i+1]$ we have also $T[j] = P[i+1]$, which would imply a swap between two identical characters of the pattern. Since the latter condition would violate Definition 1(c), during the computation of vectors $\bar{D}_j$ and $\bar{D}'_j$ we maintain the following invariant

$$\text{if the } i\text{-th bit of } \bar{D}_j \text{ is set to } \mathsf{1}\text{, then the } i\text{-th bit of } \bar{D}'_j \text{ is set to } \mathsf{0}. \qquad (3)$$

Initially, all bit vectors are set to $\mathsf{0}^{qm}$. Then the algorithm scans the text from the first character to the last one and, for each position $j \geq 0$, it computes the vector $\bar{D}_j$ in terms of $\bar{D}_{j-1}$ and $\bar{D}'_{j-1}$, by performing the following bitwise operations:

$$\begin{array}{ll}
\bar{D}_j \leftarrow \bar{D}_{j-1} \ll q & \bar{\mathcal{S}}_j = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}_{j-1}\} \\
\bar{D}_j \leftarrow \bar{D}_j \mid 1 & \bar{\mathcal{S}}_j = \bar{\mathcal{S}}_j \cup \{(0,0)\} \\
\bar{D}_j \leftarrow \bar{D}_j \ \& \ B[T[j]] & \bar{\mathcal{S}}_j = \bar{\mathcal{S}}_j \setminus \{(i,k) : P[i] \neq T[j]\} \\
\bar{D}_j \leftarrow \bar{D}_j \mid H^1 & \bar{\mathcal{S}}_j = \bar{\mathcal{S}}_j \cup K \\
\bar{D}_j \leftarrow \bar{D}_j + (H^2 \ll 1) & \forall\ (i,k) \in K \text{ change } (i,k) \text{ with } (i,k+1) \text{ in } \bar{\mathcal{S}}_j,
\end{array}$$

where we have set

$$\begin{array}{l}
H^1 = ((\bar{D}'_{j-1} \ll q) \ \& \ B[T[j-1]]) \\
H^2 = ((\bar{D}'_{j-1} \ll q) \ \& \ M[T[j-1]]), \text{ and} \\
K \ = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}'_{j-1} \wedge P[i] = T[j-1]\} \,.
\end{array}$$

Similarly, $\bar{D}'_j$ is computed by performing the following bitwise operations:

$$\begin{array}{ll}
\bar{D}'_j \leftarrow \bar{D}_{j-1} \ll q & \bar{\mathcal{S}}'_j = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}_{j-1}\} \\
\bar{D}'_j \leftarrow \bar{D}'_j \mid 1 & \bar{\mathcal{S}}'_j = \bar{\mathcal{S}}'_j \cup \{(0,0)\} \\
\bar{D}'_j \leftarrow \bar{D}'_j \ \& \ B[T[j+1]] & \bar{\mathcal{S}}'_j = \bar{\mathcal{S}}'_j \setminus \{(i,k) : P[i] \neq T[j+1]\} \\
\bar{D}'_j \leftarrow \bar{D}'_j \ \& \ \sim \bar{D}_j & \bar{\mathcal{S}}'_j = \bar{\mathcal{S}}'_j \setminus \{(i,k) : (i,k) \in \bar{\mathcal{S}}_j\}.
\end{array}$$

During the j-th iteration of the algorithm, if the rightmost bit of the $(m-1)$-st block of $\bar{D}_j$ is set to $\mathsf{1}$, i.e. if $(\bar{D}_j \ \& \ \mathsf{1}\mathsf{0}^{q(m-1)}) \neq \mathsf{0}^m$, we report a swap match at position j. Additionally, the number of swaps needed to transform the pattern to its swapped occurrence in the text is contained in the $q-1$ leftmost bits of the $(m-1)$-st block of $\bar{D}_j$ which can be extracted by performing a bitwise shift of $(q(m-1)+1)$ positions to the right.

As in the case of the BP-Cross-Sampling algorithm, in practice we can use only two vectors to maintain $\bar{D}_j$ and $\bar{D}'_j$, for $j = 0, \ldots, n-1$. The BP-Approximate-Cross-Sampling algorithm, shown in Fig. 2(B), achieves $\mathcal{O}(\lceil (mn \log m)/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m \log m/w \rceil)$ extra-space. If the length of the pattern is such that $m(\log(\lfloor m/2 \rfloor + 1) + 1) \leq w$, then the algorithm reports all swapped matches in $\mathcal{O}(n)$ time and $\mathcal{O}(\sigma)$ extra space.

4. Backward-Cross-Sampling algorithms

In this section we present a practical algorithm for solving the swap matching problem, called Backward-Cross-Sampling, which though characterized by a $\mathcal{O}(mn^2)$-time complexity in practice is faster than the Cross-Sampling algorithm.

In Section 4.2 we extend the Backward-Cross-Sampling algorithm in order to solve the Approximate Pattern Matching Problem with Swaps, preserving the same time and space complexities. Later, in Section 4.3, we present efficient implementations of the algorithms based on bit parallelism, which achieve a $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity, when the pattern fits within few computer words, i.e., if $m \leq c_1 w$, for some small constant c_1.

4.1. *Standard Swap Matching Problem*

The Backward-Cross-Sampling algorithm inherits from the Cross-Sampling algorithm the same doubly crossed structure of its iterative computation. However, it searches for all occurrences of the pattern in the text by scanning characters from right to left, as in the Backward DAWG Matching (BDM) algorithm for the exact single pattern matching problem [11].

The BDM algorithm processes the pattern by constructing a *directed acyclic word graph* (DAWG) of the reversed pattern. The text is processed in windows of size m, which are searched for the longest prefix of the pattern from right to left by means of the DAWG. At the end of each search phase, either a longest prefix or a match is found. If no match is found, the window is shifted to the start position of the longest prefix, otherwise it is shifted to the start position of the second longest prefix. As in the BDM algorithm, the Backward-Cross-Sampling algorithm processes the text in windows of size m. Each attempt is identified by the last position j of the current window of the text. The window is searched for the longest prefix of the pattern which has a swapped occurrence ending at position j of the text. At the end of each attempt, the new value of j is computed by performing a safe shift to the right of the current window in such a way to left-align the current window of the text with the longest prefix matched in the previous attempt.

To this end, for any given position j in the text T, we let $\mathcal{S}_j^h$ denote the set of the integral values i such that the h-substring of P ending at position i has a swapped occurrence ending at position j of the text T. More formally, we have

$$\mathcal{S}_j^h = \{h-1 \leq i \leq m-1 \ : \ P[i-h+1 \,..\, i] \propto T_j\}\,,$$

for $0 \leq j < n$ and $0 \leq h \leq m$.

If $h-1 \in \mathcal{S}_j^h$, then there is a swapped occurrence of the prefix of the pattern of length h, i.e., $P[0 \,..\, h-1] \propto T_j$. In addition, it turns out that P has a swapped occurrence at location j of T if and only if $\mathcal{S}_j^m \neq \emptyset$. Indeed, if $\mathcal{S}_j^m \neq \emptyset$ then $\mathcal{S}_j^m = \{m-1\}$, for any given position j in the text.

The sets $\mathcal{S}_j^h$ can be computed efficiently by a dynamic programming algorithm, by exploiting the following very elementary property.

Lemma 10. *Let T and P be a text of length n and a pattern of length m, respectively. Then, for each $0 \leq j < n$, $0 \leq h \leq m$, and $h-1 \leq i < m$, we have that $P[i-h+1\,..\,i] \propto T_j$ if and only if one of the following two facts holds*
(a) $P[i-h+2\,..\,i] \propto T_j$ and $P[i-h+1] = T[j-h+1]$; or
(b) $P[i-h+3\,..\,i] \propto T_j$, $P[i-h+1] = T[j-h+2]$, and $P[i-h+2] = T[j-h+1]$.

Let us denote by $\mathcal{W}_j^h$, for $0 \leq j < n$ and $0 \leq h < m$, the collection of all values i such that $P[i-h+1] = T[j-h]$ and the $(h-1)$-substring ending at position i of P has a swapped occurrence ending at location j of the text T. More formally

$$\mathcal{W}_j^h = \{h \leq i < m-1 \ : \ P[i-h+2\,..\,i] \propto T_j \ \text{ and } \ P[i-h+1] = T[j-h]\}\,.$$

For any given position j in the text, the base case for $h = 0$ is given by

$$\mathcal{S}_j^0 = \{i \ : \ 0 \leq i < m\} \quad \text{and} \quad \mathcal{W}_j^0 = \{0 \leq i < m-1 \ : \ P[i+1] = T[j]\}. \tag{4}$$

Additionally, Lemma 10 justifies the following recursive definitions

$$\begin{aligned} \mathcal{S}_j^{h+1} &= \{h-1 \leq i \leq m-1 \ : \ (i \in \mathcal{S}_j^h \ \text{ and } \ P[i-h] = T[j-h]) \text{ or} \\ &\qquad\qquad\qquad\qquad\qquad (i \in \mathcal{W}_j^h \ \text{ and } \ P[i-h] = T[j-h+1])\,\} \\ \mathcal{W}_j^{h+1} &= \{h \leq i \leq m-1 \ : \ i \in \mathcal{S}_j^h \ \text{ and } \ P[i-h] = T[j-h-1]\}\,. \end{aligned} \tag{5}$$

Such relations, coupled with the initial conditions (4), allow one to compute the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ in an iterative fashion.

The code of the Backward-Cross-Sampling algorithm is shown in Fig. 3(A). For any attempt at position j of the text, we denote by ℓ the length of the longest prefix matched in the current attempt. Then the algorithm starts its computation with $j = m-1$ and $\ell = 0$. During each attempt, the window of the text is scanned from right to left, for $h = 1$ to m. If, for a given value of h, the algorithm states that element $(h-1) \in \mathcal{S}_j^h$ then ℓ is updated to value h.

The algorithm is not able to remember the characters read in previous iterations. Thus, an attempt ends successfully when h reaches the value m (a match is found), or unsuccessfully when both sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are empty. In any case, at the end of each attempt, the start position of the window, i.e., position $j-m+1$ in the text, can be shifted to the start position of the longest proper prefix detected during the backward scan. Thus the window is advanced $m-\ell$ positions to the right. Observe that since $\ell < m$, we plainly have that $m-\ell > 0$.

To compute the worst-case time complexity of the algorithm, we first observe that, since the algorithm does not remember the length of the prefix matched in previous attempts, each character of the text is processed at most m times during the searching phase. Thus the while-cycle of line 7 is executed $\mathcal{O}(nm)$ times. The for-cycles of line 9 and line 14 are executed $|\mathcal{S}_j^h|$ and $|\mathcal{W}_j^h|$ times, respectively. However, according to Lemma 6, for each position j of the text we can report only a single swapped occurrence of the substring $P[i-h+1\ldots i]$ in T_j, for each $h-1 \leq i < m$, which implies that $|\mathcal{S}_j^h| \leq m$ and $|\mathcal{W}_j^h| < m$.

Therefore the Backward-Cross-Sampling algorithm has a $\mathcal{O}(nm^2)$-time complexity and requires $\mathcal{O}(m)$ extra space to represent the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$.

```
(A) Backward-Cross-Sampling $(P, m, T, n)$
$j \leftarrow m-1$
while $j < n$ do
    $h \leftarrow \ell \leftarrow 0$
    $\mathcal{S}_j^0 \leftarrow \{i \ : \ 0 \leq i < m\}$
    $\mathcal{W}_j^0 \leftarrow \{0 \leq i < m-1 \ : \ P[i+1] = T[j]\}$
    while $h < m$ and $\mathcal{S}_j^h \cup \mathcal{W}_j^h \neq \emptyset$ do
        $\mathcal{S}_j^{h+1} \leftarrow \mathcal{W}_j^{h+1} \leftarrow \emptyset$
        if $(h-1) \in \mathcal{S}_j^h$ then $\ell \leftarrow h$
        for $i \in \mathcal{S}_j^h$ do
            if $i \geq h$ and $P[i-h] = T[j-h]$
                then $\mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \{i\}$
            if $i > h$ and $P[i-h] = T[j-h-1]$
                then $\mathcal{W}_j^{h+1} \leftarrow \mathcal{W}_j^{h+1} \cup \{i\}$
        for $i \in \mathcal{W}_j^h$ do
            if $i \geq h$ and $j-h < n-1$
                and $P[i-h] = T[j-h+1]$
                then $\mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \{i\}$
        $h \leftarrow h+1$
    if $(h-1) \in \mathcal{S}_j^h$ then Output$(j)$
    $j \leftarrow j+m-\ell$
```

```
(B) Approximate-BCS $(P, m, T, n)$
$j \leftarrow m-1$
while $j < n$ do
    $h \leftarrow \ell \leftarrow c \leftarrow 0$
    $\mathcal{S}_j^0 \leftarrow \{i \ : \ 0 \leq i < m\}$
    $\mathcal{W}_j^0 \leftarrow \{0 \leq i < m-1 \ : \ P[i+1] = T[j]\}$
    while $h < m$ and $\mathcal{S}_j^h \cup \mathcal{W}_j^h \neq \emptyset$ do
        $\mathcal{S}_j^{h+1} \leftarrow \mathcal{W}_j^{h+1} \leftarrow \emptyset$
        if $(h-1) \in \mathcal{S}_j^h$ then $\ell \leftarrow h$
        for $i \in \mathcal{S}_j^h$ do
            if $i \geq h$ and $P[i-h] = T[j-h]$
                then $\mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \{i\}$
            if $i > h$ and $P[i-h] = T[j-h-1]$
                then $\mathcal{W}_j^{h+1} \leftarrow \mathcal{W}_j^{h+1} \cup \{i\}$
        for $i \in \mathcal{W}_j^h$ do
            if $i \geq h$ and $j-h < n-1$
                and $P[i-h] = T[j-h+1]$
                then $\mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \{i\}$
        if $m-1 \in \mathcal{S}_j^{h+1}$ and $m-1 \notin \mathcal{S}_j^h$
            then $c \leftarrow c+1$
        $h \leftarrow h+1$
    if $(h-1) \in \mathcal{S}_j^h$ then Output$(j,c)$
    $j \leftarrow j+m-\ell$
```

Fig. 3. **(A)** The Backward-Cross-Sampling algorithm for the swap matching problem. **(B)** The Approximate-BCS algorithm for the approximate swap matching problem.

4.2. *Approximate Swap Matching Problem*

The Approximate-BCS algorithm searches for all the swap occurrences of a pattern P (of length m) in a text T (of length n) using the same right-to-left scan used by the Backward-Cross-Sampling algorithm described above.

Before entering into details we need to introduce some results on which the approximate version of the Backward-Cross-Sampling algorithm is based.

The following result follows immediately from (5).

Lemma 11. *Let P and T be a pattern of length m and a text of length n, respectively. Moreover let $m-1 \leq j \leq n-1$ and $0 \leq i < m$. If $i \in \mathcal{S}_j^\gamma$, then it follows that $i \in (\mathcal{S}_j^h \cup \mathcal{W}_j^h)$, for $1 \leq h \leq \gamma$.* ■

The following technical lemma helps in identifying identical characters in the pattern using information gathered in the sets $\mathcal{S}_j^\gamma$, $\mathcal{S}_j^{\gamma-1}$, and $\mathcal{W}_j^{\gamma-1}$.

Lemma 12. *Let P and T be a pattern of length m and a text of length n, respectively. Then, for every $m-1 \leq j \leq n-1$ and $0 \leq i < m$ such that $i \in (\mathcal{S}_j^\gamma \cap \mathcal{W}_j^{\gamma-1} \cap \mathcal{S}_j^{\gamma-1})$, we have $P[i-\gamma+1] = P[i-\gamma+2]$.*

Proof. From $i \in (\mathcal{S}_j^\gamma \cap \mathcal{S}_j^{\gamma-1})$ and $i \in \mathcal{W}_j^{\gamma-1}$ it follows that $P[i-\gamma+1] = T[j-\gamma+1]$ and $P[i-\gamma+2] = T[j-\gamma+1]$. Thus $P[i-\gamma+1] = P[i-\gamma+2]$. □

Lemma 13. *Let P and T be a pattern of length m and a text of length n, respectively. Moreover let $m-1 \leq j \leq n-1$ and $0 \leq i < m$. Then, if $i \in \mathcal{S}_j^{\gamma}$, there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $i \in (\mathcal{S}_j^{\gamma} \setminus \mathcal{S}_j^{\gamma-1})$.*

Proof. Before entering into details we remember that, by Definition 1, a swap can take place between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $P[i-\gamma+1] = T[j-\gamma+2]$, $P[i-\gamma+2] = T[j-\gamma+1]$, and $P[i-\gamma+1] \neq P[i-\gamma+2]$.

Now, suppose that $i \in \mathcal{S}_j^{\gamma}$ and that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We proceed by contradiction to prove that $i \notin \mathcal{S}_j^{\gamma-1}$. We have

(i)	$i \in \mathcal{S}_j^{\gamma}$	(by hypothesis)
(ii)	$P[i-\gamma+2] = T[j-\gamma+1] \neq P[i-\gamma+1]$	(by hypothesis)
(iii)	$i \in \mathcal{S}_j^{\gamma-1}$	(by contradiction)
(iv)	$i \notin \mathcal{W}_j^{\gamma-1}$	(by (ii), (iii), and Lemma 12)
(v)	$P[i-\gamma+1] = T[j-\gamma+1]$	(by (i) and (iv))

obtaining a contradiction between (ii) and (v).

Next, suppose that $i \in (\mathcal{S}_j^{\gamma} \setminus \mathcal{S}_j^{\gamma-1})$. We prove that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We have

(i)	$i \in \mathcal{S}_j^{\gamma}$ and $i \notin \mathcal{S}_j^{\gamma-1}$	(by hypothesis)
(ii)	$i \in \mathcal{W}_j^{\gamma-1}$	(by (i) and Lemma 11)
(iii)	$i \in \mathcal{S}_j^{\gamma-2}$	(by (ii) and (5))
(iv)	$P[i-\gamma+1] = T[j-\gamma+2]$	(by (i) and (ii))
(v)	$P[i-\gamma+2] = T[j-\gamma+1]$	(by (ii))
(vi)	$P[i-\gamma+2] \neq T[j-\gamma+2] = P[i-\gamma+1]$	(by (i) and (iii)).

□

The following corollary is an immediate consequence of Lemmas 13 and 11.

Corollary 14. *Let P and T be strings of length m and n, respectively, over a common alphabet Σ. Then, for $m-1 \leq j \leq n-1$, P has a swapped occurrence in T at location j with k swaps, i.e., $P \propto_k T_j$, if and only if $(m-1) \in \mathcal{S}_j^m$ and $|\Delta_j| = k$, where $\Delta_j = \{1 \leq h < m \ : \ (m-1) \ \in \ (\mathcal{S}_j^{h+1} \setminus \mathcal{S}_j^h)\}$.* ■

In consideration of the preceding corollary, the Approximate-BCS algorithm maintains a counter which is incremented every time $(m-1) \in (\mathcal{S}_j^{h+1} \setminus \mathcal{S}_j^h)$, for any $1 < h \leq m$, in order to count the swaps for an occurrence ending at a given position j of the text.

For any attempt at position j of the text, let us denote by ℓ the length of the longest prefix matched in the current attempt. Then the algorithm starts its computation with $j = m-1$ and $\ell = 0$. During each attempt, the window of the text is scanned from right to left, for $h = 1, \ldots, m$. If, for a given value of h, the algorithm discovers that $(h-1) \in \mathcal{S}_j^h$, then ℓ is set to the value h.

The algorithm is not able to remember the characters read in previous iterations. Thus, an attempt ends successfully when h reaches the value m (a match is found), or unsuccessfully when both sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are empty. In any case, at the end of each attempt, the start position of the window, i.e., position $j-m+1$ in the text, can be shifted to the start position of the longest proper prefix detected during the backward scan. Thus the window is advanced $m-\ell$ positions to the right. Observe that since $\ell < m$, we plainly have that $m-\ell > 0$.

The code of the Approximate-BCS algorithm is shown in Fig. 3(B). Its time complexity is $\mathcal{O}(nm^2)$ in the worst case and requires $\mathcal{O}(m)$ extra space.

4.3. *Bit-Parallel Implementation*

In this section we present practical implementations of the Backward-Cross-Sampling algorithms based on the bit-parallelism technique. The resulting algorithms work as the BNDM (Backward Nondeterministic DAWG Matching) algorithm [14], which is a bit-parallel implementation of the BDM algorithm, where the simulation of a nondeterministic automaton takes place by updating the state vector much as in the Shift-And algorithm [7].

In the bit-parallel variant of the Backward-Cross-Sampling algorithm, the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are represented as lists of m bits, D_j^h and C_j^h respectively.

The $(i-h+1)$-st bit of D_j^h is set to 1 if $i \in \mathcal{S}_j^h$, i.e., if $P[i-h+1\,..\,i] \propto T_j$, whereas the $(i-h+1)$-st bit of C_j^h is set to 1 if $i \in \mathcal{W}_j^h$, i.e., if $P[i-h+2\,..\,i] \propto T_j$ and $P[i-h+1] = T[j-h]$. All remaining bits are set to 0. Notice that if $m \leq w$, each bit vector fits in a single computer word, whereas if $m > w$ we need $\lceil m/w \rceil$ computer words to represent each of the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$.

For each character c of the alphabet Σ, the algorithm maintains a bit mask $M[c]$ whose i-th bit is set to 1 if $P[i] = c$.

As in the Backward-Cross-Sampling algorithm, the text is processed in windows of size m, identified by the last position j, and the first attempt starts at position $j = m-1$. For any searching attempt at location j of the text, the bit vectors D_j^1 and C_j^1 are initialized to $M[T[j]] \mid (M[T[j+1]] \mathbin{\&} (M[T[j]] \ll 1))$ and $M[T[j-1]]$, respectively, according to the base cases shown in (4) and recursive expressions shown in (5). Then the current window of the text, i.e. $T[j-m+1\,..\,j]$, is scanned from right to left, by reading character $T[j-h+1]$, for increasing values of h. Namely, for each value of $h > 1$, the bit vector D_j^{h+1} is computed in terms of D_j^h and C_j^h, by performing the following bitwise operations:

$(a)\quad D_j^{h+1} \leftarrow (D_j^h \ll 1) \mathbin{\&} M[T[j-h]]\,,$
$(b)\quad D_j^{h+1} \leftarrow D_j^{h+1} \mid ((C_j^h \ll 1) \mathbin{\&} M[T[j-h+1]])\,.$

Concerning (a), by a left shift of D_j^h, all elements of $\mathcal{S}_j^h$ are added to the set $\mathcal{S}_j^{h+1}$. Then, by performing a bitwise and with the mask $M[T[j-h]]$, all elements i such that $P[i-h] \neq T[j-h]$ are removed from $\mathcal{S}_j^{h+1}$. Similarly, the bit operations in (b) have the effect of adding to $\mathcal{S}_j^{h+1}$ all elements i in $\mathcal{W}_j^h$ such that

$P[i-h] = T[j-h+1]$. Formally, we have the following correspondence:

$(a')\quad \mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^h \setminus \{i \in \mathcal{S}_j^h \ : \ P[i-h] \neq T[j-h]\}\,,$

$(b')\quad \mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \mathcal{W}_j^h \setminus \{i \in \mathcal{W}_j^h \ : \ P[i-h] \neq T[j-h+1]\}\,.$

Similarly, C_j^{h+1} is computed by performing the following bitwise operations:

$$(c)\quad C_j^{h+1} \leftarrow (D_j^h \ll 1) \ \& \ M[T[j-h-1]]$$

which have the effect of adding to the set $\mathcal{W}_j^{h+1}$ all elements of the set $\mathcal{S}_j^h$ (by shifting D_j^h to the left by one position) and of removing all elements i such $P[i] \neq T[j-h-1]$ holds (by a bitwise and with the mask $M[T[j-h-1]]$).

More formally, we have the following symbolic correspondence:

$$(c')\quad \mathcal{W}_j^{h+1} \leftarrow \mathcal{S}_j^h \setminus \{i \in \mathcal{S}_j^h \ : \ P[i-h] \neq T[j-h-1]\}\,.$$

As in the Backward-Cross-Sampling algorithm, an attempt ends when $h = m$ or $(D_j^h | C_j^h) = 0$. If $h = m$ and $D_j^h \neq 0$, a swap match at position j of the text is reported. In any case, if $h < m$ is the largest value such that $D_j^h \neq 0$, then a prefix of the pattern, of length $\ell = h$, which has a swapped occurrence ending at position j, has been found. Thus a safe shift of $m - \ell$ positions to the right can take place.

In practice, we can use just two vectors to implement the sets D_j^h and C_j^h. Thus, during the h-th iteration of the algorithm at a given location j of the text, vector D_j^h is transformed into vector D_j^{h+1} and vector C_j^h is transformed into vector C_j^{h+1}. The resulting BP-Backward-Cross-Sampling algorithm is shown in Fig. 4(A). It achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma\lceil m/w \rceil)$ extra space, where σ is the alphabet size. If the length of the pattern is $m \leq w$, then the algorithm finds all swapped matches in $\mathcal{O}(nm)$ time and $\mathcal{O}(\sigma)$ extra space.

In the bit-parallel version of the Approximate-BCS algorithm, named Approximate-BPBCS, the sets $\mathcal{S}_j^h$, $\mathcal{W}_j^h$, and $\mathcal{C}_j^h$ are represented and computed as in the BP-Backward-Cross-Sampling algorithm, using equations (a), (b), and (c) as described above.

Moreover, in order to count the number of swaps, observe that the $(i-h+1)$-st bit of D_j^h is set to 1 if $i \in \mathcal{S}_j^h$. Thus, the condition $(m-1) \in (\mathcal{S}_j^{h+1} \setminus \mathcal{S}_j^h)$ can be implemented by the following bitwise condition:

$(d)\quad ((D^{h+1} \ \& \ \sim (D^h \ll 1)) \ \& \ (1 \ll h)) \neq 0\,.$

The counter for keeping track of the number of swaps requires $\log(\lfloor m/2 \rfloor + 1)$ bits to be implemented. This compares favorably with the BP-Approximate-Cross-Sampling algorithm which uses instead m counters of $\log(\lfloor m/2 \rfloor + 1)$ bits, one for each prefix of the pattern.

The resulting Approximate-BPBCS algorithm is shown in Fig. 4(B). It achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma\lceil m/w \rceil + \log(\lfloor m/2 \rfloor + 1))$ extra-space, where σ is the alphabet size. If the pattern fits in few machine words, then the algorithm finds all swapped matches and their corresponding counts in $\mathcal{O}(nm)$-time and $\mathcal{O}(\sigma)$ extra-space.

```
(A) BP-Backward-Cross-Sampling $(P, m, T, n)$
1.  $F \leftarrow 10^{m-1}$
2.  for $c \in \Sigma$ do $M[c] \leftarrow 0^m$
3.  for $i \leftarrow 0$ to $m-1$ do
4.      $M[P[i]] \leftarrow M[P[i]] \mid F$
5.      $F \leftarrow F \gg 1$
6.  $j \leftarrow m-1$
7.  $F \leftarrow 10^{m-1}$
8.  while $j < n$ do
9.      $h \leftarrow 1, \ell \leftarrow 0$
10.     $D \leftarrow M[T[j]]$
11.     if $j < n-1$ then
12.         $H \leftarrow M[T[j+1]]$ & $(M[T[j]] \ll 1)$
13.         $D \leftarrow D \mid H$
14.     $C \leftarrow M[T[j-1]]$
15.     while $h < m$ and $(D \mid C) \neq 0$ do
16.         $D' \leftarrow D \ll 1$
17.         if $D$ & $F \neq 0$ then $\ell \leftarrow h$
18.         $H \leftarrow (C \ll 1)$ & $M[T[j-h+1]]$
19.         $C \leftarrow D'$ & $M[T[j-h-1]]$
20.         $D \leftarrow D'$ & $M[T[j-h]]$
21.         $D \leftarrow D \mid H$
22.         $h \leftarrow h+1$
23.     if $D \neq 0$ then Output($j$)
24.     $j \leftarrow j + m - \ell$
```

```
(B) Approximate-BPBCS $(P, m, T, n)$
1.  $F \leftarrow 10^{m-1}$
2.  for $c \in \Sigma$ do $M[c] \leftarrow 0^m$
3.  for $i \leftarrow 0$ to $m-1$ do
4.      $M[P[i]] \leftarrow M[P[i]] \mid F$
5.      $F \leftarrow F \gg 1$
6.  $j \leftarrow m-1$
7.  $F \leftarrow 10^{m-1}$
8.  while $j < n$ do
9.      $h \leftarrow 1, \ell \leftarrow c \leftarrow 0$
10.     $D \leftarrow M[T[j]]$
11.     if $j < n-1$ then
12.         $H \leftarrow M[T[j+1]]$ & $(M[T[j]] \ll 1)$
13.         $D \leftarrow D \mid H$
14.     $C \leftarrow M[T[j-1]]$
15.     while $h < m$ and $(D \mid C) \neq 0$ do
16.         $D' \leftarrow D \ll 1$
17.         if $D$ & $F \neq 0$ then $\ell \leftarrow h$
18.         $H \leftarrow (C \ll 1)$ & $M[T[j-h+1]]$
19.         $C \leftarrow D'$ & $M[T[j-h-1]]$
20.         $D \leftarrow D'$ & $M[T[j-h]]$
21.         $D \leftarrow D \mid H$
22.         if $(D$ & $\sim D')$ & $(1 \ll h) \neq 0$
23.             then $c \leftarrow c+1$
24.         $h \leftarrow h+1$
25.     if $D \neq 0$ then Output($j$,$c$)
26.     $j \leftarrow j + m - \ell$
```

Fig. 4. Two bit-parallel algorithms. **(A)** The BP-Backward-Cross-Sampling algorithm for the swap matching problem. **(B)** The Approximate-BPBCS algorithm for the approximate swap matching problem.

5. Tuning the Dynamic Programming Recurrence

The Cross-Sampling and Backward-Cross-Sampling algorithms, and all the variants based on them, have to read three text characters per iteration to update the corresponding sets; for example, to compute the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$, the Cross-Sampling algorithm reads the characters $T[j-1]$, $T[j]$, and $T[j+1]$. It is easy to devise a similar characterization of the sets definition and of the recurrence so that the algorithm has to access only one character per iteration. We describe this kind of variant for the Cross-Sampling algorithm, but it is trivial to adapt it to the Backward-Cross-Sampling case. The definition of $\mathcal{S}_j$ is as in the original algorithm. Instead, $\mathcal{S}'_j$ is defined as follows:

$$\mathcal{S}'_j = \{0 \leq i < m-1 \mid P_{i-1} \propto T_{j-1} \text{ and } P[i+1] = T[j]\}\,.$$

Following the new definition of $\mathcal{S}'_j$, the recurrence to compute both $\mathcal{S}_j$ and $\mathcal{S}'_j$ is modified in

$$\begin{aligned} \mathcal{S}_{j+1} = \{i \leq m-1 \mid \ & ((i-1) \in \mathcal{S}_j \cup \{-1\} \text{ and } P[i] = T[j+1]) \text{ or} \\ & ((i-1) \in \mathcal{S}'_j \text{ and } P[i-1] = T[j+1])\,\} \\ \mathcal{S}'_{j+1} = \{i < m-1 \mid \ & (i-1) \in \mathcal{S}_j \cup \{0\} \text{ and } P[i+1] = T[j+1]\}\,. \end{aligned} \tag{6}$$

Based on the modified recurrence we can devise a different bit-parallel simulation of the CROSS-SAMPLING algorithm; in particular the bit vector D_j can be computed with the following bitwise operations:

$$\begin{array}{ll}
D_j \leftarrow D_{j-1} \ll 1 & \mathcal{S}_j = \{i : (i-1) \in \mathcal{S}_{j-1}\} \\
D_j \leftarrow D_j \mid 1 & \mathcal{S}_j = \mathcal{S}_j \cup \{0\} \\
D_j \leftarrow D_j \,\&\, M[T[j]] & \mathcal{S}_j = \mathcal{S}_j \setminus \{i : P[i] \neq T[j]\} \\
D_j \leftarrow D_j \mid H^1 & \mathcal{S}_j = \mathcal{S}_j \cup \{i : (i-1) \in \mathcal{S}'_{j-1} \wedge P[i-1] = T[j]\},
\end{array}$$

where $H^1 = \big((D'_{j-1} \,\&\, M[T[j]]) \ll 1\big)$.

Similarly, the bit vector D'_j can be computed with the following bitwise operations:

$$\begin{array}{ll}
D'_j \leftarrow D_{j-1} \ll 1 & \mathcal{S}'_j = \{i : (i-1) \in \mathcal{S}_{j-1}\} \\
D'_j \leftarrow D'_j \mid 1 & \mathcal{S}'_j = \mathcal{S}'_j \cup \{0\} \\
D'_j \leftarrow D'_j \,\&\, (M[T[j]] \gg 1) & \mathcal{S}'_j = \mathcal{S}'_j \setminus \{i : P[i+1] \neq T[j]\}.
\end{array}$$

6. Experimental Results

Next we present experimental data which allow to compare under various conditions the reviewed string matching algorithms in terms of their running times.

All algorithms have been implemented in the C programming language and were used to search for the same strings in large fixed text buffers on a PC with Intel Pentium M processor of 1.7GHz and a memory of 512Mb. In particular, all algorithms have been tested on random texts and on three real world problems, with patterns of length $m = 4, 8, 16, 32$. In the case of random texts, the algorithms have been tested on three Randσ problems, for $\sigma = 8, 32, 128$. Each Randσ problem consists in searching a set of 400 random patterns of a given length in a 4Mb random text over a common alphabet of size σ, with a uniform character distribution. We notice moreover that in our results computed for the approximate swap matching problem we use sets of 100 random patterns.

The tests on real world problems have been performed on a genome sequence, on a protein sequence, and on a natural language text. The genome used is a sequence of $4,638,690$ base pairs of *Escherichia coli*, taken from the file E.coli of the Large Canterbury Corpus.[a] The protein sequence used in the tests is a 2.4Mb file with 20 different characters from the human genome. Finally, as natural language text we used the file world192.txt (The CIA World Fact Book) from the Large Canterbury Corpus, which contains $2,473,400$ characters drawn from an alphabet of 93 different characters.

6.1. *Results for Standard Swap Matching*

For the Standard Swap Matching problem we have compared under various conditions the following string matching algorithms in terms of their running times:

[a] http://www.data-compression.info/Corpora/CanterburyCorpus/

the Iliopoulos-Rahman algorithm (IR), the Cross-Sampling algorithm (CS), the BP-Cross-Sampling algorithm (BPCS), the Backward-Cross-Sampling algorithm (BCS) and the BP-Backward-Cross-Sampling algorithm (BPBCS).

We have chosen to exclude from our experimental comparison the naive algorithm and all algorithms based on the FFT technique, since the overhead of such algorithms is quite high, resulting in very bad performances.

In the tables below, running times have been expressed in hundredths of seconds and the best results are bold-faced.

	Rand8 problem				Rand32 problem				Rand128 problem			
m	4	8	16	32	4	8	16	32	4	8	16	32
IR	**3.45**	3.42	3.44	3.56	2.92	2.95	2.94	2.95	3.55	3.61	3.59	3.64
CS	66.6	67.2	67.5	69.0	60.0	59.7	59.7	59.2	59.9	59.6	59.3	59.1
BPCS	3.96	3.90	3.90	3.91	3.03	3.05	3.08	3.06	3.12	3.15	3.16	3.13
BCS	62.1	41.1	29.4	22.4	46.2	29.0	20.5	15.6	42.7	25.7	17.7	13.3
BPBCS	4.14	**2.00**	**1.18**	**0.80**	**2.65**	**1.93**	**1.00**	**0.24**	**2.00**	**1.04**	**0.75**	**0.18**

In the case of random texts the experimental results show that the BPBCS algorithm obtains the best run-time performance in most cases. In particular, for very short patterns and small alphabets, our algorithm is second only to the IR algorithm. We notice that the algorithms IR, CS, and BPCS show a linear behavior, whereas both algorithms BCS and BPBCS are characterized by a decreasing trend. Observe moreover that, in the case of small alphabets and pattern longer than 16 characters, the BPBCS algorithm is at least three times faster than BPCS and IR. Such a relation increases to thirty times for large alphabets.

	genome sequence ($\sigma = 4$)				protein sequence ($\sigma = 20$)				natural language ($\sigma = 93$)			
m	4	8	16	32	4	8	16	32	4	8	16	32
IR	**3.07**	**3.06**	3.08	3.10	**1.99**	2.00	2.00	1.99	**1.85**	1.82	1.85	1.85
CS	83.0	79.9	79.3	79.4	45.1	45.2	45.6	44.4	36.9	36.6	36.4	36.2
BPCS	6.82	3.95	3.92	3.94	2.03	2.01	2.05	2.02	2.05	1.97	1.99	1.98
BCS	102	67.0	49.0	38.6	31.1	22.4	16.4	12.6	30.4	19.3	13.6	10.3
BPBCS	10.1	3.93	**2.01**	**1.12**	2.13	**1.18**	**0.59**	**0.07**	2.00	**0.99**	**0.21**	**0.01**

The above experimental results concerning the real world problems show that in most cases the BPBCS algorithm obtains the best results and only sporadically is second to the IR algorithm. Moreover, in the case of natural language texts and long patterns, the BPBCS algorithm is about 100 times faster than the IR algorithm.

6.2. *Results for Approximate Swap Matching*

Next we report experimental results relative to an extensive comparison under various conditions of the following algorithms: the Approximate-Cross-Sampling (ACS), the BP-Approximate-Cross-Sampling (BPACS), the Approximate-BCS (ABCS), the Approximate-BPBCS (BPABCS), the Iliopoulos-Rahman algorithm with a naive check of the swaps (IR*) and the BP-Backward-Cross-Sampling algorithm with a naive check of the swaps (BPBCS*)

We have chosen to include in our comparison also the algorithms IR* and BPBCS*, since the algorithms IR and BPBCS turned out, in [8], to be the most efficient solutions for the swap matching problem. Instead, as before, the naive algorithm and algorithms based on the FFT technique have not been taken into consideration, as their overhead is quite high, resulting in poor performances.

	Rand8 problem				Rand32 problem				Rand128 problem			
m	4	8	16	32	4	8	16	32	4	8	16	32
ACS	4.76	4.75	4.78	4.79	5.28	5.08	5.26	5.29	5.07	5.05	4.99	5.21
ABCS	11.6	7.27	4.73	3.30	9.41	5.83	3.95	2.89	8.67	5.28	3.28	2.52
BPACS	0.83	0.83	0.83	0.82	0.77	0.74	0.77	0.79	0.83	0.83	0.83	0.83
BPABCS	0.41	**0.22**	**0.14**	**0.09**	0.29	**0.18**	**0.11**	**0.07**	0.24	**0.14**	**0.09**	**0.07**
IR*	**0.28**	0.27	0.27	0.28	**0.27**	0.27	0.27	0.27	0.35	0.35	0.35	0.33
BPBCS*	0.38	0.24	0.15	0.10	0.28	0.19	0.11	0.07	**0.23**	0.15	0.09	0.07

The experimental results on random texts show that the BPABCS algorithm obtains the best run-time performance in most cases. For very short patterns and small alphabets, our algorithm is second only to the IR* algorithm. In the case of very short patterns and large alphabets, our algorithm is second only to the BPBCS* algorithm. We notice that the algorithms IR*, ACS, and BPACS show a linear behavior, whereas ABCS and BPABCS are characterized by a decreasing trend.

	genome segence ($\sigma = 4$)				protein sequence ($\sigma = 20$)				natural language ($\sigma = 93$)			
m	4	8	16	32	4	8	16	32	4	8	16	32
ACS	5.62	5.64	5.63	6.04	3.77	3.78	3.72	3.74	3.17	2.75	2.75	2.75
ABCS	18.0	11.21	7.52	5.35	7.04	4.55	3.16	2.35	6.17	4.05	2.70	1.86
BPACS	0.95	0.91	0.76	0.84	0.56	0.58	0.56	0.51	0.49	0.49	0.49	0.49
BPABCS	0.64	0.31	**0.23**	0.14	0.24	**0.14**	**0.08**	**0.05**	0.19	**0.11**	**0.07**	**0.04**
IR*	**0.26**	**0.28**	0.31	0.31	0.38	0.39	0.38	0.38	0.17	0.16	0.16	0.16
BPBCS*	0.67	0.36	0.23	**0.14**	**0.24**	0.14	0.08	**0.05**	**0.16**	0.12	0.07	0.05

From the above experimental results on real world problems, it turns out that the BPABCS algorithm obtains in most cases the best results and, in the case of very short patterns, is second to IR* (for the genome sequence) and to BPBCS* (for the protein sequence and the natual language text buffer).

7. Conclusions

We have presented new efficient algorithms for both the standard and approximate version of the Swap Matching problem. In particular, we have devised two algorithms with $\mathcal{O}(nm)$ and $\mathcal{O}(nm^2)$ worst case time complexity, respectively. We have also shown efficient implementations of them, based on bit-parallelism, which achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ worst-case time complexity, with patterns whose length is comparable to the word-size of the target machine. From an extensive comparison with some of the most effective algorithms for the swap matching problem, it turns out that our algorithms are very flexible and achieve very good results in practice.

References

[1] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. In *IEEE Symposium on Foundations of Computer Science*, pages 144–153, 1997.

[2] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.

[3] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003.

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

[5] A. Amir, M. Lewenstein, and E. Porat. Approximate swapped matching. *Inf. Process. Lett.*, 83(1):33–39, 2002.

[6] P. Antoniou, C. Iliopoulos, I. Jayasekera, and M. Rahman. Implementation of a swap matching algorithm using a graph theoretic model. In *Bioinformatics Research and Development, Second International Conference, BIRD 2008*, volume 13 of *Communications in Computer and Information Science*, pages 446–455. Springer, 2008.

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

[8] M. Campanelli, D. Cantone, and S. Faro. A new algorithm for efficient pattern matching with swaps. In *Combinatorial Algorithms, 20th International Workshop, IWOCA 2009, Hradec nad Moravicí, Czech Republic, June 28-July 2, 2009, Revised Selected Papers*, volume 5874 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2009.

[9] M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta. An efficient algorithm for approximate pattern matching with swaps. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2009*, pages 90–104, Czech Technical University, Prague, Czech Republic, 2009.

[10] 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, Spindleruv Mlýn, Czech Republic, January 24-30, 2009. Proceedings*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009.

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

[12] 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*, volume 4910 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 2008.

[13] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95*, volume 937 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1995.

[14] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *J. Exp. Algorithmics*, 5:4, 2000.