# An Efficient Algorithm for Approximate Pattern Matching with Swaps

Matteo Campanelli[1], Domenico Cantone[2], Simone Faro[2], and Emanuele Giaquinta[2]

[1] Università di Catania, Scuola Superiore di Catania
Via San Nullo 5/i, I-95123 Catania, Italy
[2] Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
`macampanelli@ssc.unict.it`, {`cantone` | `faro` | `giaquinta`}`@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 the Approximate Pattern Matching problem with Swaps one seeks to compute, 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 present new efficient algorithms for the Approximate Swap Matching problem. In particular, we first present a $\mathcal{O}(nm^2)$ algorithm, where $m$ is the length of the pattern and $n$ is the length of the text, which is a variation of the Backward-Cross-Sampling algorithm, a recent solution to the swap matching problem. Subsequently, we propose an efficient implementation of our algorithm, based on the bit-parallelism technique. The latter solution achieves a $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity, where $\sigma$ is the dimension of the alphabet.
From an extensive comparison with some of the most recent and effective algorithms for the approximate 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 $\Sigma$ of size $\sigma$. 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 [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 an 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 [12]. 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 [11]. 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.

More recently, in 2009, Cantone and Faro [9] presented a first approach for solving the swap matching problem with short patterns in linear time. Their algorithm, named Cross-Sampling, though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named BP-Cross-Sampling, 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 a subsequent paper [8] a more efficient algorithm, named Backward-Cross-Sampling and based on a similar structure as the one of the Cross-Sampling algorithm, has been proposed. The Backward-Cross-Sampling scans the text from right to left and has a $\mathcal{O}(nm^2)$-time complexity, whereas its bit-parallel implementation, named BP-Backward-Cross-Sampling, works in $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity. However, despite their higher worst-case running times, in practice the algorithms Backward-Cross-Sampling and BP-Backward-Cross-Sampling show a better behavior than their predecessors Cross-Sampling and BP-Cross-Sampling, respectively.

In this paper we are interested in the approximate variant of the swap matching problem. 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 text location $j$.

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 [9] Cantone and Faro presented also an extension of the Cross-Sampling algorithm, named Approximate-Cross-Sampling, for the approximate swap matching problem. However, its bit-parallel implementation has a notably high space overhead, since it requires $(m\log(\lfloor m/2\rfloor+1)+m)$ bits, with $m$ the length of the pattern.

In this paper we present a variant of the Backward-Cross-Sampling algorithm for the approximate swap matching problem, which works in $\mathcal{O}(nm^2)$-time and requires $\mathcal{O}(m)$-space. Its bit-parallel implementation, in contrast with the BP-Approximate-Cross-Sampling algorithm, does not add any space overhead and maintains a worst-case $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity, when the pattern size is comparable to the word size in the target machine, and is very fast in practice.

The rest of the paper is organized as follows. In Section 2 we recall some preliminary definitions. Then in Section 3 we describe the Approximate-Cross-Sampling algorithm and its bit-parallel variant. In Section 4 we present a variant of the Backward-Cross-Sampling algorithm for the approximate swap matching problem and its straightforward bit-parallel implementation. Then we compare, in Section 5, our newly proposed algorithms against the most effective algorithms present in literature and, finally, we briefly draw our conclusions in Section 6.

## 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 length$(P) = m$. In particular, for $m = 0$ we obtain the empty string $\varepsilon$. We denote by $P[i]$ the $(i+1)$-st character of $P$, for $0 \leq i <$ 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 $0 \leq i \leq j <$ 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 $0 \leq i <$ length$(P)$. Similarly, we say that $P'$ is a prefix of $P$ if $P' = P[0 \,..\, i-1]$, for some $0 \leq i \leq$ 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 $\varepsilon$. 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 $\pi : \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$ 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 are 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 $\pi$ 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 $\pi$ 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 $\pi$ 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 $j \in \{m-1, \dots, n-1\}$ 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 $j \in \{m-1, \dots, n-1\}$ and $0 \le k \le \lfloor m/2 \rfloor$, 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.

**Lemma 6 ([9]).** *Let $P$ and $R$ be strings of length $m$ over an alphabet $\Sigma$ and suppose that there exists a swap permutation $\pi$ such that $\pi(P) = R$. Then $\pi$ is unique.*

*Proof.* Suppose, by way of contradiction, that there exist two different swap permutations $\pi$ and $\pi'$ such that $\pi(P) = \pi'(P) = R$. Then there must exist an index $i$ such that $\pi(i) \neq \pi'(i)$. Without loss of generality, let us assume that $\pi(i) < \pi'(i)$ and suppose that $i$ be the smallest index such that $\pi(i) \neq \pi'(i)$. Since $\pi(i), \pi'(i) \in \{i-1, i, i+1\}$, by Definition 1(b), it is enough to consider the following three cases:

**Case 1:** $\pi(i) = i-1$ and $\pi'(i) = i$.
Then, by Definition 1(a), we have $\pi(i-1) = i$, so that $P[\pi(i-1)] = P[i] = P[\pi'(i)] = P[\pi(i)]$, thus violating Definition 1(c).
**Case 2:** $\pi(i) = i$ and $\pi'(i) = i+1$.
Since by Definition 1(a) we have $\pi'(i+1) = i$, then $P[\pi'(i+1)] = P[i] = P[\pi(i)] = P[\pi'(i)]$, thus again violating Definition 1(c).
**Case 3:** $\pi(i) = i-1$ and $\pi'(i) = i+1$.
By Definition 1(c) we have $\pi(i-1) = \pi'(i+1) = i$. Thus $\pi'(i-1) \neq i = \pi(i-1)$, contradicting the minimality of $i$. □

**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 $j \in \{m-1, \dots, n-1\}$, then there exists a unique swapped occurrence of $P$ in $T$ ending at position $j$.* □

## 3 The Approximate-Cross-Sampling Algorithm

The Approximate-Cross-Sampling algorithm [9] computes the swap occurrences of all prefixes of a pattern $P$ (of length $m$) in continuously increasing prefixes of a text $T$ (of length $n$), using a dynamic programming approach. Additionally, for each occurrence of $P$ in $T$, the algorithm computes also the number of swaps necessary to convert the pattern in its swapped occurrence.

In particular, during its $(j+1)$-st iteration, for $j = 0, 1, \dots, n-1$, it is established whether $P_i \propto_k T_j$, for each $i = 0, 1, \dots, m-1$, by exploiting information gathered during previous iterations as follows.

Let us put

$$\bar{\mathcal{S}}_j =_{\text{Def}} \{(i,k) \mid 0 \le i \le m-1 \text{ and } P_i \propto_k T_j\}$$
$$\bar{\lambda}_j =_{\text{Def}} \begin{cases} \{(0,0)\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise}, \end{cases}$$

for $0 \le j \le n-1$, and

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

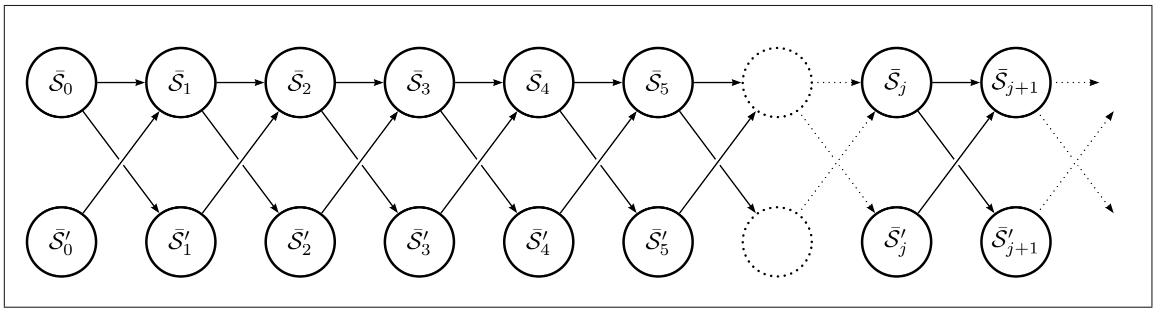

**Figure 1.** A graphic representation of the iterative fashion for computing sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ for increasing values of $j$.

for $1 \leq j < n-1$. Then the following recurrences hold:

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

where the base cases are given by $\mathcal{S}_0 = \bar{\lambda}_0$ and $\mathcal{S}'_0 = \bar{\lambda}_1$.

Such relations allow one to compute the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ in an iterative fashion, where $\bar{\mathcal{S}}_{j+1}$ is computed in terms of both $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$, whereas $\bar{\mathcal{S}}'_{j+1}$ needs only $\bar{\mathcal{S}}_j$ for its computation. The resulting dependency graph has a doubly crossed structure, from which the name of the algorithm in Fig. 2(A), Approximate-Cross-Sampling, for the swap matching problem. Plainly, the time complexity of the Approximate-Cross-Sampling algorithm is $\mathcal{O}(nm)$.

In [9], a bit-parallel implementation of the Approximate-Cross-Sampling algorithm, called BP-Approximate-Cross-Sampling, has been presented.

The BP-Approximate-Cross-Sampling algorithm[1] uses a representation of the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ as lists of $qm$ bits, $\bar{D}_j$ and $\bar{D}'_j$ respectively, where $m$ is the length of the pattern and $q = \log(\lfloor m/2 \rfloor + 1) + 1$. If $(i,k) \in \bar{\mathcal{S}}_j$, where $0 \leq i < m$ 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 correspond to the value $k$ (we need exactly $q$ bits to represent a value between 0 and $\lfloor m/2 \rfloor$). The same considerations hold for the sets $\bar{\mathcal{S}}'_j$. Notice that if $mq \leq w$, each list fits completely in a single computer word, whereas if $mq > w$ one needs $\lceil mq/w \rceil$ computer words to represent each of the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$.

For each character $c$ of the alphabet $\Sigma$, the algorithm maintains a bit mask $M[c]$, where the rightmost bit of the $i$-th block is set to 1 if $P[i] = c$, and a bit mask $B[c]$, whose $i$-th block have all its bits set to 1 if $P[i] = c$.

The algorithm also maintains two bit vectors, $\bar{D}$ and $\bar{D}'$, whose configurations during the computation are respectively denoted by $\bar{D}_j$ and $\bar{D}'_j$, as the location $j$ advances over the input text. For convenience, we introduce also the bit vectors $\bar{D}_{-1}$ and $\bar{D}'_{-1}$, which are both set to $0^{qm}$. While scanning the text from left to right, the algorithm computes for each position $j \geq 0$ the bit vector $\bar{D}_j$ in terms of $\bar{D}_{j-1}$ and $\bar{D}'_{j-1}$, by performing the following bitwise operations (in brackets the corresponding operations on the set $\bar{\mathcal{S}}_j$ represented by $\bar{D}_j$):

[1] Here we provide some minor corrections to the code of the BP-Approximate-Cross-Sampling algorithm presented in [9].

```
(A) APPROXIMATE-CROSS-SAMPLING $(P, m, T, n)$
1.  $\bar{S}_0^0 \leftarrow \bar{S}_0^1 \leftarrow \emptyset$
2.  if $P[0] = T[0]$ then $\bar{S}_0^0 \leftarrow \{(0,0)\}$
3.  if $P[0] = T[1]$ then $\bar{S}_0^1 \leftarrow \{(0,0)\}$
4.  for $j \leftarrow 1$ to $n-1$ do
5.     $\bar{S}_j^0 \leftarrow \bar{S}_j^1 \leftarrow \emptyset$
6.     for $(i,k) \in \bar{S}_{j-1}^0$ do
7.        if $i < m-1$ then
8.           if $P[i+1] = T[j]$
9.              then $\bar{S}_j^0 \leftarrow \bar{S}_j^0 \cup \{(i+1,k)\}$
10.          if $j < n-1$ and $P[i+1] = T[j+1]$
11.             then $\bar{S}_j^1 \leftarrow \bar{S}_j^1 \cup \{(i+1,k)\}$
12.       else Output$((j-1,k))$
13.    for $(i,k) \in \bar{S}_{j-1}^1$ do
14.       if $i < m-1$ and $P[i+1] = T[j-1]$
15.          then $\bar{S}_j^0 \leftarrow \bar{S}_j^0 \cup \{(i+1,k+1)\}$
16.    if $P[0] = T[j]$ then $\bar{S}_j^0 \leftarrow \bar{S}_j^0 \cup \{(0,0)\}$
17.    if $j < n-1$ and $P[0] = T[j+1]$
18.       then $\bar{S}_j^1 \leftarrow \bar{S}_j^1 \cup \{(0,0)\}$
19. for $(i,k) \in \bar{S}_{n-1}^0$ do
20.    if $i = m-1$ then Output$(n-1,k)$
```

```
(B) BP-APPROXIMATE-CROSS-SAMPLING $(P, m, T, n)$
1.   $q \leftarrow \log(\lfloor m/2 \rfloor + 1) + 1$
2.   $F \leftarrow 0^{qm-1}1$
3.   $G \leftarrow 0^{q(m-1)}1^q$
4.   for $c \in \Sigma$ do
5.        $M[c] \leftarrow 0^{qm}$
6.        $B[c] \leftarrow 0^{qm}$
7.   for $i \leftarrow 0$ to $m-1$ do
8.        $M[P[i]] \leftarrow M[P[i]] \mid F$
9.        $B[P[i]] \leftarrow B[P[i]] \mid G$
10.       $F \leftarrow (F \ll q)$
11.       $G \leftarrow (G \ll q)$
12.  $F \leftarrow 0^{q-1}10^{q(m-1)}$
13.  $\bar{D} \leftarrow \bar{D}' \leftarrow 0^{qm}$
14.  for $j \leftarrow 0$ to $n-1$ do
15.       $H^0 \leftarrow (\bar{D} \ll q) \mid 1$
16.       $H^1 \leftarrow (\bar{D}' \ll q) \;\&\; B[T[j-1]$
17.       $H^2 \leftarrow (\bar{D}' \ll q) \;\&\; M[T[j-1]$
18.       $\bar{D} \leftarrow (H^0 \;\&\; B[T[j]]) \mid H^1$
19.       $\bar{D} \leftarrow \bar{D} \;+\; (H^2 \ll 1)$
20.       $\bar{D}' \leftarrow (H^0 \;\&\; B[T[j+1]]) \;\&\; \sim \bar{D}$
21.       if $(\bar{D} \;\&\; F) \neq 0^{qm}$ then
22.            $k \leftarrow (\bar{D} \gg (q(m-1)+1))$
23.            Output$(j,k)$
```

**Figure 2. (A)** The APPROXIMATE-CROSS-SAMPLING algorithm for the approximate swap matching problem. **(B)** Its bit-parallel variant BP-APPROXIMATE-CROSS-SAMPLING.

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

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

Similarly, the bit vector $\bar{D}'_j$ is computed in the $j$-th iteration of the algorithm in terms of $\bar{D}_{j-1}$, by performing the following bitwise operations (in brackets the corresponding operations on the set $\bar{\mathcal{S}}'_j$ represented by $\bar{D}'_j$):

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

During the $j$-th iteration, if the rightmost bit of the $(m-1)$-st block of $\bar{D}_j$ is set to 1, i.e. if $(\bar{D}_j \;\&\; 10^{q(m-1)}) \neq 0^m$, a swap match is reported at position $j$. The total number of swaps is contained in the $q-1$ leftmost bits of the $(m-1)$-st block of $\bar{D}_j$, which can be retrieved by performing a bitwise shift on $\bar{D}_j$ of $(q(m-1)+1)$ positions to the right.

The code of the BP-APPROXIMATE-CROSS-SAMPLING algorithm is shown in Fig. 2(B). It achieves a $\mathcal{O}(\lceil mn \log m/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m \log m/w \rceil)$ extra space, where $\sigma$ is the size of the alphabet. If $m(\log(\lfloor m/2 \rfloor + 1) + 1) \leq c_1 w$, where $c_1$ is a small integer constant, then the algorithm requires $\mathcal{O}(n)$-time and $\mathcal{O}(\sigma)$ extra space.

## 4 New Algorithms for the Approximate Swap Matching Problem

In this section we present a new practical algorithm for solving the swap matching problem, called Approximate-BCS (Approximate Backward Cross Sampling), which is characterized by a $\mathcal{O}(mn^2)$-time and $\mathcal{O}(m)$-space complexity, where $m$ and $n$ are the length of the pattern and text, respectively.

Our algorithm is an extension of the Backward-Cross-Sampling algorithm [8], for the standard swap matching problem. It inherits from the Approximate-Cross-Sampling algorithm the same doubly crossed structure in its iterative computation, but searches for all occurrences of the pattern in the text by scanning characters backwards, from right to left.

Later, in Section 4.2, we present an efficient implementation based on bit parallelism of the Approximate-BCS algorithm, which achieves 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 The Approximate-BCS Algorithm

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 right-to-left scans in windows of size $m$, as in the Backward DAWG Matching (BDM) algorithm for the exact single pattern matching problem [10]. In addition, for each occurrence of $P$ in $T$, the algorithm counts the number of swaps necessary to convert the pattern in its swapped occurrence.

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 Approximate-BCS 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, a 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 it with the longest prefix matched in the previous attempt.

To this end, if we put

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

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

for $0 \leq j < n$ and $0 \leq h \leq m$, then the following recurrences hold:

$$\begin{aligned}\mathcal{S}_j^{h+1} &= \{h-1 \leq i \leq m-1 \mid (i \in \mathcal{S}_j^h \text{ and } P[i-h] = T[j-h]) \text{ or} \\ &\qquad\qquad (i \in \mathcal{W}_j^h \text{ and } P[i-h] = T[j-h+1])\} \\ \mathcal{W}_j^{h+1} &= \{h \leq i \leq m-1 \mid i \in \mathcal{S}_j^h \text{ and } P[i-h] = T[j-h-1]\}.\end{aligned} \tag{2}$$

where the base cases are given by

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

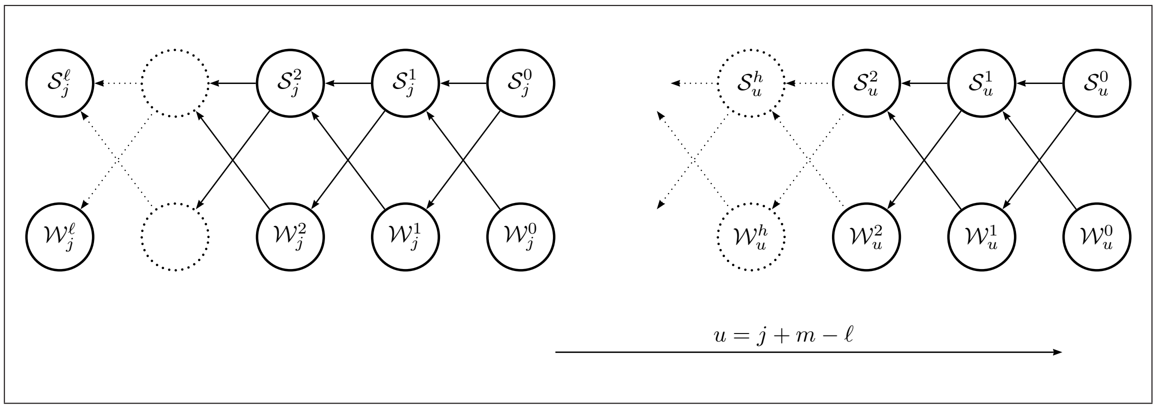

**Figure 3.** A graphic representation of the iterative fashion for computing the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ for increasing values of $h$. A first attempt starts at position $j$ of the text and stops with $h = \ell$. The subsequent attempt starts at position $u = j + m - l$.

Such relations allow one to compute the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ in an iterative fashion, where $\mathcal{S}_j^{h+1}$ is computed in terms of both $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$, whereas $\mathcal{W}_j^{h+1}$ needs only $\mathcal{S}_j^h$ for its computation. The resulting dependency graph has a doubly crossed structure as shown in Figure 3.

Plainly the set $S_j^h$ includes all the values $i$ such that the $h$-substring of $P$ ending at position $i$ has a swapped occurrence ending at position $j$ in $T$. Thus, if $(h-1) \in S_j^h$, then there is a swapped occurrence of the prefix of length $h$ of $P$. Hence, it follows that $P$ has a swapped occurrence ending at position $j$ if and only if $(m-1) \in S_j^m$.

Observe however that the only prefix of length $m$ is the pattern $P$ itself. Thus $(m-1) \in S_j^m$ if and only if $S_j^m \neq \emptyset$.

The following result follows immediately from (2).

**Lemma 8.** *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 $i \in S_j^\gamma$, then it follows that $i \in (S_j^h \cup W_j^h)$, for $1 \leq h \leq \gamma$.* □

**Lemma 9.** *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 $i \in (S_j^\gamma \cap W_j^{\gamma-1} \cap S_j^{\gamma-1})$, we have $P[i-\gamma+1] = P[i-\gamma+2]$.*

*Proof.* From $i \in (S_j^\gamma \cap S_j^{\gamma-1})$ it follows that $P[i-\gamma+1] = T[j-\gamma+1]$. Also, from $i \in W_j^{\gamma-1}$ it follows that $P[i-\gamma+2] = T[j-\gamma+1]$. Thus $P[i-\gamma+1] = P[i-\gamma+2]$. □

The following lemma will be used.

**Lemma 10.** *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 $i \in S_j^\gamma$, there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $i \in (S_j^\gamma \setminus S_j^{\gamma-1})$.*

*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 $i \in S_j^\gamma$ and that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We proceed by contradiction to prove that $i \notin S_j^{\gamma-1}$. Thus, we have

(i) $i \in S_j^\gamma$ (by hypothesis)
(ii) $P[i-\gamma+2] = T[j-\gamma+1] \neq P[i-\gamma+1]$ (by hypothesis)
(iii) $i \in S_j^{\gamma-1}$ (by contradiction)
(iv) $i \notin W_j^{\gamma-1}$ (by (ii), (iii), and Lemma 9)
(v) $P[i-\gamma+1] = T[j-\gamma+1]$ (by (i) and (iv))

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

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

(i) $i \in S_j^\gamma$ and $i \notin S_j^{\gamma-1}$ (by hypothesis)
(ii) $i \in W_j^{\gamma-1}$ (by (i) and Lemma 8)
(iii) $i \in S_j^{\gamma-2}$ (by (ii) and (2))
(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 10 and 8.

**Corollary 11.** *Let $P$ and $T$ be strings of length $m$ and $n$, respectively, over a common alphabet $\Sigma$. 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*

$$(m-1) \in S_j^m \quad and \quad |\Delta_j| = k\,,$$

*where $\Delta_j = \{1 \leq h < m \ :\ (m-1) \in (S_j^{h+1} \setminus S_j^h)\}$.* □

In consideration of the preceding corollary, the Approximate-BCS algorithm maintains a counter which is incremented every time $(m-1) \in (S_j^{h+1} \setminus S_j^h)$, 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 $\ell$ 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 $\ell$ 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 $m - \ell > 0$.

Moreover, in order to avoid accessing the text character at position $j-h+1 = n$, when $j = n-1$ and $h = 0$, the algorithm benefits of the introduction of a sentinel character at the end of the text.

The code of the Approximate-BCS algorithm is shown in Figure 4(A). Its time complexity is $\mathcal{O}(nm^2)$ in the worst case and requires $\mathcal{O}(m)$ extra space to represent the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$.

## 4.2 The Approximate-BPBCS Algorithm

In [8], an efficient bit-parallel implementation of the Backward-Cross-Sampling algorithm, called BP-Backward-Cross-Sampling, has also been presented. In this section we illustrate a practical bit-parallel implementation of the Approximate-BCS algorithm, named Approximate-BPBCS, along the same lines of the BP-Backward-Cross-Sampling algorithm.

In the Approximate-BPBCS 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, where $m$ is the length of the pattern.

The $(i-h+1)$-th bit of $D_j^h$ is set to 1 if $i \in \mathcal{S}_j$, i.e., if $P[i-h+1 \,..\, i] \propto T_j$, whereas the $(i-h+1)$-th 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 $\Sigma$, the algorithm maintains a bit mask $M[c]$ whose $i$-th bit is set to 1 if $P[i] = c$.

As in the Approximate-BCS algorithm, the text is processed in windows of size $m$, identified by their 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 $M[T[j]] \mid (M[T[j+1]] \& (M[T[j]] \ll 1))$ and $M[T[j-1]]$, respectively, according to the recurrences (2) and relative base cases. 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:

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

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 to add 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:

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

Similarly, the bit vector $C_j^{h+1}$ is computed in terms of $D_j^h$, 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 to add 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 to remove all elements $i$ such $P[i] \neq T[j-h-1]$ holds (by a bitwise and with the mask $M[T[j-h-1]]$), or, more formally:

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

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

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

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

```
(B) APPROXIMATE-BPBCS $(P, m, T, n)$
$F \leftarrow 10^{m-1}$
for $c \in \Sigma$ do $M[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m-1$ do
    $M[P[i]] \leftarrow M[P[i]] \mid F$
    $F \leftarrow F \gg 1$
$T[n] \leftarrow P[0]$
$j \leftarrow m - 1$
$F \leftarrow 10^{m-1}$
while $j < n$ do
    $h \leftarrow 1$, $\ell \leftarrow 0$
    $D \leftarrow M[T[j]]$
    $D \leftarrow D \mid (M[T[j+1]] \& (M[T[j]] \ll 1))$
    $C \leftarrow M[T[j-1]]$
    $c \leftarrow 0$
    while $h < m$ and $(D \mid C) \neq 0$ do
        $D' \leftarrow D \ll 1$
        if $F \& D \neq 0$ then $\ell \leftarrow h$
        $H \leftarrow (C \ll 1) \ \& \ M[T[j-h+1]]$
        $C \leftarrow D' \ \& \ M[T[j-h-1]]$
        $D \leftarrow D' \ \& \ M[T[j-h]]$
        $D \leftarrow D \mid H$
        if $(D \ \& \ \sim D') \ \& \ (1 \ll h) \neq 0$
            then $c \leftarrow c+1$
        $h \leftarrow h+1$
    if $D \neq 0$ then Output$(j,c)$
    $j \leftarrow j + m - \ell$
```

**Figure 4. (A)** The APPROXIMATE-BCS algorithm for the approximate swap matching problem. **(B)** Its bit-parallel variant APPROXIMATE-BPBCS.

As in the APPROXIMATE-BCS 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 $j$ of the text, has been found. Thus, a safe shift of $m - \ell$ position to the right can take place.

In practice, two vectors only are enough to implement the sets $D_j^h$ and $C_j^h$, for $h = 0, 1, \ldots, m$, as one can transform the vector $D_j^h$ into the vector $D_j^{h+1}$ and the vector $C_j^h$ into the vector $C_j^{h+1}$, during the $h$-th iteration of the algorithm at a given location $j$ of the text.

The counter for taking note 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 $\sigma$ 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.

## 5 Experimental Results

Next we report and comment experimental results relative to an extensive comparison under various conditions of the following approximate swap matching algorithms:

- Approximate-Cross-Sampling (ACS)
- BP-Approximate-Cross-Sampling (BPACS)
- Approximate-BCS (ABCS)
- Approximate-BPBCS (BPABCS)
- Iliopoulos-Rahman algorithm with a naive check of the swaps (IR&C)
- BP-Backward-Cross-Sampling algorithm with a naive check of the swaps (BPBCS&C)

We have chosen to include in our comparison also the algorithms IR&C and BP-BCS&C, since the algorithms IR and BPBCS turned out, in [8], to be the most efficient solutions for the swap matching problem. Instead, 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.

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 AMD Turion 64 X2 processor with mobile technology TL-60 of 2 GHz and a RAM memory of 4 GB. In particular, all algorithms have been tested on six Rand$\sigma$ problems, for $\sigma = 4, 8, 16, 32, 64$ and 128, on a genome, on a protein sequence, and on a natural language text buffer, with patterns of length $m$ = 4, 8, 12, 16, 20, 24, 28, 32.

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

### Running Times for Random Problems

In the case of random texts, all algorithms have been tested on six Rand$\sigma$ problems. Each Rand$\sigma$ problem consists in searching a set of 100 random patterns for any given length value in a 4 Mb random text over a common alphabet of size $\sigma$, with a uniform character distribution.

Running times for a Rand4 problem

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 5.916 | 5.768 | 5.835 | 5.860 | 5.753 | 5.739 | 5.571 | 5.604 |
| ABCS | 17.132 | 10.681 | 8.504 | 7.278 | 6.322 | 6.096 | 5.778 | 5.341 |
| BPACS | 0.817 | 0.794 | 0.752 | 0.800 | 0.784 | 0.799 | 0.818 | 0.747 |
| BPABCS | 0.573 | 0.341 | **0.255** | **0.204** | **0.177** | **0.159** | **0.141** | **0.129** |
| IR&C | **0.275** | **0.275** | 0.275 | 0.276 | 0.275 | 0.279 | 0.276 | 0.282 |
| BPBCS&C | 0.614 | 0.358 | 0.262 | 0.212 | 0.182 | 0.161 | 0.145 | 0.132 |

Running times for a Rand8 problem

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 4.769 | 4.756 | 4.762 | 4.786 | 4.761 | 4.808 | 4.765 | 4.796 |
| ABCS | 11.675 | 7.273 | 5.632 | 4.736 | 4.167 | 3.782 | 3.511 | 3.305 |
| BPACS | 0.832 | 0.830 | 0.828 | 0.831 | 0.830 | 0.829 | 0.827 | 0.827 |
| BPABCS | 0.413 | **0.229** | **0.175** | **0.145** | **0.127** | **0.114** | **0.104** | **0.096** |
| IR&C | **0.282** | 0.279 | 0.279 | 0.277 | 0.280 | 0.279 | 0.283 | 0.285 |
| BPBCS&C | 0.388 | 0.249 | 0.193 | 0.157 | 0.141 | 0.121 | 0.111 | 0.101 |

Running times for a Rand16 problem

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 5.210 | 5.291 | 5.162 | 5.282 | 5.198 | 5.201 | 5.202 | 5.131 |
| ABCS | 10.200 | 6.314 | 5.297 | 4.554 | 3.932 | 3.511 | 3.448 | 3.140 |
| BPACS | 0.786 | 0.807 | 0.780 | 0.783 | 0.812 | 0.806 | 0.743 | 0.721 |
| BPABCS | 0.346 | **0.198** | **0.144** | **0.118** | **0.103** | **0.093** | **0.086** | **0.081** |
| IR&C | **0.275** | 0.274 | 0.279 | 0.274 | 0.275 | 0.277 | 0.279 | 0.274 |
| BPBCS&C | 0.330 | 0.211 | 0.155 | 0.126 | 0.110 | 0.099 | 0.091 | 0.085 |

Running times for a Rand32 problem

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 5.285 | 5.080 | 5.228 | 5.262 | 5.175 | 5.190 | 5.216 | 5.296 |
| ABCS | 9.414 | 5.831 | 4.437 | 3.955 | 3.521 | 3.232 | 2.954 | 2.890 |
| BPACS | 0.776 | 0.746 | 0.796 | 0.775 | 0.834 | 0.807 | 0.791 | 0.796 |
| BPABCS | 0.294 | **0.184** | **0.138** | **0.113** | **0.097** | **0.086** | **0.078** | **0.073** |
| IR&C | **0.275** | 0.276 | 0.276 | 0.276 | 0.279 | 0.275 | 0.275 | 0.277 |
| BPBCS&C | 0.285 | 0.191 | 0.146 | 0.119 | 0.103 | 0.091 | 0.083 | 0.077 |

Running times for a Rand64 problem

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 5.101 | 5.108 | 5.254 | 5.174 | 5.155 | 5.098 | 5.095 | 5.262 |
| ABCS | 8.857 | 5.350 | 4.165 | 3.502 | 3.273 | 2.972 | 2.717 | 2.692 |
| BPACS | 0.838 | 0.808 | 0.769 | 0.714 | 0.835 | 0.806 | 0.807 | 0.766 |
| BPABCS | 0.267 | **0.162** | **0.127** | **0.108** | **0.095** | **0.086** | **0.078** | **0.073** |
| IR&C | 0.272 | 0.276 | 0.275 | 0.280 | 0.281 | 0.283 | 0.279 | 0.279 |
| BPBCS&C | **0.255** | 0.165 | 0.130 | 0.111 | 0.098 | 0.089 | 0.082 | 0.076 |

Running times for a Rand128 problem

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 5.070 | 5.052 | 5.091 | 4.996 | 5.088 | 4.940 | 4.968 | 5.216 |
| ABCS | 8.672 | 5.288 | 3.994 | 3.289 | 2.941 | 2.778 | 2.660 | 2.523 |
| BPACS | 0.833 | 0.836 | 0.836 | 0.836 | 0.835 | 0.835 | 0.836 | 0.833 |
| BPABCS | 0.248 | **0.148** | **0.115** | **0.098** | **0.087** | **0.080** | **0.075** | **0.071** |
| IR&C | 0.352 | 0.354 | 0.354 | 0.353 | 0.353 | 0.353 | 0.354 | 0.333 |
| BPBCS&C | **0.230** | 0.151 | 0.117 | 0.099 | 0.090 | 0.082 | 0.077 | 0.072 |

The experimental results show that the BPABCS 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&C algorithm. In the case of very short patterns and large alphabets, our algorithm is second only to the BPBCS&C algorithm. In addition we notice that the algorithms IR&C, ACS, and BPACS show a linear behavior, whereas the algorithms ABCS and BPABCS are characterized by a decreasing trend.

## Running Times for Real World Problems

The tests on real world problems have been performed on a genome sequence and on a natural language text buffer. The genome we used for the tests is a sequence of $4,638,690$ base pairs of *Escherichia coli* taken from the file E.coli of the Large Canterbury Corpus.[1] The tests on the protein sequence have been performed using a 2.4 Mb file containing a protein sequence from the human genome with 22 different characters. The experiments on the natural language text buffer have been done with the file world192.txt (The CIA World Fact Book) of the Large Canterbury Corpus. The file contains $2,473,400$ characters drawn from an alphabet of 93 different characters.

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

Running times for a genome segence ($\sigma = 4$)

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 5.629 | 5.643 | 5.654 | 5.636 | 5.644 | 5.640 | 5.647 | 6.043 |
| ABCS | 18.018 | 11.261 | 8.805 | 7.523 | 6.700 | 6.117 | 5.710 | 5.359 |
| BPACS | 0.950 | 0.914 | 0.917 | 0.766 | 0.874 | 0.934 | 0.935 | 0.843 |
| BPABCS | 0.647 | **0.318** | **0.266** | **0.232** | **0.195** | **0.174** | **0.160** | 0.147 |
| IR&C | **0.262** | 0.287 | 0.314 | 0.311 | 0.311 | 0.311 | 0.310 | 0.311 |
| BPBCS&C | 0.678 | 0.367 | 0.290 | 0.233 | 0.204 | 0.176 | **0.160** | **0.146** |

Running times for a protein sequence ($\sigma = 22$)

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 3.777 | 3.784 | 3.671 | 3.729 | 3.766 | 3.703 | 3.716 | 3.741 |
| ABCS | 7.045 | 4.557 | 3.734 | 3.162 | 2.806 | 2.661 | 2.600 | 2.351 |
| BPACS | 0.565 | 0.581 | 0.561 | 0.563 | 0.584 | 0.580 | 0.534 | 0.519 |
| BPABCS | 0.249 | **0.142** | **0.103** | **0.084** | **0.074** | **0.066** | **0.061** | **0.058** |
| IR&C | 0.388 | 0.390 | 0.391 | 0.389 | 0.391 | 0.391 | 0.396 | 0.389 |
| BPBCS&C | **0.241** | 0.145 | 0.107 | 0.087 | 0.075 | 0.068 | 0.062 | **0.058** |

Running times for a natural language text buffer ($\sigma = 93$)

| $m$ | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| ACS | 3.170 | 2.757 | 2.748 | 2.756 | 2.761 | 2.745 | 2.746 | 2.754 |
| ABCS | 6.175 | 4.054 | 3.164 | 2.705 | 2.306 | 2.288 | 2.042 | 1.866 |
| BPACS | 0.492 | 0.497 | 0.492 | 0.491 | 0.492 | 0.491 | 0.494 | 0.493 |
| BPABCS | 0.194 | **0.114** | **0.086** | **0.071** | **0.062** | **0.056** | **0.051** | **0.049** |
| IR&C | 0.171 | 0.165 | 0.164 | 0.168 | 0.165 | 0.165 | 0.165 | 0.167 |
| BPBCS&C | **0.164** | 0.126 | 0.094 | 0.076 | 0.070 | 0.059 | 0.056 | 0.055 |

From the above experimental results, 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&C (for the genome sequence) and to BPBCS&C (for the protein sequence and the natual language text buffer).

## 6 Conclusions

In this paper we have presented new efficient algorithms for the Approximate Swap Matching problem. In particular, we have devised an extension of the Backward-Cross-Sampling general algorithm, named Approximate-BCS, and of its bit-parallel implementation BP-Backward-Cross-Sampling, named Approximate-BPBCS.

The Approximate-BCS algorithm achieves a $\mathcal{O}(nm^2)$-time complexity and requires $\mathcal{O}(nm)$ additional space, whereas the Approximate-BPBCS algorithm achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and, when the pattern fits in few machine words, finds all swapped matches and their corresponding counts in $\mathcal{O}(nm)$-time.

In contrast with the BP-Approximate-Cross-Sampling algorithm, the Approximate-BPBCS algorithm requires $\mathcal{O}(\sigma\lceil m/w \rceil + \log(\lfloor m/2 \rfloor + 1))$ extra space and is thus preferable to the former in the case of longer patterns.

From an extensive experimentation, it turns out that the Approximate-BPBCS algorithm is very fast in practice and obtains the best results in most cases, being second only to algorithms based on a naive check of the number of swaps in the case of very short patterns.

## 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, 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.* Information Processing Letters, 68(3) 1998, pp. 125–132.
5. A. Amir, M. Lewenstein, and E. Porat: *Approximate swapped matching.* Inf. Process. Lett., 83(1) 2002, pp. 33–39.
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, vol. 13 of Communications in Computer and Information Science, Springer, 2008, pp. 446–455.
7. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
8. 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, Lecture Notes in Computer Science, Springer, 2009.
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. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
11. 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.
12. S. Muthukrishnan: *New results and open problems related to non-standard stringology*, in Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, vol. 937 of Lecture Notes in Computer Science, Springer, 1995, pp. 298–317.