A New Algorithm for Efficient Pattern Matching with Swaps

Matteo Campanelli[1], Domenico Cantone[2], and Simone Faro[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@dmi.unict.it, faro@dmi.unict.it

Abstract. The Pattern Matching problem with Swaps consists in finding all occurrences of a pattern P in a text T, when disjoint local swaps in the pattern are allowed.
In this paper, we present a new efficient algorithm for the Swap Matching problem with short patterns. In particular, we devise a $\mathcal{O}(nm^2)$ general algorithm, named BACKWARD-CROSS-SAMPLING, and show an efficient implementation of it, based on bit-parallelism, which achieves $\mathcal{O}(nm)$ worst-case time and $\mathcal{O}(\sigma)$-space complexity, with patterns whose length m is comparable to the word-size of the target machine (n and σ are respectively the size of the text and of the alphabet).
From an extensive comparison with some of the most recent and effective algorithms for the swap matching problem, it turns out that our algorithm is very flexible and achieves very good results in practice.
Key words: 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 character set Σ of size σ. More precisely, the pattern is said to *swap-match the text at a given location* j if adjacent pattern characters can be swapped, if necessary, so as to make it identical to the substring of the text ending (or, equivalently, starting) at location j. All swaps are constrained to be disjoint, i.e., each character can be involved in at most one swap. Moreover, 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, and network security, and many others. Following [5], 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 [10]. The first nontrivial result was reported by Amir *et al.* [1], who provided a $\mathcal{O}(nm^{\frac{1}{3}} \log m)$-time 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 [9]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on bit parallelism, 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 [7] presented a first approach for solving the swap matching problem with short patterns in linear time. More precisely, they devised a simple algorithm, named Cross-Sampling, which, though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient implementation based on bit-parallelism, achieving $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(\sigma)$ space complexity for short patterns fitting in few machine words.

In this paper, we present a new efficient algorithm for solving the swap matching problem. In particular, we provide a $\mathcal{O}(nm^2)$ general algorithm, named Backward-Cross-Sampling algorithm, which inherits much the same iterative structure of the Cross-Sampling algorithm, but is based on a right-to-left scan of the text, giving better results in practice. We will also describe an efficient implementation of the algorithm, characterized by a $\mathcal{O}(nm)$ worst-case time and $\mathcal{O}(\sigma)$-space complexity, for patterns of length comparable 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. Sections 3 describes the Cross-Sampling algorithm and its bit-parallel variant. In Section 4 we present the Backward-Cross-Sampling algorithm for the swap matching problem and then, in Section 5, we illustrate an efficient implementation of it based on bit-parallelism. Results of an extensive experimental comparison under various conditions with the most efficient algorithms present in the literature are reported in Section 6. Finally, we will briefly draw our conclusions in Section 7.

2 Notions and Basic Definitions

A string P of length $m \geq 0$ is represented as a finite array $P[0 \mathinner{..} m-1]$. In such a case we also write length$(P) = m$. In particular, for $m = 0$ we obtain the empty string, denoted by ε. We denote by $P[i]$ the $(i+1)$-st character of P, for $0 \leq i <$ length(P). Likewise, we denote by $P[i \mathinner{..} j]$ 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 h-substring of a string S is a substring of S of length h. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i \mathinner{..} \text{length}(P)-1]$, for some $0 \leq i <$ length(P). Similarly, we say that P' is a prefix of P if $P' = P[0 \mathinner{..} i-1]$, for some $0 \leq i \leq$ length(P). We denote by P_i the nonempty prefix $P[0 \mathinner{..} i]$ of P of length $i+1$, for $0 \leq i < m$. If $i < 0$, we convene 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 $\pi : \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$ 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 are not swapped).

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

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., $\pi(P) = T[j-m+1 \mathinner{..} j]$. In such a case we write $P \propto T_j$.*

Definition 3 (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., $P \propto T_j$.*

The following elementary result will be used later.

Lemma 1 ([7]). *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 1. *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 The Cross-Sampling Algorithm

The Cross-Sampling algorithm [7] 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. More precisely, during its $(j+1)$-st iteration, for $j = 0, 1, \ldots, n-1$, we establish whether $P_i \propto T_j$, for each $i = 0, 1, \ldots, m-1$, by exploiting information gathered during previous iterations. To this end, if we put

$$\begin{aligned} \lambda_j &= \begin{cases} \{0\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise}, \end{cases} && \text{for } 0 \le j \le n-1 \\ \mathcal{S}_j &= \{0 \le i \le m-1 \ : \ P_i \propto T_j\}, && \text{for } 0 \le j \le n-1 \\ \mathcal{S}'_j &= \{0 \le i < m-1 \ : \ P_{i-1} \propto T_{j-1} \ \text{ and } \ P[i] = T[j+1]\} && \text{for } 1 \le j \le n-1. \end{aligned}$$

then the following recurrences hold:

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

where the base cases are given by $\mathcal{S}_0 = \lambda_0$ and $\mathcal{S}'_0 = \lambda_1$.

Such relations allow one to compute the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$ in an iterative fashion, where $\mathcal{S}_{j+1}$ 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. Plainly, the time complexity of the Cross-Sampling algorithm is $\mathcal{O}(nm)$.

[7] presents also an efficient implementation of the Cross-Sampling algorithm based on the bit-parallelism technique [6], called BP-Cross-Sampling algorithm. We recall that 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 BP-Cross-Sampling algorithm uses a representation of the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$ as lists of m bits, D_j and D'_j respectively (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 $P[i] = T[j+1]$. The remaining bits are set to 0. Notice that if $m \le w$, each list fits completely in a single computer word, whereas if $m > w$ we need $\lceil m/w \rceil$ computer words to represent each of the sets $\mathcal{S}_j$ and $\mathcal{S}'_j$.

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 $P[i] = c$.

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

```
(A) CROSS-SAMPLING (P, m, T, n)
S_0 ← S'_0 ← ∅
if P[0] = T[0] then S_0 ← {0}
if P[0] = T[1] then S'_0 ← {0}
for j ← 1 to n − 1 do
   S_j ← S'_j ← ∅
   for i ∈ S_{j−1} do
      if i < m − 1 then
         if P[i + 1] = T[j]
            then S_j ← S_j ∪ {i + 1}
         if j < n − 1 and P[i + 1] = T[j + 1]
            then S'_j ← S'_j ∪ {i + 1}
      else Output(j − 1)
   for i ∈ S'_{j−1} do
      if i < m − 1 and P[i + 1] = T[j − 1]
         then S_j ← S_j ∪ {i + 1}
   if P[0] = T[j] then S_j ← S_j ∪ {0}
   if j < n − 1 and P[0] = T[j + 1]
      then S'_j ← S'_j ∪ {0}
for i ∈ S_{j−1} do
   if i = m − 1 then Output(n − 1)
```

```
(B) BP-CROSS-SAMPLING (P, m, T, n)
F ← 0^{m−1}1
for c ∈ Σ do M[c] ← 0^m
for i ← 0 to m − 1 do
      M[x_i] ← M[P[i]] | F
      F ← F ≪ 1
F ← 10^{m−1}
D ← D' ← 0^m
for j ← 0 to n − 1 do
      H ← (D ≪ 1) | 1
      D ← (H & M[T[j]])
      D' ← (D' ≪ 1) & M[T[j − 1]]
      D ← D | D'
      D' ← H & M[T[j + 1]]
      if (D & F) ≠ 0^m then
            Output(j)
```

Fig. 1. (A) The CROSS-SAMPLING algorithm for solving the swap matching problem. **(B)** The BP-CROSS-SAMPLING algorithm based on bit-parallelism.

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

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

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

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

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

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

4 The Backward-Cross-Sampling Algorithm

In this section we present a new practical algorithm for solving the swap matching problem, called BACKWARD-CROSS-SAMPLING.

The new algorithm inherits from the CROSS-SAMPLING algorithm the same doubly crossed structure in 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 [8].

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, 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 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

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

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 2. *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*

- $P[i-h+2\,..\,i] \propto T_j$ *and* $P[i-h+1] = T[j-h+1]$;
- $P[i-h+3\,..\,i] \propto T_j$, $P[i-h+1] = T[j-h+2]$, *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

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

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

$$\mathcal{S}_j^0 = \{i \ : \ 0 \leq i < m\} \quad \text{and} \quad \mathcal{W}_j^0 = \{0 \leq i < m-1 \ : \ P[i+1] = T[j]\}. \qquad (2)$$

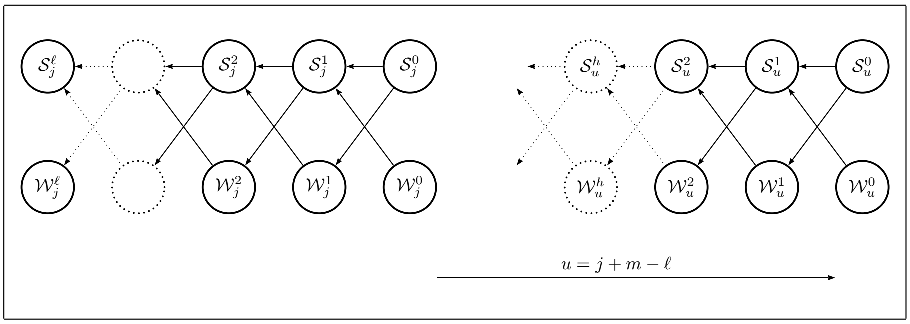

Fig. 2. A graphic representation of the iterative pattern for computing sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ for increasing values of h. A first attempt, starting at position j of the text, ends with $h = \ell$. The subsequent attempt starts at position $u = j + m - \ell$.

Additionally, Lemma 2 justifies the following recursive definitions of the sets $\mathcal{S}_j^{h+1}$ and $\mathcal{W}_j^{h+1}$ in terms of $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$, for $0 \le j < n$ and $0 \le h < m$:

$$\begin{aligned}\mathcal{S}_j^{h+1} &= \{h-1 \le i \le m-1 \;:\; (i \in \mathcal{S}_j^h \text{ and } P[i-h] = T[j-h]) \text{ or} \\ &\qquad\qquad\qquad\qquad\quad (i \in \mathcal{W}_j^h \text{ and } P[i-h] = T[j-h+1])\} \\ \mathcal{W}_j^{h+1} &= \{h \le i \le m-1 \;:\; i \in \mathcal{S}_j^h \text{ and } P[i-h] = T[j-h-1]\}\,.\end{aligned} \tag{3}$$

Such relations, coupled with the initial conditions (2), allow one to compute the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ in an iterative fashion as shown in Fig. 2.

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 $m - \ell > 0$.

Moreover, in order to avoid accessing the text character of 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.

To compute the worst-case time complexity of the algorithm, preliminarily we 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 1, for each position j of the

(A) Backward-Cross-Sampling (P, m, T, n)

```
T[n] ← P[0]
j ← m − 1
while j < n do
    h ← 0
    S_j^0 ← {i : 0 ≤ i < m}
    W_j^0 ← {0 ≤ i < m − 1 : P[i + 1] = T[j]}
    while h < m and S_j^h ∪ W_j^h ≠ ∅ do
        if (h − 1) ∈ S_j^h then ℓ ← h
        for each i ∈ S_j^h do
            if i ≥ h and P[i − h] = T[j − h]
                then S_j^{h+1} ← S_j^{h+1} ∪ {i}
            if i > h and P[i − h] = T[j − h − 1]
                then W_j^{h+1} ← W_j^{h+1} ∪ {i}
        for each i ∈ W_j^h do
            if i ≥ h and P[i − h] = T[j − h + 1]
                then S_j^{h+1} ← S_j^{h+1} ∪ {i}
        h ← h + 1
    if (h − 1) ∈ S_j^h then Output(j)
    j ← j + m − ℓ
```

(B) BP-Backward-Cross-Sampling (P, m, T, n)

```
F ← 10^{m−1}
for c ∈ Σ do M[c] ← 0^m
for i ← 0 to m − 1 do
    M[P[i]] ← M[P[i]] | F
    F ← F ≫ 1
T[n] ← P[0]
j ← m − 1
F ← 10^{m−1}
while j < n do
    h ← 1, ℓ ← 0
    D ← M[T[j]]
    D ← D | (M[T[j + 1]]&(M[T[j]] ≪ 1))
    C ← M[T[j − 1]]
    while h < m and (D | C) ≠ 0 do
        if F&D ≠ 0 then ℓ ← h
        H ← (C ≪ 1) & M[T[j − h + 1]]
        C ← (D ≪ 1) & M[T[j − h − 1]]
        D ← (D ≪ 1) & M[T[j − h]]
        D ← D | H
        h ← h + 1
    if D ≠ 0 then Output(j)
    j ← j + m − ℓ
```

Fig. 3. (A) The Backward-Cross-Sampling algorithm for the swap matching problem. **(B)** The BP-Backward-Cross-Sampling algorithm (based on bit-parallelism).

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 $\mathcal{W}_j^h$.

5 The BP-Backward-Cross-Sampling Algorithm

In this section we present a practical implementation of the Backward-Cross-Sampling algorithm based on the bit-parallelism technique [6]. The resulting algorithm works as the BNDM (Backward Nondeterministic DAWG Match) algorithm [11], 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 [6].

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 $(i-h+1)$-th 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 $(i-h+1)$-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 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 $M[T[j]] \mid (M[T[j+1]] \& (M[T[j]] \ll 1))$ and $M[T[j-1]]$, respectively, according to the base cases shown in (2) and recursive expressions shown in (3). 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 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, 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 $M[T[j-h-1]]$).

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 of the text, has been found. Thus a safe shift of $m - \ell$ position 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. 3(B). 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.

6 Experimental Results

Next we present experimental data which allow to compare under various conditions the following string matching algorithms in terms of their running times:
- Iliopoulos-Rahman algorithm (IR)
- Cross-Sampling algorithm (CS)
- BP-Cross-Sampling algorithm (BPCS)
- Backward-Cross-Sampling algorithm (BCS)
- 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.

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 three Randσ problems, for $\sigma = 8, 32$, 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 tables below, running times have been expressed in hundredths of seconds and the best results are bold-faced.

Running Times for Random Problems
In the case of random texts, the algorithms have been tested on three Randσ problems. 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.

Running times for a Rand8 problem

m	4	8	12	16	20	24	28	32
IR	**3.450**	3.420	3.420	3.440	3.580	3.560	3.520	3.560
CS	66.670	67.210	67.230	67.590	67.850	68.280	68.670	69.060
BPCS	3.960	3.900	3.890	3.900	3.920	3.900	3.930	3.910
BCS	62.130	41.160	33.700	29.480	26.750	24.870	23.700	22.450
BPBCS	4.140	**2.000**	**1.850**	**1.180**	**1.110**	**1.000**	**0.910**	**0.800**

Running times for a Rand32 problem

m	4	8	12	16	20	24	28	32
IR	2.920	2.950	2.930	2.940	2.940	2.930	2.950	2.950
CS	60.030	59.760	59.740	59.710	59.610	59.580	59.350	59.200
BPCS	3.030	3.050	3.040	3.080	3.040	3.060	3.080	3.060
BCS	46.200	29.050	23.750	20.540	18.640	17.380	16.180	15.660
BPBCS	**2.650**	**1.930**	**1.050**	**1.000**	**0.820**	**0.600**	**0.380**	**0.240**

Running times for a Rand128 problem

m	4	8	12	16	20	24	28	32
IR	3.550	3.610	3.610	3.590	3.630	3.650	3.660	3.640
CS	59.910	59.610	59.460	59.390	59.380	59.130	59.180	59.130
BPCS	3.120	3.150	3.160	3.160	3.140	3.110	3.130	3.130
BCS	42.720	25.750	20.130	17.720	15.950	14.650	13.990	13.310
BPBCS	**2.000**	**1.040**	**0.960**	**0.750**	**0.580**	**0.390**	**0.250**	**0.180**

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 IR, CS. and BPCS show a linear behavior, whereas 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.

Running Times for Real World Problems

The tests on real world problems have been performed on a genome sequence, on a protein sequence, and on a natural language text buffer. 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.[1] The protein sequence used in the tests is a 2.4Mb file with 22 different characters from the human genome. Finally, as natural language text buffer 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.

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

m	4	8	12	16	20	24	28	32
IR	**3.070**	**3.060**	3.070	3.080	3.100	3.150	3.150	3.100
CS	83.020	79.930	79.760	79.380	79.350	79.430	79.500	79.460
BPCS	6.820	3.950	3.910	3.920	3.930	3.920	3.930	3.940
BCS	102.410	67.010	55.480	49.050	45.250	42.290	40.260	38.650
BPBCS	10.170	3.930	**2.640**	**2.010**	**1.960**	**1.830**	**1.510**	**1.120**

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

m	4	8	12	16	20	24	28	32
IR	**1.990**	2.000	1.990	2.000	1.990	2.000	1.990	1.990
CS	45.190	45.230	45.490	45.650	45.900	46.040	46.400	44.400
BPCS	2.030	2.010	2.020	2.050	2.040	2.030	2.040	2.020
BCS	31.110	22.450	18.620	16.430	15.130	14.090	13.450	12.670
BPBCS	2.130	**1.180**	**0.950**	**0.590**	**0.270**	**0.120**	**0.070**	**0.070**

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

m	4	8	12	16	20	24	28	32
IR	**1.850**	1.820	1.820	1.850	1.880	1.820	1.860	1.850
CS	36.950	36.680	36.520	36.410	36.230	36.120	36.080	36.210
BPCS	2.050	1.970	1.970	1.990	1.970	1.980	1.990	1.980
BCS	30.410	19.390	15.720	13.640	12.350	11.430	10.820	10.320
BPBCS	2.000	**0.990**	**0.610**	**0.210**	**0.050**	**0.020**	**0.013**	**0.010**

The above experimental results 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.

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

7 Conclusions

In this paper we have presented a new efficient algorithm for the Swap Matching problem with short patterns. In particular, we have devised a $\mathcal{O}(nm^2)$ general algorithm, named Backward-Cross-Sampling, and have provided an efficient implementation of it, based on bit-parallelism.

An extensive experimental comparisons showed that our algorithm is very fast in practice and obtains the best results in most cases, especially with long patterns and large alphabets.

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. P. Antoniou, C.S. Iliopoulos, I. Jayasekera, and M.S. 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.
6. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
7. 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*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009.
8. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
9. C. S. Iliopoulos and M. Sohel 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.
10. 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.
11. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Proc. of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 14–33. Springer-Verlag, Berlin, 1998.