Patter Matching with Swaps in Linear Time for Short Patterns

Domenico Cantone and Simone Faro

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

Abstract. The Pattern Matching problem with Swaps consists in finding all occurrence of a pattern P in a text T allowing a series of local swaps in the pattern where all the swaps are constrained to be disjoint. In the Approximate Pattern Matching problem with Swaps the output is, for every text location where there is a swapped match of P, the number of swaps necessary to create the swapped version that matches such a location.
In this paper, we present a new approach for solving both Swap Matching and Approximate Swap Matching Problems in linear time for short patterns. In particular we devise an efficient general algorithm, named Cross-Sampling, with a $O(nm)$ and show how to obtain an efficient implementation, based on bit-parallelism, which achieves $O(n)$ worst case time and $O(\sigma)$ space complexity for patterns having length similar to the word-size of the target machine.

Key words: Design and analysis of algorithms, combinatorial algorithms on words, pattern matching, pattern matching with swaps, nonstandard pattern matching

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 of a pattern P of length m in a text T of length n, both being sequences of characters drawn from a finite character set Σ. The pattern is said to match the text at a given location j if adjacent pattern characters can be swapped, if necessary, so as to make the pattern identical to the substring of the text ending (or equivalently, starting) at location j. All the swaps are constrained to be disjoint, i.e., each character is involved in at most one swap. Moreover identical adjacent characters cannot be swapped.

This problem is interesting as a fundamental computer science problem and is a basic need of many practical applications such as text retrieval, music retrieval, computational biology, data mining, network security, among many others.

The problem was introduced in 1995 as one of the open problems in nonstandard string matching [Mut95]. However until a decade there were no known upper bounds better than the naive $O(nm)$ algorithm for the swap matching problem. The first nontrivial results on this problem was obtained by Amir et al [AAL$^+$97]. They showed that the case when the size of the alphabet set Σ exceeds 2 can be reduced to the case when it is exactly 2 with a time overhead of $O(\log^2 \sigma)$ (The reduction overhead was reduced to $O(\log \sigma)$ in the journal version [1]). They then showed how to solve the problem for alphabet sets of size 2 in time $O(nm^{\frac{1}{3}} \log m)$. Amir et al. [ALLL98] also give certain special cases for which $O(m \log^2 m)$ time can be obtained. However, these cases are rather restrictive. More recently, Amir et al. [ACH$^+$03] solved the Swap Matching problem in time $O(n \log m \log \sigma)$. All the above solutions to swap matching depend 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 techniques has been presented by Iliopoulos and Rahman in [IR08]. They presented a new graph-theoretic approach to model the problem devising an efficient algorithm, based on bit parallelism, which runs in $O((n+m) \log m)$ if the pattern is similar in size to the size of word 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 text location j (provided there is a swap match at j). In [ALP02] Amir et al. presented an algorithm that counts the number of swaps at every location where there is a swapped matching in time $O(n \log m \log \sigma)$. Consequently, the total time for solving the approximate pattern matching with swaps problem is $O(n \log m \log \sigma)$.

In this paper, we present a first approach for solving both Swap Matching and Approximate Swap Matching Problems in linear worst case time for short patterns. More precisely we devise a new simple algorithm to solve the problem, named Cross-Sampling, with a $O(nm)$ worst case time complexity. Then we show how to obtain an efficient implementation of the algorithm, based on bit-parallelism, which achieves $O(n)$ worst case time and $O(\sigma)$ 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 present some preliminary definitions. Section 3 presents the new Cross-Sampling algorithm for the swap matching and approximate swap matching problem. In Section 4, we use bit-parallelism to obtain efficient implementations of the Cross-Sampling algorithms. Finally, we briefly conclude in Section 5.

2 Notions and Basic Definitions

A string P is represented as a finite array $P[0 \mathinner{..} m-1]$, with $m \geq 0$. In such a case we say that P has length m and write length$(P) = m$. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < \text{length}(P)$. Likewise, by $P[i \mathinner{..} j]$ we denote the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < \text{length}(P)$. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i \mathinner{..} \mathrm{length}(P)-1]$, for some $0 \le i < \mathrm{length}(P)$. Similarly, we say that P' is a prefix of P if $P' = P[0 \mathinner{..} i-1]$, for some $0 \le i \le \mathrm{length}(P)$. We denote with symbol P_i the nonempty prefix of P of length $i+1$, with $0 \le i < m$. In addition, 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:*

1. *if $\pi(i) = j$ then $\pi(j) = i$ (characters are swapped).*
2. *for all i, $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters are swapped).*
3. *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 use $\pi(P)$ to denote the *swapped version* of P, where $\pi(P) = P[\pi(0)].P[\pi(1)]...P[\pi(m-1)]$.

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 of T if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j, i.e. $P[\pi(i)] = T[j-m+i+1]$, for $i = 0...m-1$. In such a case we write $P \propto T_j$.*

Observe that if we assume there is a swap match ending at location j of the text, then the number of swaps, k, needed to transform P in its swapped version $\pi(P) = T[j-m+1...j]$ is equal to half the number of mismatches of P at location i. Thus the value of k is in 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.*

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 $0 \le k \le \lfloor m/2 \rfloor$, such that P has a swapped occurrence in T at location j with k swaps.*

The following Lemma will be used later.

Lemma 1. *Let P and R be strings of length m over an alphabet Σ and suppose that exists a swap permutation π such that $\pi(P) = R$. Then π is unique.*

Proof. Suppose there exist two different permutations π and π' such that $\pi(P) = \pi'(P) = R$. Then exists an index i such that $\pi(i) \neq \pi'(i)$ but $P[\pi(i)] = P[\pi'(i)] = R[i]$. By Definition 1 $\pi(i), \pi'(i) \in \{i-1, i, i+1\}$. Without loss of generality we suppose $\pi(i) < \pi'(i)$ and suppose i is the smallest index such that $\pi(i) \neq \pi'(i)$ but $P[\pi(i)] = P[\pi'(i)]$. We can observe the following three different cases:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	b	b	a	b	a	b	a	a	b	b	a	b	a	a
P_0	-	0	0	-	0	-	0	-	-	0	0	-	0	-	-
P_1	-	1	-	0	1	0	1	0	-	1	-	0	1	0	-
P_2	-	-	1	1	0	-	0	-	-	-	1	1	0	-	-
P_3	-	-	-	1	-	0	2	0	-	1	-	1	-	0	-
P_4	-	-	-	-	-	-	-	2	0	-	-	-	-	-	0
P_5	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-
P_6	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-

Fig. 1. The matrix of swap occurrences of prefixes for a pattern $P = babaaab$ of length 7 and a text $T = abbababaabbabaa$ of length 15. A value k in row P_i and column j means that $P_i \propto_k T_j$, whereas a symbol - means that $P_i \not\propto T_j$

1. $\pi(i) = i - 1$ and $\pi'(i) = i$
   Then by Definition 1(1) we have $\pi(i-1) = i$. This implies $P[\pi(i-1)] = P[\pi'(i)] = P[\pi(i)] = P[i]$ which violates Definition 1(3).
2. $\pi(i) = i$ and $\pi'(i) = i + 1$
   Then by Definition 1(1) we have $\pi'(i+1) = i$. This implies $P[\pi'(i+1)] = P[\pi(i)] = P[\pi'(i)] = P[i]$ which violates Definition 1(3).
3. $\pi(i) = i - 1$ and $\pi'(i) = i + 1$
   By hypothesis we have $\pi(i-1) = \pi'(i+1) = i$. Thus $\pi'(i-1) \neq i = \pi(i-1)$. Moreover we have $P[\pi'(i-1)] = R[i-1] = P[i] = P[\pi(i-1)]$. which violates the hypothesis that i is the smallest index such that $\pi(i) \neq \pi'(i)$ but $P[\pi(i)] = P[\pi'(i)]$. ■

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

3 A New Approach to the Swap Matching Problem

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

Suppose T is a text o of length n, and P is a pattern of length m. Our algorithm solves the swap matching problem by computing the occurrences of all prefixes of the pattern in continuously increasing prefixes of the text. That is, during its first iteration the algorithm computes the occurrences of prefixes P_i such that $P_i \propto T_0$. Then, during the j-th iteration, it computes all occurrences of prefixes P_i such that $P_i \propto T_j$, using information gathered during previous iterations.

Fig. 1 shows the $n \times m$ matrix of swap occurrences of prefixes for a pattern $P = babaaab$ of length 7 and a text $T = abbababaabbabaa$ of length 15. Each row of the matrix is labeled with a prefix P_i of the pattern while columns are labeled with locations of the text. A value k in row P_i and column j means that $P_i \propto_k T_j$, whereas a symbol - means that $P_i \not\propto T_j$. Our algorithm computes the non null elements of the j-th column of the matrix for increasing values of j.

The following very elementary fact helps to define the algorithm's strategy.

Lemma 2. *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 either* $i = 0$, *or* $P_{i-1} \propto T_{j-1}$
- $P[i] = T[j-1]$, $P[i-1] = T[j]$ *and either* $i = 1$, *or* $P_{i-2} \propto T_{j-2}$ ■

To begin with, let S_j^0 denote the collection of all values i such that the prefix P_i of P has a swapped occurrence ending at position j of the text, for $0 \le j \le n$. Moreover let S_j^1 denote the collection of all values i such that the prefix P_{i-1} of P has a swapped occurrence ending at position $j-1$ of the text and $P[i] = T[j+1]$. Formally

$$\begin{aligned} \mathcal{S}_j^0 &= \{0 \le i \le m-1 \mid \ P_i \propto T_j\} \\ \mathcal{S}_j^1 &= \{0 \le i < m-1 \mid (P_{i-1} \propto T_{j-1} \vee i = 0) \ \text{ and } \ P[i] = T[j+1]\} \end{aligned} \tag{1}$$

Then the problem of finding the positions j in T such that $P \propto T_j$ translates to the problem of finding all values j such that $m-1 \in S_j^0$.

Consider the example shown in Fig. 1. The elements of set S_j^0 are non null values in the j-th column of the matrix. Thus $S_1^0 = \{0,1\}$, $S_2^0 = \{0,2\}$, $S_6^0 = \{0,1,2,3\}$, $S_8^0 = \{4,5\}$ and $S_9^0 = \{0,1,3,6\}$. Since $6 \in S_9^0$ we can report a swapped match at position 9 of the text.

To efficiently compute the sets defined above notice that the sets S_0^0 and S_0^1 can be defined as follows

$$\lambda_j = \begin{cases} \{0\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise} \end{cases}, \qquad S_0^0 = \lambda_0, \quad \text{and} \quad S_0^1 = \lambda_1. \tag{2}$$

Moreover Lemma 2 justifies the following recursive definitions of the sets S_{j+1}^0 and S_{j+1}^1 in terms of S_j^0 and S_j^1, for $0 < j < n$:

$$\begin{aligned} \mathcal{S}_{j+1}^0 &= \{i \le m-1 \mid \ ((i-1) \in S_j^0 \ \text{ and } \ P[i] = T[j+1]) \text{ or} \\ &\qquad\qquad\qquad\quad ((i-1) \in S_j^1 \ \text{ and } \ P[i] = T[j]) \ \} \cup \lambda_{j+1} \\ \mathcal{S}_{j+1}^1 &= \{i < m-1 \mid \ (i-1) \in S_j^0 \ \text{ and } \ P[i] = T[j+2]\} \cup \lambda_{j+2} \end{aligned} \tag{3}$$

where we assume that, for a given set S, if $i \in S$ than $S \cup \{i\} = S$.

Such relations, coupled with the initial conditions in Eq.2, allow one to compute the sets S_j^0 and S_j^1 in an iterative fashion, as shown in Fig. 2 (on the top). Observe that S_j^0 is computed in terms of both S_{j-1}^0 and S_{j-1}^1, while S_j^1 needs only S_{j-1}^0 to be computed. The resulting shape is a double crossed link, from

Computation of S_9^0	Computation of S_9^1
$4 \in S_8^0$ but $P[5] \neq T[9] \rightarrow 5 \notin S_9^0$	$4 \in S_8^0$ but $P[5] \neq T[10] \rightarrow 5 \notin S_9^1$
$5 \in S_8^0$ and $P[6] = T[9] \rightarrow \mathbf{6} \in S_9^0$	$5 \in S_8^0$ and $P[6] = T[10] \rightarrow 6 \in S_9^1$
$0 \in S_8^1$ and $P[1] = T[8] \rightarrow 1 \in S_9^0$	$P[0] = T[10] \rightarrow 0 \in S_9^1$
$2 \in S_8^1$ and $P[3] = T[8] \rightarrow 3 \in S_9^0$	
$P[0] = T[9] \rightarrow 0 \in S_9^0$	
$S_9^0 = \{0, 1, 3, \mathbf{6}\}$	$S_9^1 = \{0, 6\}$

Fig. 2. (On the top) A graphic representation of the iterative fashion for computing sets S_j^0 and S_j^1 for increasing values of j. **(On the bottom)** Computation of the sets S_9^0 and S_9^1 in terms of sets S_8^0 and S_8^1 for a pattern $P = babaaab$ of length 7 and a text $T = abbababaabbabaa$ of length 15. We notice that $S_8^0 = \{4, 5\}$ and $S_8^1 = \{0, 2\}$.

which the name of the algorithm of Fig. 3, Cross-Sampling, which solves the swap matching problem by computing sets S_j^0 and S_j^1 for increasing values of j

Fig. 2 shows also (on the bottom) the computation of the sets S_9^0 and S_9^1 in terms of sets S_8^0 and S_8^1 for the pattern $P = babaaab$ and a text $T = abbababaabbabaa$ presented in the example of Fig. 1.

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 11 are executed $|S_j^0|$ and $|S_j^1|$ times, respectively. However according with Lemma 1, 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 $0 \leq i < m$, which implies $|S_j^0| \leq m$ and $|S_j^1| < m$. Thus the time complexity of the resulting algorithm is $\mathcal{O}(nm)$.

3.1 Solving the Approximate Swap Matching Problem

Now we show how is possible to extend the Cross-Sampling algorithm to solve the approximate swap matching problem. To begin with we extend Lemma 2 with the following

Lemma 3. *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* $P_{i-1} \propto_k T_{j-1}$

(A) CROSS-SAMPLING (P, m, T, n)

```
S^0_0 ← S^1_0 ← ∅
if P[0] = T[0] then S^0_0 ← {0}
if P[0] = T[1] then S^1_0 ← {0}
for j = 1 to n − 1 do
  S^0_j ← S^1_j ← ∅
  for each i ∈ S^0_{j−1} do
    if i < m − 1 then
      if P[i + 1] = T[j]
        then S^0_j ← S^0_j ∪ {i + 1}
      if j < n − 1 and P[i + 1] = T[j + 1]
        then S^1_j ← S^1_j ∪ {i + 1}
    else Output(j − 1)
  for each i ∈ S^1_{j−1} do
    if i < m − 1 and P[i + 1] = T[j − 1]
      then S^0_j ← S^0_j ∪ {i + 1}
  if P[0] = T[j] then S^0_j ← S^0_j ∪ {0}
  if j < n − 1 and P[0] = T[j + 1]
    then S^1_j ← S^1_j ∪ {0}
for each i ∈ S^0_{j−1} do
  if i = m − 1 then Output(n − 1)
```

(B) APPROXIMATE-CROSS-SAMPLING (P, m, T, n)

```
S̄^0_0 ← S̄^1_0 ← ∅
if P[0] = T[0] then S̄^0_0 ← {(0, 0)}
if P[0] = T[1] then S̄^1_0 ← {(0, 0)}
for j = 1 to n − 1 do
  S̄^0_j ← S̄^1_j ← ∅
  for each (i, k) ∈ S̄^0_{j−1} do
    if i < m − 1 then
      if P[i + 1] = T[j]
        then S̄^0_j ← S̄^0_j ∪ {(i + 1, k)}
      if j < n − 1 and P[i + 1] = T[j + 1]
        then S̄^1_j ← S̄^1_j ∪ {(i + 1, k)}
    else Output((j − 1, k))
  for each (i, k) ∈ S̄^1_{j−1} do
    if i < m − 1 and P[i + 1] = T[j − 1]
      then S̄^0_j ← S̄^0_j ∪ {(i + 1, k + 1)}
  if P[0] = T[j] then S̄^0_j ← S̄^0_j ∪ {(0, 0)}
  if j < n − 1 and P[0] = T[j + 1]
    then S̄^1_j ← S̄^1_j ∪ {(0, 0)}
for each (i, k) ∈ S̄^0_{n−1} do
  if i = m − 1 then Output(n − 1, k)
```

Fig. 3. (A) The CROSS-SAMPLING algorithm for solving the swap matching problem. **(B)** The APPROXIMATE-CROSS-SAMPLING algorithm for solving the approximate swap matching problem

– $P[i] = T[j-1]$, $P[i-1] = T[j]$ *and either* $(i = 1 \wedge k = 1)$, *or* $P_{i-2} \propto_{k-1} T_{j-2}$ ■

Then we define sets $\bar{S}^0_j$ and $\bar{S}^1_j$ to 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 $0 \le j \le n$. Formally

$$\begin{aligned}
\bar{S}^0_j &= \{(i,k) \mid 0 \le i \le m-1 \text{ and } P_i \propto_k T_j\} \\
\bar{S}^1_j &= \{(i,k) \mid 0 \le i < m-1 \text{ and } (P_{i-1} \propto_k T_{j-1} \vee i = 0) \text{ and } P[i] = T[j+1]\}
\end{aligned}$$

Then the approximate swap matching problem translates to the problem of finding all pairs j such that $(m-1, k) \in \bar{S}^0_j$, for $0 \le k < \lfloor m/2 \rfloor$.

Sets $\bar{S}^0_0$ and $\bar{S}^1_0$ can be defined as follows

$$\bar{\lambda}_j = \begin{cases} \{(0,0)\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise} \end{cases}, \qquad \bar{S}^0_0 = \bar{\lambda}_0, \quad \text{and} \quad \bar{S}^1_0 = \bar{\lambda}_1.$$

while Lemma 3 justifies the following recursive definition of the sets $\bar{S}^0_{j+1}$ and $\bar{S}^1_{j+1}$ in terms of $\bar{S}^0_j$ and $\bar{S}^1_j$, for $j < n$:

$$\begin{aligned}
\bar{S}^0_{j+1} = \{(i,k) \mid i \le m-1 \text{ and } &((i-1,k) \in \bar{S}^0_j \text{ and } P[i] = T[j+1]) \text{ or} \\
&((i-1,k-1) \in \bar{S}^1_j \text{ and } P[i] = T[j])\ \} \cup \bar{\lambda}_{j+1} \\
\bar{S}^1_{j+1} = \{(i,k) \mid i < m-1 \text{ and } &(i-1,k) \in \bar{S}^0_j \text{ and } P[i] = T[j+2]\} \cup \bar{\lambda}_{j+2}
\end{aligned}$$

Computation of $\bar{S}_9^0$	Computation of $\bar{S}_9^1$
$(4,0) \in \bar{S}_8^0$ but $P[5] \neq T[9] \rightarrow (5,0) \notin \bar{S}_9^0$	$(4,0) \in \bar{S}_8^0$ but $P[5] \neq T[10] \rightarrow (5,0) \notin \bar{S}_9^1$
$(5,2) \in \bar{S}_8^0$ and $P[6] = T[9] \rightarrow \mathbf{(6,2)} \in \bar{S}_9^0$	$(5,2) \in \bar{S}_8^0$ and $P[6] = T[10] \rightarrow (6,2) \in \bar{S}_9^1$
$(0,0) \in \bar{S}_8^1$ and $P[1] = T[8] \rightarrow (1,1) \in \bar{S}_9^0$	$P[0] = T[10] \rightarrow (0,0) \in \bar{S}_9^1$
$(2,0) \in \bar{S}_8^1$ and $P[3] = T[8] \rightarrow (3,1) \in \bar{S}_9^0$	
$P[0] = T[9] \rightarrow (0,0) \in \bar{S}_9^0$	
$S_9^0 = \{(0,0),(1,1),(3,1),\mathbf{(6,2)}\}$	$S_9^1 = \{(0,0),(6,2)\}$

Fig. 4. Computation of the sets $\bar{S}_9^0$ and $\bar{S}_9^1$ in terms of sets $\bar{S}_8^0$ and $\bar{S}_8^1$ for a pattern $P = babaaab$ of length 7 and a text $T = abbababaabbabaa$ of length 15. We notice that $\bar{\bar{S}}_8^0 = \{(4,0),(5,2)\}$ and $\bar{\bar{S}}_8^1 = \{(0,0),(2,0)\}$.

Fig. 4 shows the computation of the sets $\bar{S}_9^0$ and $\bar{S}_9^1$ in terms of sets $\bar{S}_8^0$ and $\bar{S}_8^1$ for the pattern $P = babaaab$ and a text $T = abbababaabbabaa$ presented in the example of Fig. 1.

Fig. 3(B) shows the Approximate-Cross-Sampling algorithm for solving the approximate swap matching problem. The worst case time complexity of the algorithm is $\mathcal{O}(nm)$.

4 A Linear Algorithm for Short patterns

In this section we present simple algorithms to search swapped occurrence of a pattern in a text which makes use of bit-parallelism [BYG92]. This technique consists in taking 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 S_j^0 and S_j^1 as lists of m bits, D_j^0 and D_j^1 respectively, where m is the length of the pattern. The i-th bit of D_j^0 is set to 1 if $i \in S_j^0$, i.e. if $P_i \propto T_j$ while the i-th bit of D_j^1 is set to 1 if $i \in S_j^1$, i.e. if $P_{i-1} \propto T_{j-1}$ and $P[i] = T[j+1]$. All other 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 S_j^0 and S_j^1.

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 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 vector D_j^0 in terms of D_{j-1}^0 and D_{j-1}^1, by performing the following bitwise operations:

(A) Bit Vectors	(B) Computation of $\bar{D}^0_9$	(C) Computation of $\bar{D}^1_9$
$M[a] =$ 0111010	$(D^0_8 \ll 1)\|1 :$ 1100001 &	$(D^0_8 \ll 1)\|1 :$ 1100001 &
$M[b] =$ 1000101	$M[b] :$ 1000101 =	$M[b] :$ 1000101 =
	1000001 |	1000001
$D^0_8 =$ 0110000	$(D^1_8 \ll 1)\&M[a] :$ 0001010 =	
$D^1_8 =$ 0000101	1001011	

Fig. 5. **(A)** Bit vectors precomputed by the algorithm and **(B-C)** the computation of the sets $\bar{S}^0_9$ and $\bar{S}^1_9$ in terms of sets $\bar{S}^0_8$ and $\bar{S}^1_8$ for a pattern $P = babaaab$ of length 7 and a text $T = abbababaabbabaa$ of length 15. We notice that $\bar{S}^0_8 = \{(4,0),(5,2)\}$ and $\bar{S}^1_8 = \{(0,0),(2,0)\}$.

$$
\begin{array}{ll}
D^0_j \leftarrow D^0_{j-1} \ll 1 & S^0_j = \{i : (i-1) \in S^0_{j-1}\} \\
D^0_j \leftarrow D^0_j \mid 1 & S^0_j = S^0_j \cup \{0\} \\
D^0_j \leftarrow D^0_j \;\&\; M[T[j]] & S^0_j = S^0_j \setminus \{i : P[i] \neq T[j]\} \\
D^0_j \leftarrow D^0_j \mid H^1 & S^0_j = S^0_j \cup \{i : (i-1) \in S^1_{j-1} \wedge P[i] = T[j-1]\}
\end{array}
$$

where $H^1 = \big((D^1_{j-1} \ll 1) \;\&\; M[T[j-1]]\big)$.

Similarly the bit vector D^1_j is computed during the j-th iteration of the algorithm in terms of D^0_{j-1}, by performing the following bitwise operations:

$$
\begin{array}{ll}
D^1_j \leftarrow D^0_{j-1} \ll 1 & S^1_j = \{i : (i-1) \in S^0_{j-1}\} \\
D^1_j \leftarrow D^1_j \mid 1 & S^1_j = S^1_j \cup \{0\} \\
D^1_j \leftarrow D^1_j \;\&\; M[T[j+1]] & S^1_j = S^1_j \setminus \{i : P[i] \neq T[j+1]\}
\end{array}
$$

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

Fig. 5 shows the computation of the bit vectors D^0_9 and D^1_9 in terms of sets D^0_8 and D^1_8 for the pattern $P = babaaab$ and the text $T = abbababaabbabaa$ presented in Fig. 1.

In practice we can use only two vectors to implement sets D^0_j and D^1_j. Thus during iteration j of the algorithm vector D^0_{j-1} is transformed in vector D^0_j while vector D^1_{j-1} is transformed in vector D^1_j. The BP-Cross-Sampling algorithm is shown in Fig. 6(A). It achieves $O(\lceil mn/w \rceil)$ worst-case time and require $O(\sigma\lceil m/w \rceil)$ extra-space. If the length of the pattern is $m \leq w$ then the algorithm reports all swapped matches in $O(n)$ time and $O(\sigma)$ extra space.

4.1 Approximate Pattern Matching with Swaps

Similarly to the algorithm presented above, the simulation of the Approximate-Cross-Sampling algorithm is performed by representing the sets $\bar{S}^0_j$ and $\bar{S}^1_j$ as a list of q bits, D^0_j and D^1_j respectively, where where $q = \log(\lfloor m/2 \rfloor + 1) + 1$ and m is the length of the pattern. If the pair $(i,k) \in \bar{S}^0_j$, for $0 \leq i < m$ and

(A) BP-CROSS-SAMPLING (P, m, T, n)

```
F ← 0^{m−1}1
for each 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^0 ← D^1 ← 0^m
for j ← 0 to n − 1 do
    H ← (D^0 ≪ 1) | 1
    D^0 ← (H & M[T[j]])
    D^1 ← (D^1 ≪ 1) & M[T[j − 1]]
    D^0 ← D^0 | D^1
    D^1 ← H & M[T[j + 1]]
    if (D^0 & F) ≠ 0^m then
        Output(j)
```

(B) BP-APPROXIMATE-CROSS-SAMPLING (P, m, T, n)

```
q ← log(⌊m/2⌋ + 1) + 1
F ← 0^{qm−1}1
G ← 0^{q(m−1)}1^q
for each c ∈ Σ do
    M[c] ← 0^{qm}
    B[c] ← 0^{qm}
for i ← 0 to m − 1 do
    M[P[i]] ← M[P[i]] | F
    B[P[i]] ← B[P[i]] | G
    F ← (F ≪ q)
    G ← (G ≪ q)
F ← 0^{q−1}10^{q(m−1)}
D^0 ← D^1 ← 0^{qm}
for j ← 0 to n − 1 do
    H^0 ← (D^0 ≪ q) | 1
    H^1 ← ((D^1 ≪ q) | 1) & M[T[j − 1]]
    D^0 ← (H^0 & B[T[j]]) | H^1
    D^0 ← D^0 + (H^1 ≪ 1)
    D^1 ← (H^0 & M[T[j + 1]]) & D^0
    if (D^0 & F) ≠ 0^{qm} then
        k ← (D^0 ≫ (q(m − 1) + 1))
        Output(j, k)
```

Fig. 6. (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

$0 \leq k \leq \lfloor m/2 \rfloor$, then the rightmost bit of the i-th block of D^0_j is set to 1 and the leftmost $q-1$ bits of the i-th block contain the value k (we need exactly $\log(\lfloor m/2 \rfloor + 1)$ to represent a value between 0 and $\lfloor m/2 \rfloor$). Otherwise if the pair (i, k) does not belong to S^0_j then the rightmost bit of the i-th block of D^0_j is set to 0. In a similar way we maintain the current configuration of the set $\bar{S}^1_j$.

If $m \log(\lfloor m/2 \rfloor + 1) + m \leq w$ 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 $\bar{S}^0_j$ and $\bar{S}^1_j$.

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 1 if $P[i] = c$. Moreover the algorithm maintains, for each character $c \in \Sigma$, a bit mask $B[c]$ where the i-th block have all bits set to 1 if $P[i] = c$, while all other bits are set to 0.

Consider the example shown in Fig. 1 where the pattern $P = babaaab$ has length 7. Then each block is made up by q bits where $q = \log(\lfloor 7/2 \rfloor + 1) + 1 = 3$. The leftmost two bits of each block contain the number of swaps k, which is a value between 0 and 3. Fig. 7(A) shows the bit vectors computed by the algorithm in the preprocessing phase.

Before entering in details we observe that if $i \in S^0_j$ and $i \in S^1_j$ 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 implies a swap between two identical characters of the pattern. This last condition violates Definition 1(3). Thus during the

(A) Bit Vectors

$$
\begin{array}{ll}
M[a] & = \texttt{000 001 001 001 000 001 000} \\
M[b] & = \texttt{001 000 000 000 001 000 001} \\
B[a] & = \texttt{000 111 111 111 000 111 000} \\
B[b] & = \texttt{111 000 000 000 111 000 111} \\
 & \\
D_8^0 & = \texttt{000 101 001 000 000 000 000} \\
D_8^1 & = \texttt{000 000 000 000 001 000 001}
\end{array}
$$

(B) Computation of $\bar{D}_9^0$

$$
\begin{array}{rll}
(D_8^0 \ll q)|1: & \texttt{101 001 000 000 000 000 001} & \& \\
B[b]: & \underline{\texttt{111 000 000 000 111 000 111}} & = \\
 & \texttt{101 000 000 000 000 000 001} & | \\
(D_8^1 \ll q)\&M[a]: & \underline{\texttt{000 000 000 001 000 001 000}} & = \\
 & \texttt{101 000 000 001 000 001 001} & + \\
(D_8^1 \ll (q+1))\&M[a]: & \underline{\texttt{000 000 000 010 000 010 000}} & = \\
 & \texttt{101 000 000 011 000 011 001} &
\end{array}
$$

Fig. 7. Computation of the sets D_9^0 in terms of sets D_8^0 and D_8^1 for a pattern $P = babaaab$ of length 7 and a text $T = abbababaabbabaa$ of length 15. We notice that $\bar{S}_8^0 = \{(4,0),(5,2)\}$ and $\bar{S}_8^1 = \{(0,0),(2,0)\}$.

computation of vectors D_j^0 and D_j^1 we keep the following invariant

$$\text{if the } i\text{-th bit of } D_j^0 \text{ is set to 1} \Rightarrow \text{the } i\text{-th bit of } D_j^1 \text{ is set to 0} \tag{4}$$

The bit vectors are initialized to 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 vector D_j^0 in terms of D_{j-1}^0 and D_{j-1}^1, by performing the following bitwise operations:

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

where we have set $H^1 = (D_{j-1}^1 \ll q)\ \&\ M[T[j-1]]$ and consequently the set K is define by $K = \{(i,k) : (i-1,k) \in \bar{S}_{j-1}^1 \wedge P[i] = T[j-1]\}$.

Similarly the bit vector D_j^1 is computed during the j-th iteration of the algorithm in terms of D_{j-1}^0, by performing the following bitwise operations:

$$
\begin{array}{ll}
D_j^1 \leftarrow D_{j-1}^0 \ll q & \bar{S}_j^1 = \{(i,k) : (i-1,k) \in \bar{S}_{j-1}^0\} \\
D_j^1 \leftarrow D_j^1 \mid 1 & \bar{S}_j^1 = \bar{S}_j^1 \cup \{(0,0)\} \\
D_j^1 \leftarrow D_j^1 \ \&\ B[T[j+1]] & \bar{S}_j^1 = \bar{S}_j^1 \setminus \{(i,k) : P[i] \neq T[j+1]\} \\
D_j^1 \leftarrow D_j^1 \ \&\ \sim D_j^0 & \bar{S}_j^1 = \bar{S}_j^1 \setminus \{(i,k) : (i,k) \in \bar{S}_j^0\}
\end{array}
$$

During the j-th iteration of the algorithm, if the rightmost bit of the $(m-1)$-th block of D_j^0 is set to 1, i.e. if $(D_j^0\ \&\ 10^{q(m-1)}) \neq 0^m$, we report a swap match at position j. Moreover 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 $(m-1)$-th block of D_j^0 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 implement sets D_j^0 and D_j^1. Thus during iteration j of the algorithm vector D_{j-1}^0 is transformed in vector D_j^0 while vector D_{j-1}^1

is transformed in vector D_j^1. The BP-Approximate-Cross-Sampling algorithm, shown in Fig. 6, achieves $O(\lceil (mn\log m)/w \rceil)$ worst-case time and require $O(\sigma\lceil m\log m/w \rceil)$ 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 $O(n)$ time and $O(\sigma)$ extra space.

5 Conclusions

In this paper, we have presented a new approach for solving both Swap Matching and Approximate Swap Matching Problems. In particular we devised an efficient algorithm, named Cross-Sampling, with a $O(nm)$ worst case and a $O(n)$ average time complexity for alphabet with a uniform distribution of characters. Then we have shown how to obtain an efficient implementation of the Cross-Sampling algorithm, based on bit-parallelism, achieving $O(n)$ worst case time and $O(\sigma)$ space complexity for patterns having length similar to the word-size of the target machine. This is the first algorithm which solves the Swap Matching and the Approximate Swap Matching Problem in linear time even if for short patterns.

References

[AAL+97] Amihood Amir, Yonatan Aumann, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Pattern matching with swaps. In *IEEE Symposium on Foundations of Computer Science*, pages 144–153, 1997.

[ACH+03] Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003.

[ALLL98] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Efficient special cases of pattern matching with swaps. *Information Processing Letters*, 68(3):125–132, 1998.

[ALP02] Amihood Amir, Moshe Lewenstein, and Ely Porat. Approximate swapped matching. *Inf. Process. Lett.*, 83(1):33–39, 2002.

[BYG92] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[IR08] Costas S. Iliopoulos and M. Sohel Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In *SOFSEM*, pages 316–327, 2008.

[Mut95] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *CPM*, pages 298–317, 1995.