Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

String Pattern Matching with Swaps

Václav Blažej

Supervisor: RNDr. Tomáš Valla, Ph.D

11th May 2015

Acknowledgements

Foremost I thank my supervisor Tomáš Valla for his enthusiasm and patience. I also thank my parents for the unceasing encouragement and attention. I am also grateful to all my friends, who supported me through this venture.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 11th May 2015 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Václav Blažej. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Blažej, Václav. *String Pattern Matching with Swaps.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Vyhledávání řetězcových vzorků s použitím záměn je problém hledání všech výskytů vzorků v textu, přičemž je ve vzorku dovoleno zaměňovat sousední symboly. Cílem je navrhnout rychlý vyhledávací algoritmus, který využije bitového paralelismu bitových instrukcí koncového stroje. Nedávno jsme nalezli závažnou chybu v algoritmu od [Ahmed et al.: The swap matching problem revisited, Theor. Comp. Sci. 2014], kterou detailně popíšeme. Zároveň ukážeme proč tento algoritmus nelze jednoduše opravit. Dále vyvodíme nový algoritmus, který je založen na jiných principech a ukážeme jeho správnost. Nakonec tento algoritmus generalizujeme tak, aby dokázal vyřešit problém Wildcardového vyhledávání řetězcových vzorků s použitím záměn.

Klíčová slova Návrh a analýza algoritmů, Vyhledávání řetězcových vzorků s použitím záměn, Řetězec, Wildcardové vyhledávání vzorků

Abstract

Pattern matching with swaps problem is to find all occurrences of pattern in text while allowing pattern to swap adjacent symbols. The goal is to design fast matching algorithm that takes advantage of the bit parallelism of bitwise machine instructions. We recently found a fatal flaw in the algorithm by [Ahmed et al.: The swap matching problem revisited, Theor. Comp. Sci. 2014] which we describe in detail. Moreover we show why this algorithm cannot be fixed in any simple way. Furthermore we devise a new algorithm which is based on different principles and we prove its correctness. Finally we generalize this algorithm to solve the wildcard pattern matching with swaps problem.

Keywords Design and analysis of algorithms, Pattern Matching with Swaps, Swap Matching problem, String, Wildcard Swap Matching problem

Contents

Introduction **1**

1 Notations and basic definitions **3**
1.1 A Graph theoretic model 3
1.2 Use of Graph theoretic model for matching 5

2 Smalgo algorithms and why they cannot work **9**
2.1 Shift-And algorithm 9
2.2 Smalgo algorithms 10
2.3 Flaw in Smalgo algorithms 16

3 Our correct algorithm **19**
3.1 Graph theoretic take on Shift-And algorithm 19
3.2 Our algorithm for Swap Matching using Graph theoretic model 21

4 Wildcard Swap Matching **25**
4.1 Wildcard matching with Shift-And algorithm 29
4.2 WGSM algorithm 30

Conclusion and further research **35**

Bibliography **37**

A Acronyms **39**

B Contents of enclosed CD **41**

List of Figures

1.1 P-Graph $\mathcal{P}_P$ for the pattern $P = abcbbac$ 4
1.2 Example T-Graph $\mathcal{T}_T$ for a pattern $T = abcbbac$ 5
1.3 BMA of $T_{[2,6]} = abcab$ on a P-Graph of the pattern $P = acbab$ . . . 8

2.1 Symbols of the P-Graph for $P = acbab$ Smalgo-II could confuse . . 13
2.2 Smalgo-I flaw represented in the P-Graph for $P = abab$ 16

3.1 The P-Graph for the pattern $P = accab$ 24

4.1 The W-Graph with three $*$ tokens next to each other 31

List of Tables

2.1 D-masks initialization 10
2.2 Shift-And algorithm execution 10
2.3 Masks initialization for $\tilde{P}$ of $P = acbab$ 11
2.4 *Up*, *Middle* and *Down* masks initialization for $P = acbab$ 15
2.5 D-masks for $P = abab$ 17
2.6 P-masks for $P = abab$ 17
2.7 Smalgo-I algorithm execution for $P = abab$ and $T = aaba$ 17

3.1 Example of parallel execution of basic matching algorithm 21
3.2 GSM algorithm D-masks initialization for $P = accab$ 24
3.3 GSM algorithm execution for $P = accab$ and $T = acacba$ 24

4.1 Masks initialization for the pattern $P = a{*}?[ba] * c$ 28
4.2 WGSM D-masks initialization for $P = *c * a*$ 32
4.3 WGSM S-masks initialization for $P = *c * a*$ 33
4.4 WGSM algorithm execution for $P = *c * a*$ and $T = aca$ 33

Introduction

The *Pattern Matching problem with Swaps* (Swap Matching problem, for short) is a variation of the Pattern Matching problem. Swap Matching problem is to find all occurrences of any *swap version* of a pattern P in a text T where P and T of length p and t respectively are sequences of symbols of an alphabet Σ. By swap version of the pattern P we mean such a sequence of symbols that can be created from P by swapping non-identical adjacent symbols while ensuring that each symbol is swapped at most once. Algorithm which solves Swap Matching problem returns a set of indices which represent where swap matches of P in T begin (or alternatively end). Swap Matching problem is an intensively studied problem due to its use in practical applications such as text and music retrieval, data mining, network security and biological computing.

The Swap Matching problem was introduced in 1995 as an open problem in non-standard string matching [11]. The first result was reported by Amir et al. [2] in 1997, who provided $O(tp^{\frac{1}{3}} \log p)$ solution for alphabets of size 2, while also showing that alphabets of size exceeding 2 can be reduced to size 2 with little overhead. Amir et al. [4] also came up with solution with $O(p \log^2(p))$ time complexity for some very restrictive cases. Later Amir et al. [3] solved the Swap Matching problem in $O(t \log p \log |\Sigma|)$. Note that all above algorithms are based on the fast Fourier transform (FFT) technique.

In 2008 Iliopoulos and Rahman in [10] came up with the first efficient solution to the Swap Matching problem without using the FFT technique. They introduced a new graph theoretic approach to model the problem. They also presented an algorithm based on bit parallelism which runs in $O((n + m) \log m)$ time if the pattern length is similar to the word-size in the target machine.

In 2009, Cantone and Faro [7] presented the Cross Sampling algorithm for solving the Swap Matching problem in $O(n)$ time and $O(|\Sigma|)$ space complexity, assuming that the pattern length is similar to the word-size in the target machine.

In the same year Campanelli, Cantone and Faro [5] improved Cross Sampling algorithm using notions from Backward directed acyclic word graph matching (BDM) algorithm and named the new algorithm Backward Cross Sampling. This algorithm also assumes short pattern length. Although implementation of Backward Cross Sampling has $O(|\Sigma|)$ space and $O(nm)$ time complexity which is worse than Cross Sampling, it improves real world performance.

In 2013, Faro [9] presented a new model to solve Swap Matching problem using reactive automata. Author also presented a new algorithm with $O(n)$ time complexity assuming short patterns.

In 2014, Ahmed et al. [1] revisited Swap Matching problem using ideas from the algorithm by Iliopoulos and Rahman [10]. They devised two algorithms named Smalgo-I and Smalgo-II which both run in $O(n)$ for short pattern.

Our contribution

We noticed a fatal flaw in Smalgo-I and Smalgo-II algorithms [1] and tracked it back to the Iliopoulos and Rahman [10] which was the first attempt to solve Swap Matching problem without the FFT technique.

First we introduce all the basic definitions in Chapter 1. Then we describe the flaw in great detail in Chapter 2 where we also show input pattern and text sequences which cause the flaw to happen. Next we explain why the flaw is not repairable in any reasonable way. In Chapter 3 we show our own algorithm which uses the model described in [10] in a new way. In Chapter 4 we present a wildcard variant of the Swap Matching problem and alteration of our algorithm which solves it.

Chapter 1

Notations and basic definitions

A *string* S over an alphabet Σ is a finite sequence of symbols from Σ. By $|S|$ we denote the length of the sequence S. By S_i we mean the i-th symbol of S and we define a *substring* $S_{[i,j]} = S_i S_{i+1} \dots S_j$ for $1 \le i \le j \le |S|$, and *prefix* $\hat{S}_i = S_{[1,i]}$ for $1 \le i \le |S|$.

Definition 1. (See [5].) For a string S a *swapped version* $\pi(S)$ is a string $\pi(S) = S_{\pi(1)} S_{\pi(2)} \dots S_{\pi(n)}$ where π is a *swap permutation* $\pi : \{1 \dots n\} \to \{1 \dots n\}$ such that:

1. if $\pi(i) = j$ then $\pi(j) = i$ (characters at positions i and j are swapped),
2. for all $i, \pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters are swapped),
3. if $\pi(i) \ne i$ then $S_{\pi(n)} \ne S_i$ (identical characters are not swapped).

The string we are searching in is called *text* and the string we search for is called *pattern*.

Definition 2. Given a text $T = T_1 T_2 \dots T_t$ and a pattern $P = P_1 P_2 \dots P_p$, P is said to *swap match* T at location i if there exists a swapped version $\pi(P)$ that matches T at location i, that means $\pi(P) = T_{[i,i+p-1]}$.

1.1 A Graph theoretic model

Our algorithms are based on model introduced by Iliopoulos and Rahman [10]. In this section we briefly describe this model.

For a pattern P of length p we construct a labelled graph $\mathcal{P}'_P = (V', E', \sigma)$ with vertices V', edges E' and vertex labelling function $\sigma : V' \to \Sigma$. Let $V' = \{m_{r,c} \; ; -1 \le r \le 1, 1 \le c \le p\}$ where each vertex $m_{r,c}$ is identified with an element of a grid $3 \times p$. For each $m_{r,c}$ we set $\sigma(m_{r,c}) = P_{r+c}$. We use $[w, q]$

to denote the vertex $m_{w,q}$ when it is clear that we talk about vertices on a grid.

We set $E' := E'_1 \cup E'_2 \cup \cdots \cup E'_{p-1}$ where E'_j is defined as

$$E'_j := \Big\{ \left([k,j], [i,j+1] \right); k \in \{-1,0\}, i \in \{0,1\} \Big\} \cup \Big\{ \left([1,j], [-1,j+1] \right) \Big\}.$$

Furthermore we define the graph $\mathcal{P}_P := (V, E, \sigma)$ such that $V := V' \setminus \{Q \cup \{m_{-1,0}, m_{1,p-1}\}\}$ where

$$Q = \{m_{1,i}, m_{-1,i+1}; \sigma(m_{1,i}) = \sigma(m_{-1,i+1}), 1 \leq i \leq p-1\}.$$

We call $\mathcal{P}_P$ the *P-Graph.*

Note that $p \leq |V(\mathcal{P}_P)| \leq 3p-2$ and $(p-1) \leq |E(\mathcal{P}_P)| \leq 5(p-1)-4$ and that $\mathcal{P}_P$ is directed acyclic graph.

We construct P-Graph in a way so that $\mathcal{P}_P$ corresponds to all swap permutations (Lemma 2). Every path from the first column to a vertex represents a prefix of some swap permutation of P. Edge connecting two vertices represents that labels of those vertices are symbols which are in some swap permutation one after the other. When we find a path in P-Graph so that labels of vertices along this path create a string $\pi(P)$ which matches T on a position $k, 1 \leq k \leq (t-p)$ we found a swap match of P in T on the position k because some swap permutation of P matches this path. This implies that we can use P-Graph to find swap matches of P in T (Lemma 1).

The idea behind the way the $\mathcal{P}_P$ is constructed is as follows. We construct the graph $\mathcal{P}'_P$ and then we remove extra vertices and their incident edges to obtain $\mathcal{P}_P$. We remove vertices $m_{-1,1}$ and $m_{1,p}$ because these vertices would represent characters from invalid indices 0 and $p+1$. We remove vertices $m_{1,i}$ and $m_{-1,i+1}$ for each $1 \leq i \leq p-1$ when $P_i = P_{i+1}$ because those vertices represent swap of identical symbols which is forbidden in Swap Matching problem. For a P-Graph example see Fig. 1.1.

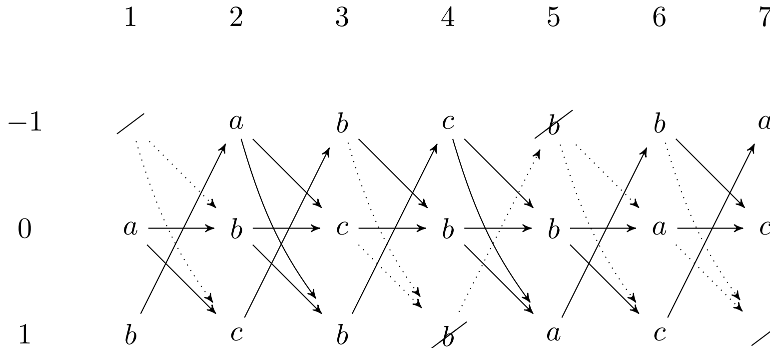

Figure 1.1: P-Graph $\mathcal{P}_P$ for the pattern $P = abcbbac$

The P-Graph now represents all possible swap permutations of the pattern P in the following sense.

Vertices $m_{0,j}$ represent pattern without any swaps. Possible swap of characters P_j and P_{j+1} is represented by vertices $m_{1,j}$ and $m_{-1,j+1}$. Edges represent symbols which can be consecutive. Each path from column 1 to column p represents unique swap pattern and each swap pattern is represented this way which is proved in Lemma 2.

1.2 Use of Graph theoretic model for matching

Definition 3. Let T be a string. The T-*Graph* of T is a graph $\mathcal{T}_T = (V, E, \sigma)$ where $V = \{v_i; 1 \leq i \leq |T|\}$, $E = \{(v_i, v_{i+1}); 1 \leq i \leq |T-1|\}$ and $\sigma : V \to \Sigma$ such that $\sigma(v_i) = T_i$.

$$a \longrightarrow b \longrightarrow c \longrightarrow b \longrightarrow b \longrightarrow a \longrightarrow c$$

Figure 1.2: Example T-Graph $\mathcal{T}_T$ for a pattern $T = abcbbac$

Definition 4. For a given directed acyclic labelled graph G with vertices $s, e \in G$, vertex labelling σ and a directed path f from s to e, *path string* of f is a string $S = \sigma(f) = \sigma(s) \ldots \sigma(e)$.

Definition 5. For a directed acyclic graph G with vertices $s, e \in G$, *maximal path* is a path f from s to e so that there exists no path from x to s where $x \in G, x \neq s$ and there exists no path from e to y where $y \in G, e \neq y$. (There exists no longer path.)

Definition 6. Directed acyclic labelled graph G_1 *matches* directed acyclic labelled graph G_2 at position j if there exists a maximal path $f \subseteq G_1$ and a path $d \subseteq G_2$ for which $\sigma(f)_{[1,p]} = \sigma(d)_{[j,j+p-1]}$.

As shown by Ahmed et al. [1] we can represent Swap Matching problem as path finding in a P-Graph. For the sake of better understanding of the techniques we also include the proof.

Lemma 1. (Ahmed [1]) Given a pattern P of length p and a text T of length t, suppose $\mathcal{P}_P$ and $\mathcal{T}_T$ are P-Graph and T-Graph of P and T, respectively. Then, P swap matches T at location $i \in [1 \ldots t]$ of T if and only if $\mathcal{P}_P$ matches G_T at position $i \in [1 \ldots t]$ of $\mathcal{T}_T$.

Proof. Suppose we have a pattern P and a corresponding P-Graph. Vertices of the P-Graph can be represented as elements of the grid m. At each column of the grid m, we have all the symbols as vertices considering the possible swaps as explained below. Each vertex in row (-1) and $(+1)$ represents a swapped

situation. Now consider column i of m corresponding to $\mathcal{P}_P$. According to definition, we have $m_{-1,i} = P_{i-1}$ and $m_{1,i-1} = P_i$. These two vertices represent the swap of P_i and P_{i-1}. Now, if this swap takes place, then in the resulting pattern, P_{i-1} must be followed by P_i. To ensure that, in $\mathcal{P}_P$, the only edge starting at $m_{1,i-1}$, goes to $m_{-1,i}$. On the other hand, from $m_{-1,i}$ we can either go to $m_{0,i+1}$ or to $m_{1,i+1}$: the former is when there is no swap for the next pair and the later is when there is another swap for the next pair. Recall that, according to the definition, the swaps are disjoint. Finally, the vertices in row 0 represents the normal (non-swapped) situation. As a result, from each $m_{0,i}$ we have an edge to $m_{0,i+1}$ and an edge to $m_{1,i+1}$: the former is when there is no swap for the next pair as well and the later is when there is a swap for the next pair. So it is easy to see that all the paths of length $p-1$ in $\mathcal{P}_P$ represent all combinations considering all possible swaps in P. Hence the result follows. □

We can obtain the same result with use of Lemma 2. Since every possible permutation is uniquely represented in the graph $\mathcal{P}_P$ with a maximal path it is sufficient to find a maximal path f such that $\sigma(f) = T_{[i,i+p]}$ to know if P swap matches T.

Definition 7. Strings P and T *prefix match* n characters on position i if $P_1P_2 \dots P_n = T_iT_{i+1} \dots T_{i+n}$.

Lemma 2. *For each swapped version of P there is a maximal path f in $\mathcal{P}_P$ with unique labelling $\sigma(f) = \pi(P)$.*

Proof. According to definition, any maximal path in $\mathcal{P}_P$ has length $p = |P|$. Since $\sigma(m_{r,c}) = P_{r+c}$ a path f in $\mathcal{P}_P$ which includes vertices $m_{0,i}$ for $1 \leq i \leq p$ represents the pattern P without any swaps. If $P_j \neq P_{j+1}$ for $1 \leq j < p$ then there exist vertices $m_{1,j}$ and $m_{-1,j+1}$ in $\mathcal{P}_P$. For each swap in P on a position i we can substitute vertices $m_{0,i}$ and $m_{0,i+1}$ for $m_{1,i}$ and $m_{-1,i+1}$ in f. The substitution will not break path f because $m_{0,i}$ and $m_{1,i}$ have same predecessors and $m_{0,i+1}$ and $m_{-1,i+1}$ have same successors. The substitution ensures that $\pi(P_i) = \sigma(m_{1,i})$ and $\pi(P_{i+1}) = \sigma(m_{-1,i+1})$. The substitution also forbids any other substitution which includes $m_{0,i}$ or $m_{0,i+1}$ since there is no path leading from $m_{1,i}$ to $m_{1,i+1}$ and from $m_{-1,i}$ to $m_{-1,i+1}$ so swaps are always disjoint as in the swapped version $\pi(P)$ of pattern P. □

Basic matching algorithm

In this section we describe an algorithm which can determine if there is a match of the pattern P of length p in the text T of length t on a position k using the graph $\mathcal{P}_P$.

To use basic matching algorithm (BMA) graph $\mathcal{P}_P = (V, E, \sigma)$ has to satisfy the following:

- $\mathcal{P}_P$ is directed acyclic graph,
- $V = V_1 \cup V_2 \cup \cdots \cup V_p$ (we can divide vertices to columns),
- $E = \{(a, b); a \in V_i, b \in V_{i+1}, 1 \leq i < p\}$ (edges lead to next column).

BMA is designed to run on every graph which satisfies these conditions. Since P-Graph and T-Graph satisfy these preliminaries we can use BMA for $\mathcal{P}_P$ and $\mathcal{T}_T$. For the sake of convenience we will define BMA in terms of $\mathcal{P}_P$ but keep in mind that this can be altered easily to be functional for any graph which satisfies the conditions.

Definition 8. Let *start vertices* of $\mathcal{P}_P$ be a set of vertices $Q = V_1$.

Definition 9. Let *accepting vertices* of $\mathcal{P}_P$ be a set of vertices $F = V_p$.

Initialize the algorithm by setting $D'_1 := Q$ (step 1). D'_1 now holds information about vertices which are end of some path f for which $\sigma(f)$ possibly prefix matches 1 symbol of $T_{[k,k+p-1]}$. To make sure that the path f represents a prefix match we need to check if a label of the last vertex of the path f matches symbol T_k (step 2). If no prefix match is left we did not find a match (step 3a). If some prefix match is left we need to check if we already have a complete match (step 3b). If the algorithm did not stop it means that we have some prefix match but it is not a complete match. Therefore we can extend this prefix match by one symbol and check if it is a valid prefix match (steps 3c and 3d). Since we extend prefix match each step we repeat these steps (3) until the prefix match is as long as the pattern.

1. Let $D'_1 := Q$.
2. Let $D_1 := \{x; x \in D'_1, \sigma(x) = T_k\}$.
3. Repeat the following steps for $i = 1, 2, 3, \ldots$
   a) If $D_i = \emptyset$ then finish.
   b) If $D_i \cap F \neq \emptyset$ then we have found a match and finish.
   c) Define the next iteration set D'_{i+1} as vertices which are descendants of D_i as $D'_{i+1} := \{d \in V(\mathcal{P}_P); vd \in E(\mathcal{P}_P)$ for some $v \in D_i\}$.
   d) Let $D_{i+1} := \{x; x \in D'_{i+1}, \sigma(x) = T_{k+i}\}$.

Having vertices in sets is not very intuitive so we present another way to describe this algorithm. The algorithm can be easily divided into *steps* (different from the steps described above). We say that algorithm is in j-th step according to index i in step 3.

Definition 10. A boolean labelling function $I : V \to \{0, 1\}$ of vertices of $\mathcal{P}_P$ is called *prefix match signal*.

We denote value of the prefix match signal in j-th step as I^j and we define the following operations:

- *propagate signal along the edges*, is a operation which sets $I^j(v) := 1$ if there exists an edge $(u, v), I^{j-1}(u) = 1$,
- *filter signal by a symbol* c, is a operation which sets $I(v) := 0$ for each v where $I(v) = 1, \sigma(v) \neq c$,
- *match check*, is a operation where we check if a match occurred or not.

With these definitions in hand we can describe BMA in terms of prefix match signals as follows:

1. Let $I^0(v) := 1$ for each $v \in Q$ and filter signals by a symbol T_k.
2. Repeat the following steps for $i = 1, 2, 3, \ldots$
   a) If $I(v) = 0$ for every $v \in \mathcal{P}_P$ then finish.
   b) If $I(v) = 1$ for any $v \in F$ then we have found a match and finish.
   c) Propagate signals along the edges and filter them by a symbol T_{k+i}.

Example 1. Suppose we want to use the BMA to figure out if $P = acbab$ swap matches $T = babcabc$ at a position $j = 2$.

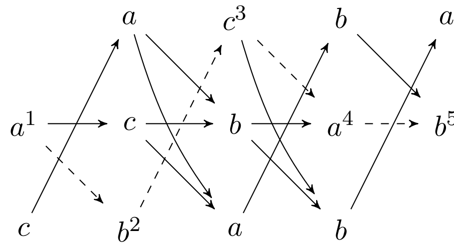

Figure 1.3: BMA of $T_{[2,6]} = abcab$ on a P-Graph of the pattern $P = acbab$

Example of matching in Fig. 1.3 shows how prefix match signal propagates along the dashed edges. Exponents j above the vertices represent for which vertices $I^j = 1$.

Chapter 2

Smalgo algorithms and why they cannot work

In this section we discuss how algorithms Smalgo-I and Smalgo-II [1] work, their flaw and we also show inputs which cause false positives.

But first we define Shift-And algorithm for standard pattern matching (without swaps) on which both algorithms are based on.

2.1 Shift-And algorithm

The following definition is based on Shift-Or algorithm [8]. Shift-And algorithm represents prefix matches by a bit value of 1 as opposed to Shift-Or algorithm which does the same thing with a value of 0. This also implies that they use different bitwise operations.

Definition 11 (*Shift-And* algorithm)**.** For a pattern P and a text T of length p and t respectively, let R be a bit array of size p. Vector R^j is the value of the array R after text symbol T_j has been processed. It contains information about all matches of prefixes of P that end at the position j in the text. For $1 \leq i \leq p$,

$$R_i^j = \begin{cases} 1 & \text{if } P_{[1,i]} = T_{[j-i+1,j]}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

The vector R^{j+1} can be computed from R^j as follows. For each i such that $R_i^j = 1$,

$$R_{i+1}^{j+1} = \begin{cases} 1 & \text{if } P_{i+1} = T_{j+1}, \\ 0 & \text{otherwise,} \end{cases} \tag{2.2}$$

and

$$R_1^{j+1} = \begin{cases} 1 & \text{if } P_1 = T_{j+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

If $R_p^{j+1} = 1$ then a complete match can be reported.

The transition from R^j to R^{j+1} can be computed very fast as follows. For each $c \in \Sigma$ let D^c be a bit array of size p such that for $1 \leq i \leq p, D_i^c = 1$ if and only if $P_i = c$.

The array D^c denotes the positions of the character c in the pattern P. Each D^c can be preprocessed before the search. And the computation of R^{j+1} reduces to three bitwise operations, *LShift*, | and &. Where LShift is left shift by one, | is Or and & is And bitwise operation.

$$R^{j+1} = \Big(\mathit{LShift}(R^j) \mid 1 \Big) \;\&\; D^{T_{j+1}} \tag{2.4}$$

Assuming that the pattern length is no longer than the memory-word size of the machine, the space and time complexity of the preprocessing phase is $O(p + |\Sigma|)$. The time complexity of the searching phase is $O(t)$, thus independent from the alphabet size and the pattern length.

Example 2. Run of the Shift-And algorithm for a pattern $P = acbab$ and a text $T = acbacbabcbabaca$

Table 2.1: D-masks initialization

P_i	D^a	D^b	D^c
a	1	0	0
c	0	0	1
b	0	1	0
a	1	0	0
b	0	1	0

Table 2.2: Shift-And algorithm execution

	a	c	b	a	c	b	a	b	c	b	a	b	a	c	a
a	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1
c	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0
b	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
a	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Since $R_5^8 = 1$ algorithm reports a match on a position $j - p + 1$ which is the position 4 for this case.

2.2 Smalgo algorithms

Both algorithms are inspired by the Shift-And algorithm which we explained in section 2.1. They also use few terms which are mentioned next.

Definition 12. A symbol w is called *degenerate* when it is a set of symbols from a finite alphabet such that $w \subseteq \Sigma$ and $w \neq \emptyset$.

Definition 13. A string S is said to be *degenerate*, if it is built over an alphabet of degenerate symbols.

Definition 14. Given a strings X and a degenerate string Y each of length ℓ, we say X matches Y, if and only if $X_i \in Y_i$ for $1 \leq i \leq \ell$.

2.2.1 Smalgo-I

Smalgo-I algorithm [1] is a modification of the Shift-And algorithm for Swap Matching problem. The algorithm uses the Graph theoretic model introduced in Section 1.1.

First we create a degenerate version $\tilde{P}$ of a pattern P of length p. In the following equation we denote $\sigma(m_{x,y})$ by $m_{x,y}$ for the sake of simplicity.

$$\tilde{P} = \{m_{0,1}, m_{1,1}\} \dots \{m_{-1,x}, m_{0,x}, m_{1,x}\} \dots \{m_{-1,p}, m_{0,p}\} \tag{2.5}$$

We say that a degenerate string $\tilde{P}$ matches a text T at a position j if $T_{j+i-1} \in \tilde{P}_i$ for every $1 \leq i \leq p$.

Each symbol in $\tilde{P}$ on a position i represents set of symbols of P which can swap match to that position. To accommodate Shift-And algorithm to match degenerate patterns we need to change the way the D^c masks are defined. For each $c \in \Sigma$ let D^c be a bit array of size p such that for $1 \leq i \leq p, D^c_i = 1$ if and only if $c \in \tilde{P}_i$.

Despite the definition we do not need to compute degenerate symbols of $\tilde{P}$. We already know that a symbol at a position i represents symbols which can swap match to that position. So instead of computing $\tilde{P}_i$ we can compute which positions is a symbol able to swap to. Algorithm for computing D^c can be described by the following: For each $1 \leq i \leq p$ set $D^{P_i}_{i+k} := 1$ for $k \in \{-1, 0, 1\}$.

Table 2.3: Masks initialization for $\tilde{P}$ of $P = acbab$

P_i	$\tilde{P}_i$	D^a	D^b	D^c
a	$[ac]$	1	0	1
c	$[acb]$	1	1	1
b	$[cba]$	1	1	1
a	$[ba]$	1	1	0
b	$[ab]$	1	1	0

Suppose we have a prefix swap match of k symbols of P and T at a position j. This implies that we also have a prefix match of k symbols of $\tilde{P}$ and T at the same position. To check whether or not we have a prefix match of $k+1$ symbols we have to check if $T_{j+k+1} \in \tilde{P}_{k+1}$. But this check is clearly not sufficient for checking the prefix swap match of P and T.

For example, $P = abcde$ swap matches $T = bacce$ 3 symbols at position 1. However P does not match T 4 symbols even though $\tilde{P}$ matches T, because c can not be on positions 3 and 4 simultaneously.

We can fix this issue by introducing *P-mask* $P_{(x_1,x_2,x_3)}$ which is defined as follows:

$$P_{(x_1,x_2,x_3)i} = \begin{cases} 1 & i = 1 \\ 1 & \text{if there exist vertices } u_1, u_2 \text{ and } u_3 \text{ and edges } (u_1,u_2),(u_2,u_3) \text{ for which } u_2 = m_{r,i} \text{ where } -1 \leq r \leq 1 \text{ and } \sigma(u_n) = x_n \text{ for } 1 \leq n \leq 3, \\ 0 & \text{otherwise.} \end{cases} \quad (2.6)$$

Now, whenever we want to check if P prefix swap matches T $k+1$ symbols at position j we check for a match of $\tilde{P}$ in T and we also check if $P_{(T_{j+k-1},T_{j+k},T_{j+k+1})k+1} = 1$. This ensures that symbols are able to swap to respective positions and that those three symbols of the text T are present in some $\pi(P)$.

We construct P-masks by traversing the P-Graph. First we initialize each P-mask to 10^{p-1}. Next we traverse the P-Graph one column at a time. For each column $2 \leq c \leq p-1$ we consider every path f of length 3 in the P-Graph that starts at the column $c-1$. Such a path consists of three vertices $m_{r_1,c-1}, m_{r_2,c}, m_{r_3,c+1}$ where r_1, r_2, r_3 are row numbers and $c-1, c, c+1$ are column numbers of the vertices. For each such path we set

$$P_{\left(\sigma(m_{r_1,c-1}),\sigma(m_{r_2,c}),\sigma(m_{r_3,c+1})\right)c} := 1. \quad (2.7)$$

Notice that due to the structure of the P-Graph the maximum number of such paths starting at one column is 8.

With P-masks completed we initialize $R^1 = 1 \ \& \ D^{T_1}$ and compute R^{j+1} as follows:

$$R^{j+1} = LSO(R^j) \ \& \ D^{T_{j+1}} \ \& \ RShift(D^{T_{j+2}}) \ \& \ P_{(T_j,T_{j+1},T_{j+2})} \quad (2.8)$$

LSO is defined as $LSO(x) = LShift(x) \mid 1$. *RShift* and *LShift* indicate right shift and left shift respectively, & is And and | is Or bitwise operation.

To check whether or not a swap match occurred we check if $R^j_{p-1} = 1$. This is sufficient because during the processing we are in fact considering not only the next symbol T_{j+1} but also the symbol after that T_{j+2}.

The space and time complexity of the preprocessing is $O(p/w(p+|\Sigma|^3))$. The time complexity of the searching phase is $O(t(p/w))$. Note that complexity changes significantly if we assume that the pattern length p is similar to the word-size w of the target machine.

2.2.2 Smalgo-II

Smalgo-II algorithm is a derivative of Smalgo-I algorithm. It improves the space complexity from $O(p/w(p + |\Sigma|^3))$ to $O(p/w(p + |\Sigma|^2))$ for a cost of more complex algorithm.

In analysis of space complexity of Smalgo-I we can see that $O(p/w(|\Sigma|^3))$ space is taken by P-masks. This can be improved by making P-masks hold information about paths of length 2 instead of 3. The change makes P-masks take only $O(p/w(|\Sigma|^2))$.

In this section we refer to Smalgo-II P-masks which are different form P-masks in Smalgo-I. We define *P-mask* $P_{(x_1,x_2)}$ as follows:

$$P_{(x_1,x_2)i} = \begin{cases} 1 & i = 1 \\ 1 & \text{if there exist vertices } u_1 \text{ and } u_2 \text{ and an edge} \\ & (u_1, u_2) \text{ for which } u_2 = m_{r,i} \text{ where } -1 \leq r \leq 1 \\ & \text{and } \sigma(u_n) = x_n \text{ for } 1 \leq n \leq 2, \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

To create P-masks we need to traverse the P-Graph similarly as in Smalgo-I. First we set all P-masks to 10^{p-1}. For each column $2 \leq c \leq p$ we consider every path of length 2 which starts at the column $c - 1$ and we set

$$P_{(\sigma(m_{r_1,c-1}),\sigma(m_{r_2,c}))c} := 1. \tag{2.10}$$

From the structure of the P-Graph we can see that there are at maximum 5 such paths for each considered column.

This information is sufficient to know if a swap match occurred. But it can also match a part of the text where a swap match did not occur. This behaviour can be clearly seen in the following example.

Example 3. Suppose we want to find matches of a pattern $P = acbab$ in a text $T = acbbb$. We build a P-Graph, P-masks, D^c for every $c \in \Sigma$ and we start the algorithm. This algorithm can break whenever there are the same symbols at positions i and $i + 2$ for any $1 \leq i \leq p - 2$ as seen in Fig. 2.1.

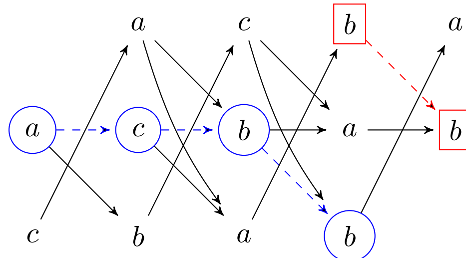

Figure 2.1: Symbols of the P-Graph for $P = acbab$ Smalgo-II could confuse

To fix this issue we need this algorithm to behave like the Smalgo-I algorithm. To tackle this we need the following definitions:

Definition 15. A *level change* indicates a change of row index in the grid of the P-Graph, having one of the following cases:

- *Upward change*: is from a position $m_{1,j}$ to $m_{-1,j+1}$,
- *Downward change*: is from a position $m_{i,j}$ to $m_{1,j+1}$ where $i \in \{-1, 0\}$,
- *Middle change*: is from a position $m_{-1,j}$ to $m_{0,j+1}$.

We can see that when a level change at a particular position occurs another level change must occur at the next position. We see that the Downward change starts a swap of two symbols, the Upward change represents the second swapped symbol and the Middle change represents end of swap when there is no swap immediately after. Therefore we know that:

- every Downward change is followed by Upward change,
- every Upward change is followed by Downward or Middle change.

To force these rules we need to define masks which represent which level changes can happen on particular positions. We do the following based on the structure of the P-Graph.

For $a, c \in \Sigma$ and $\mathcal{P}_P = (V, E, \sigma)$ we define the following masks:

- *Up mask*: $Up_j^{a,b} = 1$ if and only if an edge $(m_{1,j-1}, m_{-1,j}) \in E$, $\sigma(m_{1,j-1}) = a$ and $\sigma(m_{-1,j}) = b$ exists,
- *Down mask*: $Down_j^{a,b} = 1$ if and only if there exists at least one edge $(m_{i,j-1}, m_{1,j}) \in E$, $\sigma(m_{i,j-1}) = a$ and $\sigma(m_{1,j}) = b$ for $i \in \{-1, 0\}$,
- *Middle mask*: $Middle_j^{a,b} = 1$ if and only if there exists at least one edge $(m_{i,j-1}, m_{0,j}) \in E$, $\sigma(m_{i,j-1}) = a$ and $\sigma(m_{0,j}) = b$ for $i \in \{-1, 0\}$.

We also need to somehow express the first symbol. We do so by setting $Down_1^{x,\sigma(m_{1,1})} := 1$ for each $x \in \Sigma$. We could also set $Middle_1^{x,\sigma(m_{0,1})} := 1$ but this is not necessary, since no rule applies on the Middle change.

Example 4. We create *Up*, *Middle* and *Down* mask for a pattern $P = acbab$ shown in Table 2.4.

Table 2.4: *Up*, *Middle* and *Down* masks initialization for $P = acbab$

	Up	*Middle*	*Down*
(a, a)	00000	00000	00100
(a, b)	00010	00101	01000
(a, c)	00000	01000	10000
(b, a)	00001	00010	00000
(b, b)	00000	00001	00010
(b, c)	00100	00000	10000
(c, a)	01000	00010	00100
(c, b)	00000	00100	00010
(c, c)	00000	00000	10000

With masks prepared we alter the algorithm slightly and we get the same results as in Smalgo-I. We change the algorithm according to the mentioned rules as follows.

If a Downward change occurred then we have to check whether an Upward change occurs at the next position. We can do that by saving the previous *Down* mask and matching that value with the current *Up* mask and R^j.

If an Upward change has occurred then we have to check whether Downward change or a Middle change occurs at the next position. We can do that by saving the previous *Up* mask and matching that value with current *Down* mask, *Middle* mask and R^j.

With everything ready we initialize the algorithm by setting all the masks and R to their initial values.

$$PrevUp = 0 \tag{2.11}$$

$$PrevDown = Down^{x,m_{1,1}} \tag{2.12}$$

$$R^1 = 1 \ \& \ D^{T_1} \tag{2.13}$$

Where $x \in \Sigma$. We compute R^{j+1} as:

$$R'^{j+1} = LSO(R^j) \ \& \ P_{(T_j,T_{j+1})} \ \& \ D^{T_{j+1}}, \tag{2.14}$$

then we have to apply the rules like this:

$$R''^{j+1} := R'^{j+1} \ \& \ LShift(Down^{T_{j-1},T_j}) \ \& \ Up^{T_j,T_{j+1}} \tag{2.15}$$

$$R^{j+1} := R''^{j+1} \ \& \ LShift(Up^{T_{j-1},T_j}) \ \& \ (Middle^{T_j,T_{j+1}} \mid Down^{T_j,T_{j+1}}) \tag{2.16}$$

To find out whether the match occurred we have to check if $R^j_p = 1$.

2.3 Flaw in Smalgo algorithms

We shall see that for a pattern $P = abab$ and a text $T = aaba$ both algorithms Smalgo-I and Smalgo-II give false positives.

2.3.1 Flaw description

Concept of Smalgo-I and Smalgo-II algorithms is based on the assumption that we can find a maximal path by searching for consecutive paths of length 3 (*triplets*). But this means that the triplets have to be connected. By connected we mean that two triplets $x = (x_1, x_2, x_3)$ and $y = (y_1, y_2, y_3)$ which start in columns c and $c + 1$ respectively have two vertices in common: $x_2 = y_1$ and $x_3 = y_2$. If the assumption is not true then it is possible to check for each triplet successfully but the found substring of the text does not match any swap version of P.

We have found such a configuration and therefore the assumption is false. In Table 2.5, 2.6 and 2.7 we can see the step by step execution of Smalgo-I algorithm on a pattern $P = abab$ and a text $T = aaba$.

In Table 2.7 we see that R^3 has 1 in the 3-rd row which means that the algorithm found a pattern match on a position 1. This match is a false positive, because it is not possible to match the pattern with two b symbols in the text with only one b symbol.

The reason behind the false positive match is as follows. Algorithm checks if the first triplet of symbols (a, a, b) matches. It can match the swap pattern *aabb*. Next it checks the second triplet of symbols (a, b, a), which can match *baba*. We know that *baba* is not possible since it did not appear in the previous check, but the algorithm can not distinguish them since it checks only the triplets and nothing more. Since each step gave us positive match we get a swap match of the pattern in the text.

In the Fig. 2.2 we see that two triplets which Smalgo-I assumes have two vertices in common.

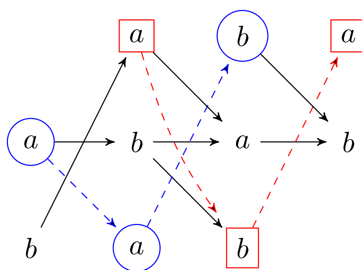

Figure 2.2: Smalgo-I flaw represented in the P-Graph for $P = abab$

Table 2.5: D-masks for $P = abab$

		D_a	D_b	D_x
1	$[ab]$	1	1	0
4	$[ba]$	1	1	0
5	$[ab]$	1	1	0
5	$[ba]$	1	1	0

Table 2.6: P-masks for $P = abab$

	$P_{(a,a,a)}$	$P_{(a,a,b)}$	$P_{(a,b,a)}$	$P_{(b,a,a)}$	$P_{(a,b,b)}$	$P_{(b,a,b)}$	$P_{(b,b,a)}$	$P_{(b,b,b)}$	$P_{(x,x,x)}$
1	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	1	0	0	0
3	0	1	1	0	1	1	1	0	0
5	0	0	0	0	0	0	0	0	0

Table 2.7: Smalgo-I algorithm execution for $P = abab$ and $T = aaba$

	-	R	D_a	$P_{(x,x,x)}$	R^1	R	D_a	LD_b	$P_{(a,a,b)}$	R^2	R	D_b	LD_a	$P_{(a,b,a)}$	R^3
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1
3	0	0	1	0	0	0	1	1	1	0	1	1	1	1	1
4	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0

Even though Smalgo-II checks level changes it does so just to simulate how Smalgo-I works. Therefore this flaw is in Smalgo-II as well.

2.3.2 Why the flaw is not easily repairable

Because of the flaws in algorithms Smalgo-I and Smalgo-II they can not solve the Swap Matching problem.

To fix the flaw in the algorithms we would have to check every swap match for correctness. This solves the issue but we have to check the swap match by another algorithm. This can be done in linear time, but when there are a lot of matches in the text T it slows down the algorithm significantly. This approach has complexity $O(tp)$ since we can find up to $O(t)$ occurrences of P in T and check of correctness takes $O(p)$.

Example 5. We have a pattern $P = abab$ and a text $T = aabaabaabaa$ of length t. This input has swap matches on positions $\{2, 5\}$. Both Smalgo algorithms report swap matches on positions $\{0, 2, 3, 5, 6\}$. Now we have to check if each of these positions represent a valid match.

We can create a text defined by a regular expression $aa\{baa\}+$ of length $t \geq 5$ which creates this worst case. Therefore time complexity of a corrected version of the Smalgo algorithm is $O(tp)$ even for pattern length similar to word-size of the target machine.

Both algorithms might be useful as an oraculum for Swap Matching problem.

Chapter **3**

Our correct algorithm

In this chapter we will show an algorithm which solves the Swap Matching problem and its correctness. We call the algorithm GSM (Graph Swap Matching). GSM uses the Graph theoretic model shown in section 1.1 and is based on Shift-And algorithm from Section 2.1.

To make concept of GSM more familiar first we present how to define Shift-And algorithm in means of the T-Graph model and basic matching algorithm (BMA) from Section 1.2 to solve Pattern Matching problem. Then we expand this idea to Swap Matching problem by using Graph theoretic model.

3.1 Graph theoretic take on Shift-And algorithm

Let T and P be a text and a pattern of length t and p respectively. We create a T-Graph $\mathcal{T}_P = (V, E, \sigma)$ of the pattern P.

We know that the T-Graph is directed acyclic graph which can be divided into columns $V_i, 1 \leq i \leq p$ where each contains one vertex v_i and that edges lead from V_j to V_{j+1}. This means that the T-Graph satisfies all preliminaries of BMA. We set start vertices $Q := \{v_1\}$ and accepting vertices $F := \{v_p\}$.

We apply BMA to $\mathcal{T}_P$ to figure out if P matches T at a position j. It is clear that we get a correct result because we check if $T(j+i-1) = \sigma(v_i) = P(i)$ for each $1 \leq i \leq p$.

To find every occurrence of P in T we would have to run BMA for each position individually. This is basically the naive approach to solve pattern matching. We can improve the algorithm significantly when we parallelize computation of p runs of BMA in the following way.

Algorithm processes one symbol at a time starting from T_1. We say that the algorithm is in the j-th step when a symbol T_j has been processed. BMA represents a prefix match as a prefix match signal $I : V \to \{0, 1\}$. We denote the value of the prefix match signal I in the j-th step by I^j. Since one run of the BMA uses only one column and therefore one vertex of the T-Graph at any time we can use other vertices to represent different runs of the BMA. To

tackle this we define $I^j(v_i)$ in the j-th step as follows

$$I^j(v_i) = \begin{cases} 1 & P_{[1,i]} = T_{[j-i+1,j]}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

We want to represent all prefix match indicators in one vector so we can manipulate them easily. To do that we prepare a bit vector R. We denote value of the vector R in j-th step as R^j. We define values of R^j as $R^j_i = I(v_i)$.

First operation which is used in BMA is called propagate signal along the edges and can be done easily by setting the signal of v_i to value of the signal of its predecessor v_{i-1}. That means for $2 \leq i \leq p$ do the following:

$$I(v_i) := I(v_{i-1}), \tag{3.2}$$
$$I(v_1) := 1. \tag{3.3}$$

The very same operation can be done easily using *LSO* bitwise operation over R which is defined as follows:

$$LSO(R) = LShift(R) \mid 1, \tag{3.4}$$

where *LShift* is left shift by one and | is standard Or bitwise operation.

We also need a way to set $I(v_i) := 0$ for each $v_i; I(v_i) = 1, \sigma(v_i) \neq T_{j+i}$ which is another basic BMA operation called filter signal by a symbol. We can do this by constructing a bit vector D^c for each $c \in \Sigma$ as follows:

$$D^c_i = \begin{cases} 1 & c = P_i, \\ 0 & \text{otherwise,} \end{cases} \tag{3.5}$$

and use this vector D^c to filter signal by a symbol c like this:

$$R \ \& \ D^c, \tag{3.6}$$

where & is And bitwise operation.

Last BMA operation we have to define is a match detection. We do this by checking $I(v_p) = 1$ and if so we know that a match starting at a position $j - p + 1$ occurred.

We can easily compute R^{j+1} from R^j as follows:

$$R^{j+1} = \left(LShift(R^j) \mid 1\right) \ \& \ D^{T_{j+1}}. \tag{3.7}$$

Step j of the final algorithm consists of computing value of vector R^j and checking if $R^j_p = 1$ and if so the algorithm reports a match.

Table 3.1: Example of parallel execution of basic matching algorithm

	a	c	b	a	c	b	a	b	c	b	a	b	a	c	a
a	1			4			7				11		13		15
c		1			4									13	
b			1			4									
a				1			4								
b								4							

Example 6. Example run of the algorithm on the pattern $P = acbab$ and the text $T = acbacbabcbabaca$ is shown in Table 3.1 where numbers denote in which step particular BMA started. Lines represent the pattern and at the same time which step are different runs of BMA in. We can see that in steps 4 and 7 two different BMA runs are computed in the same step. We can also see that the maximum number of BMA runs that can be computed in the same step is p.

3.2 Our algorithm for Swap Matching using Graph theoretic model

We use the idea from Section 3.1 and devise the GSM algorithm. Since all the notions are basically the same we use very similar approach.

Let T and P be a text and a pattern of length t and p respectively. We create a P-Graph $\mathcal{T}_P = (V, E, \sigma)$ of the pattern P.

From the definition we know that basic matching algorithm (BMA) can be used for the P-Graph. We know that the P-Graph is directed acyclic graph which can be divided into columns $V_i, 1 \leq i \leq p$ where each contains up to three vertices $m_{-1,i}, m_{0,i}, m_{1,i}$ denoted by their positions on a grid m. We also know that all edges lead from V_j to V_{j+1}. We set start vertices $Q := V_1$ and accepting vertices $F := V_p$.

We apply BMA to $\mathcal{P}_P$ to figure out if P matches T at a position j. We get the correct result because when we get a match we know that BMA had to traverse the graph and that vertices which BMA traversed form a maximal path f. Each vertex of this path had to be labelled with the symbol identical to respective symbol in the text otherwise the signal would be filtered out. Therefore we know that a swap version $\pi(P)$ of P matches the text at the position j because $\pi(P) = \sigma(f) = T_{[j,j+p-1]}$. This implies that P swap matches T at the position j.

To find every occurrence of P in T we would have to run BMA for each position individually. This is basically the naive approach to solve Swap Matching problem. We can improve the algorithm significantly when we parallelize computation of p runs of BMA in the following way.

Algorithm processes one symbol at a time starting from T_1. We say that the algorithm is in the j-th step when a symbol T_j has been processed. BMA represents a prefix match as a prefix match signal $I : V \to \{0, 1\}$. We denote the value of the prefix match signal I in the j-th step by I^j. Since one run of the BMA uses only one column of the P-Graph at any time we can use other columns to represent different runs of the BMA. To tackle this we define $I^j(v_i)$ in the j-th step as follows

$$I^j(v_i) = \begin{cases} 1 & \pi(P)_{[1,i]} = T_{[j-i+1,j]}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

We want to represent all prefix match indicators in vectors so we can manipulate them easily. We can do this by mapping rows $r \in \{-1, 0, 1\}$ of the P-Graph to vectors RU, RM and RD respectively. We denote value of the vector RX in j-th step as RX^j. We define values of the vectors as follows

$$RU^j_i = I(m_{-1,i}), \tag{3.9}$$
$$RM^j_i = I(m_{0,i}), \tag{3.10}$$
$$RD^j_i = I(m_{1,i}), \tag{3.11}$$

where the value of $I(v)$ where v does not exists is 0.

We define BMA propagate signal along the edges operation as setting the signal of $m_{r,c}$ to 1 if at least one of its predecessors have signal set to 1.

$$I(m_{-1,i}) := I(m_{1,i-1}), \tag{3.12}$$
$$I(m_{0,i}) := I(m_{-1,i-1}) \mid I(m_{0,i-1}), \tag{3.13}$$
$$I(m_{1,i}) := I(m_{-1,i-1}) \mid I(m_{0,i-1}), \tag{3.14}$$
$$I(m_{0,1}) := 1, \tag{3.15}$$
$$I(m_{1,1}) := 1, \tag{3.16}$$

where | is standard Or bitwise operation. Using *LSO* bitwise operation:

$$LSO(R) = LShift(R) \mid 1, \tag{3.17}$$

where *LShift* is left shift by one, we can define propagate signal along the edges operation like this:

$$RU := LSO(RD), \tag{3.18}$$
$$RM := LSO(RM \mid RU), \tag{3.19}$$
$$RD := LSO(RM \mid RU). \tag{3.20}$$

BMA filter signal by a symbol operation can be done by constructing a bit vector D^c for each $c \in \Sigma$ as follows:

$$DU_i^c = \begin{cases} 1 & c = P_{i-1}, P_{i-1} \neq P_i, \\ 0 & \text{otherwise}, \end{cases} \tag{3.21}$$

$$DM_i^c = \begin{cases} 1 & c = P_i, \\ 0 & \text{otherwise}, \end{cases} \tag{3.22}$$

$$DD_i^c = \begin{cases} 1 & c = P_{i+1}, P_i \neq P_{i+1} \\ 0 & \text{otherwise}, \end{cases} \tag{3.23}$$

and use this vector D^c to filter signal by a symbol c like this:

$$RU \ \& \ DU^c, \tag{3.24}$$

$$RM \ \& \ DM^c, \tag{3.25}$$

$$RD \ \& \ DD^c, \tag{3.26}$$

where & is And bitwise operation.

Last BMA operation we define is a match detection. We do this by checking if $I(m_{0,p}) = 1$ or $I(m_{-1,p}) = 1$ and if so we know that a match starting at a position $j - p + 1$ occurred.

Just by combining mentioned operations we can compute values of RX^{j+1} from RX^j as follows:

$$RU^{j+1} := LSO(RD^j) \ \& \ DU^c, \tag{3.27}$$

$$RM^{j+1} := LSO(RM^j \mid RU^j) \ \& \ DM^c, \tag{3.28}$$

$$RD^{j+1} := LSO(RM^j \mid RU^j) \ \& \ DD^c. \tag{3.29}$$

To simplify our notation we use D^c to denote DD^c, DM^c, DU^c and RX^j to denote RD^j, RM^j, RU^j. By using a specific operation on D^c or RX^j we mean use of that operation on respective vectors as defined above.

The final GSM algorithm first prepares D-masks D^c for every $c \in \Sigma$ and initiates $RX^0 := 0$. Then the algorithm computes the value of vectors RX^j and checks if either $I(m_{0,p}) = 1$ or $I(m_{-1,p}) = 1$ and if so the algorithm reports a match. This part is repeated until $j = t$.

Example 7. Let there be a pattern $P = accab$ and a text $T = acacba$ of length p and t respectively. We use the GSM algorithm to find out on which positions P swap matches T.

The algorithm first creates DX^c for every $c \in \Sigma$. Then the main part of

Table 3.2: GSM algorithm D-masks initialization for $P = accab$

i	P_{i-1}	DU^a	DU^b	DU^c	P_i	DM^a	DM^b	DM^c	P_{i+1}	DD^a	DD^b	DD^c
1		0	0	0	a	1	0	0	c	0	0	1
2	a	1	0	0	c	0	0	1	c	0	0	0
3	c	0	0	0	c	0	0	1	a	1	0	0
4	c	0	0	1	a	1	0	0	b	0	1	0
5	a	1	0	0	b	0	1	0		0	0	0

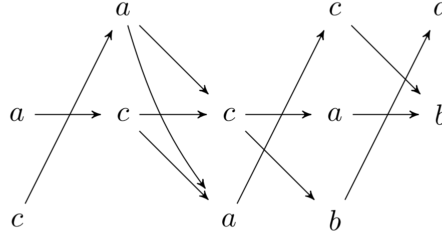

Figure 3.1: The P-Graph for the pattern $P = accab$

the algorithm runs. The LSO means use of the operation as defined in 3.20. The run in Table 3.3

Table 3.3: GSM algorithm execution for $P = accab$ and $T = acacba$

i	RX^0	LSO	D^a	RX^1	LSO	D^c	RX^2	LSO	D^a	RX^3
1	0 0 0	1 1 1	0 1 0	0 1 0	1 1 1	1 0 0	1 0 0	1 1 1	0 1 0	0 1 0
2	0 0 0	0 0 0	0 0 1	0 0 0	1 1 0	0 1 0	0 1 0	0 0 1	0 0 1	0 0 1
3	0 0 0	0 0 0	1 0 0	0 0 0	0 0 0	0 1 0	0 0 0	1 1 0	1 0 0	1 0 0
4	0 0 0	0 0 0	0 1 0	0 0 0	0 0 0	0 0 1	0 0 0	0 0 0	0 1 0	0 0 0
5	0 0 0	0 0 0	0 0 1	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 1	0 0 0
i	RX^3	LSO	D^c	RX^4	LSO	D^b	RX^5	LSO	D^a	RX^6
1	0 1 0	1 1 1	1 0 0	1 0 0	1 1 1	0 0 0	0 0 0	1 1 1	0 1 0	0 1 0
2	0 0 1	1 1 0	0 1 0	0 1 0	0 0 1	0 0 0	0 0 0	0 0 0	0 0 1	0 0 0
3	1 0 0	1 1 0	0 1 0	0 1 0	1 1 0	0 0 0	0 0 0	0 0 0	1 0 0	0 0 0
4	0 0 0	0 0 1	0 0 1	0 0 1	1 1 0	1 0 0	1 0 0	0 0 0	0 1 0	0 0 0
5	0 0 0	0 0 0	0 0 0	0 0 0	1 1 0	0 1 0	0 1 0	0 0 1	0 0 1	0 0 1

Because $RM_p^5 = 1$ and $RU_p^6 = 1$ the GSM algorithm reports matches starting on positions 1 and 2.

Chapter **4**

Wildcard Swap Matching

We can very easily change GSM to match patterns P containing basic wildcard symbols.

For a pattern P we define wildcard language $L(P)$ over an alphabet Σ as follows.

- $L(\epsilon) = \{\epsilon\}$
- For any $v \in \Sigma$ $L(Pv) = \{w \in \Sigma^*; w = uv, u \in L(P)\}$
- $L(P?) = \{w \in \Sigma^*; w = uv, u \in L(P), v \in \Sigma\}$
- $L(P*) = \{w \in \Sigma^*; w = uv, u \in L(P), v \in \Sigma^*\}$
- $L(P[v_1 v_2 \dots v_n]) = \{w \in \Sigma^*; w = uv, u \in L(P), v \in \{v_1, v_2, \dots, v_n\}\}$
- $L(P[\,!\,v_1 v_2 \dots v_n]) = \{w \in \Sigma^*; w = uv, u \in L(P), v \in \Sigma \backslash \{v_1, v_2, \dots, v_n\}\}$

This means that the pattern P can now contain following tokens:

- ? is any symbol,
- $*$ is any number of any symbols,
- $[\dots]$ is any symbol among symbols listed between brackets,
- $[\ !\ \dots]$ is any symbol of Σ which is not among symbols listed between brackets.

The pattern P is forbidden to contain two $*$ tokens next to each other because the same function can be expressed with one $*$ token.

Definition 16. *Wildcard Swap Matching* is a problem of finding matches of S in T where $S \in L(P')$ is a string from a wildcard language $L(P') = L(\pi_1(P)) \cup \dots \cup L(\pi_n(P))$.

Notice that the Wildcard Swap Matching is not a problem of finding swap matches of S in T where S is a string from a wildcard language $L(P)$. These two variants differ in that if the swaps are made before of after all the tokens are substituted with the symbols.

At this moment we should address one problem which arises with use of wildcard symbols. Swap Matching problem is defined as a pattern matching in which we can use swapped version of the pattern $\pi(P)$ instead of the pattern P. Definition also says that identical symbols can not be swapped to create the swapped version of P. Since now we cannot be sure which symbol will be used for the final match we can have difficulty forbidding swaps of identical characters. To tackle this we define GSM for wildcard matching as follows.

For a pattern P of length p we construct a labelled graph $\mathcal{W}'_P = (V', E', \sigma)$ with vertices V', edges E' and vertex labelling function $\sigma : V' \to \Sigma$. Let $V' = \{m_{r,c}\ ; -1 \le r \le 1, 1 \le c \le p\}$ where each vertex $m_{r,c}$ is identified with an element of a grid $3 \times p$ and can be referred to as $[r, c]$. For each $m_{r,c}$ we set $\sigma(m_{r,c}) = P_{r+c}$.

We set $E' := E'_1 \cup E'_2 \cup \cdots \cup E'_{p-1}$ where E'_j is defined as

$$E'_j := \Big\{ \big([k,j], [i,j+1]\big) ; k \in \{-1,0\}, i \in \{0,1\} \Big\} \cup \Big\{ \big([1,j], [-1,j+1]\big) \Big\} .$$

We define the graph $\mathcal{W}_P := (V, E, \sigma)$ such that $V := V' \setminus \{m_{-1,0}, m_{1,p-1}\}$.

Note that $\mathcal{W}_P$ is $\mathcal{P}_P$ where vertices which were removed when adjacent symbols were equal are left in the graph.

We call $\mathcal{W}_P$ the *W-Graph*.

Now we define an algorithm which is the same as GSM except one thing - it uses W-Graph instead of P-Graph. We call this algorithm GSM'.

Now we prove that those algorithms give the same results and therefore we do not care if we use P-Graph or W-Graph.

Lemma 3. For a pattern P and a text T, a run of the GSM' algorithm for the pattern P and the text T gives the same results as a run of the GSM algorithm for the same pattern P and the text T.

Proof. The only difference between GSM' and GSM is that the former uses $\mathcal{W}_P$ and the latter uses $\mathcal{P}_P$ for creating D-masks D^c for every $c \in \Sigma$. The GSM algorithm searches for a maximal path f and tries to match $\sigma(f) = T_{[j,j+|P|-1]}$ The only difference between algorithms is that GSM' can create a maximal path f' which leads through vertices which represent a swap. But since the swapped vertices has identical labels then $\sigma(f) = \sigma(f')$. Therefore GSM and GSM' give the same results. □

Now we know that it is not a problem when we use tokens in the W-Graph since the swap of identical symbols does not cause different behaviour of the algorithm.

We define WGSM algorithm as an alteration of the GSM' algorithm. WGSM takes into consideration wildcard symbols and it deals with each of these tokens in the following way.

First of all we define a position i as a token position in the P. Therefore ?, $*$ and even [ ] tokens have their unique position.

We solve ? token while preprocessing. For each ? token at a position i we set $D_i^c := 1$ for each $c \in \Sigma$. This way every symbol will match the pattern at the position i. This alteration of the preprocessing changes worst case performance since for each ? tokens we have to do $|\Sigma|$ steps.

To solve bracket token $[v_1 v_2 \dots v_n]$ at a position i we need to set D_i^c for each $c \in v_1, v_2, \dots, v_n$. For example for a pattern $aba[ab]c$ we add 1 to D_a and D_b on the position 4. Note that c is on the position 5 because the whole bracket token is on position 4. This does not make preprocessing take longer because each symbol is listed in the brackets so we can read them and set D-masks appropriately.

To solve inverse bracket token $[\,!\, v_1 v_2 \dots v_n]$ at position i we need to set D_i^c for each $c \in \Sigma \setminus v_1, v_2, \dots, v_n$. For example for a pattern $aba[!a]c$ and alphabet $\Sigma = \{a, b, c, d\}$ we set $D_4^b := 1$, $D_4^c := 1$ and $D_4^d := 1$. In the worst case this preprocessing operation can take up to $O(|\Sigma|)$ time for one position. As well as for normal brackets symbols, inverse brackets are counted as taking only one position in the pattern.

To solve star $*$ symbol we need quite a different approach then for the other wildcard symbols. We cannot solve $*$ while preprocessing since we do not know how many symbols will $*$ match which is one of the basic assumptions of GSM. For $*$ symbol we need to make sure that it does not only send prefix match signal to its successors but it also maintains prefix match signal for itself. Furthermore we need to be able to skip $*$ symbol since it can represent 0 symbols of the alphabet. To deal with all of these problems we introduce *signal bit mask* S-mask.

We define S-mask S as:

$$S_i = \begin{cases} 1 & P_i = * \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

Example 8. For a pattern $a{*}?[ba] * c$ we get D-masks and S-mask as shown in Table 4.1.

Now we need to define the following operations:

- propagate signal along the edges in the same step when signal arrived,
- hold signal about prefix match.

Sending signal right after getting allows the token to be skipped. We can do that by sending signal twice. First time signal is sent normally. But the

Table 4.1: Masks initialization for the pattern $P = a{*}?[ba]*c$

P_i	D_a	D_b	D_c	S
a	1	0	0	0
$*$	1	1	1	1
?	1	1	1	0
$[ba]$	1	1	0	0
$*$	1	1	1	1
c	0	0	1	0

second time it will be sent only from all vertices for which $\sigma(v) = *$. We can get those vertices by using vector S. This way when $*$ receives the signal it can propagate it further.

To hold the signal we just need to know where signal is and which vertices should maintain that signal until next step. This can be done by using vector S in a certain way.

To make WGSM algorithm more intuitive we first show how to solve Wildcard matching problem using Shift-And algorithm and then we show how WGSM solves Wildcard Swap Matching problem.

4.1 Wildcard matching with Shift-And algorithm

First of all we need to change the preprocessing. We define D-masks D as follows:

$$D_i^c = \begin{cases} 1 & \text{if token } P_i \text{ is} \begin{cases} \text{a symbol then } c = P_i \\ \text{? token then } true \\ \text{* token then } true \\ \text{bracket token then } c \in P_i \\ \text{inverse bracket token then } c \in \Sigma \setminus P_i \end{cases} \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

Now we have to prepare the S-mask

$$S_i = \begin{cases} 1 & P_i = *, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

To get a prefix match signal on positions where $*$ token is we do this:

$$(R^j \ \& \ S). \tag{4.4}$$

We define a PS (propagate stars) operation, which means propagate signal along the edges only from vertices labelled with $*$ token, as follows:

$$PS(R^j) = R^j \mid (LSO(R^j \ \& \ S) \ \& \ D^{T_j}). \tag{4.5}$$

Everything is now ready to begin the main algorithm. We set $R^0 := 0$ and begin the computation of R^j for steps $j = 1, 2, \ldots$ using the following equations:

$$R^{'i+1} = LSO(R^i) \ \& \ D^{T_{i+1}} \mid (R^i \ \& \ S) \tag{4.6}$$

$$R^{i+1} = PS(R^{'i+1}) \tag{4.7}$$

We check at the end of each step j if there is a match by checking if $R_p^j = 1$. Note that the algorithm will not return beginning of the match but the end because it is not easy to deduce on which position the match started due to unpredictability of how many symbols will match $*$ token.

4.2 WGSM algorithm

To define the WGSM algorithm we will use notions from Section 4.1. First the algorithm prepares the D-masks in the following way. We define D as follows:

$$D_i^c = \begin{cases} 1 & \text{if token } P_i \text{ is } \begin{cases} \text{a symbol then } c = P_i \\ \text{? token then } true \\ \text{* token then } true \\ \text{bracket token then } c \in P_i \\ \text{inverse bracket token then } c \in \Sigma \setminus P_i \end{cases} \\ 0 & \text{otherwise,} \end{cases} \tag{4.8}$$

and since we use the W-Graph we can set D-masks as follows

$$DU = LShift(D), \tag{4.9}$$

$$DM = D, \tag{4.10}$$

$$DD = RShift(D). \tag{4.11}$$

Next we prepare the S-masks in a very similar way as D-masks

$$S_i = \begin{cases} 1 & P_i = *, \\ 0 & \text{otherwise,} \end{cases} \tag{4.12}$$

Since this mask will be used with W-Graph we will need a version for each row of the W-Graph. We can shift this mask each step or we can prepare mask for each row in advance as follows.

$$SU = LShift(S) \tag{4.13}$$

$$SM = S \tag{4.14}$$

$$SD = RShift(S) \tag{4.15}$$

Where *LShift* and *RShift* are left and right shift by one respectively.

Next we define vectors which will hold prefix match informations as RU, RM and RD. For the sake of simplicity we may refer to these vectors as RX. We denote vector value in the j-th step as RX^j.

To get a prefix match signal on positions where $*$ token is we do the following:

$$(RU \ \& \ SU), \tag{4.16}$$

$$(RM \ \& \ SM), \tag{4.17}$$

$$(RD \ \& \ SD), \tag{4.18}$$

to shorten this notation we refer to set of these operations as:

$$(RX \ \& \ SX). \tag{4.19}$$

We define a PS (propagate stars) operation, which means propagate signal along the edges only from vertices labelled with $*$ token, as follows:

$$PS(RU^j) = RU^j \mid (LSO(RU^j \ \& \ SU) \ \& \ DU^{T_j}), \tag{4.20}$$
$$PS(RM^j) = RM^j \mid (LSO(RM^j \ \& \ SM) \ \& \ DM^{T_j}), \tag{4.21}$$
$$PS(RD^j) = RD^j \mid (LSO(RD^j \ \& \ SD) \ \& \ DD^{T_j}). \tag{4.22}$$

Again, to simplify our notation we may refer to set of these operations as:

$$PS(RX^j) \tag{4.23}$$

Everything is now ready for the main part of the WGSM algorithm. But before we define the exact operations we address one problem with Wildcard Swap Matching problem. It might not be obvious immediately but due to the properties of the problem it can happen that some swap version of the pattern P contains up to three $*$ tokens next to each other. This may happen when the pattern P contains $*$ tokens on positions i, $i+2$ and $i+4$ as depicted in 4.1 and star on the position i swaps to $i+1$ and star on the position $i+4$ swaps to $i+3$. This causes the problem because we need to be able to skip all three $*$ tokens in the same step. We call this the *star issue*. We can solve this issue in two different ways.

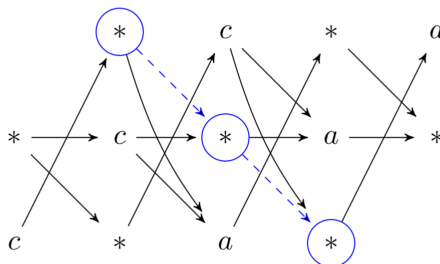

Figure 4.1: The W-Graph with three $*$ tokens next to each other

The first approach to solve the star issue is that we do the PS operation three times instead of only once. This is very easy to do, but it may do more operations then necessary. Not every pattern contains star tokens in a configuration which requires three PS operations.

The solution which uses the first approach of solving the star issue is defined as follows. We set $RX^0 := 0$ and begin the computation of RX^j for

steps $j = 1, 2, \ldots$ using the following equations.

$$RX_1{}^{i+1} = (LSO(RX^i)\ \&\ D^{T_{i+1}})\ |\ (RX^i\ \&\ S) \tag{4.24}$$

$$RX_2{}^{i+1} = PS(RX_1{}^{i+1}) \tag{4.25}$$

$$RX_3{}^{i+1} = PS(RX_2{}^{i+1}) \tag{4.26}$$

$$RX^{i+1} = PS(RX_3{}^{i+1}) \tag{4.27}$$

Since each $*$ token has D-masks set to 1 for every symbol we can omit most of the filter operations and do only one filter operation at the end of each step.

We check at the end of each step j if there is a match by checking if either $RU_p^j = 1$ or $RM_p^j = 1$. Note that the algorithm will not return beginning of the match but the end because it is not easy to deduce on which position the match started due to unpredictability of how many symbols will match $*$ token.

The second approach to solve the star issue is more complicated and we will describe just for the sake of completeness. At the end of each step we check if there is a new signal in any vertex labelled with $*$ token. By a new signal in a vertex v we mean that in j-th step $I^j(v) = 1$ but $I^{j-1}(v) = 0$. If so we need to run PS operation and repeat this process. If not we know that all signals of vertices for which $I^{j-1}(v) = 1$ were already sent with the last signal propagation.

Example 9. Let there be a pattern $P = *c*a*$ and a text $T = bca$ of length p and t respectively. We use the WGSM algorithm to find out on which positions P swap matches T. In this example the WGSM uses first approach to solve the star issue.

The algorithm uses W-Graph 4.1 to create DX^c for every $c \in \Sigma$ and S which are shown in Tables 4.2 and 4.3. The run of the algorithm is in Table 4.4.

Table 4.2: WGSM D-masks initialization for $P = *c*a*$

i	P_{i-1}	DU^a	DU^b	DU^c	P_i	DM^a	DM^b	DM^c	P_{i+1}	DD^a	DD^b	DD^c
1		0	0	0	$*$	1	1	1	c	0	0	1
2	$*$	1	1	1	c	0	0	1	$*$	1	1	1
3	c	0	0	1	$*$	1	1	1	a	1	0	0
4	$*$	1	1	1	a	1	0	0	$*$	1	1	1
5	a	1	0	0	$*$	1	1	1		0	0	0

In Table 4.4 we see that $RU_p^3 = 1$ and therefore WGSM algorithm reports a match ending at position 3.

Table 4.3: WGSM S-masks initialization for $P = *c * a*$

i	P_{i-1}	SU	SM	SD
1	$*$	0	1	0
2	c	1	0	1
3	$*$	0	1	0
4	a	1	0	1
5	$*$	0	1	0

Table 4.4: WGSM algorithm execution for $P = *c * a*$ and $T = aca$

i	RX^0	LSO	PS	PS	PS	D^a	RX^1
1	0 0 0	1 1 1	1 1 1	1 1 1	1 1 1	0 1 0	0 1 0
2	0 0 0	0 0 0	1 1 0	1 1 0	1 1 0	1 0 1	1 0 0
3	0 0 0	0 0 0	0 0 0	0 0 1	0 0 1	1 1 0	0 0 0
4	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	1 1 1	0 0 0
5	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 1 1	0 0 0
i	RX^1	LSO	PS	PS	PS	D^c	RX^2
1	0 1 0	1 1 1	1 1 1	1 1 1	1 1 1	1 1 0	1 1 0
2	1 0 0	1 1 0	1 1 0	1 1 0	1 1 0	1 1 1	1 1 0
3	0 0 0	0 0 1	0 0 1	0 0 1	0 0 1	0 1 1	0 0 1
4	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	1 0 1	0 0 0
5	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 1 0	0 0 0
i	RX^2	LSO	PS	PS	PS	D^a	RX^3
1	1 1 0	1 1 1	1 1 1	1 1 1	1 1 1	0 1 0	0 1 0
2	1 1 0	1 1 1	1 1 1	1 1 1	1 1 1	1 0 1	1 0 1
3	0 0 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 0	1 1 0
4	0 0 0	1 1 0	1 1 0	1 1 0	1 1 0	1 1 1	1 1 0
5	0 0 0	0 0 0	0 0 1	0 0 1	0 0 1	0 1 1	0 0 1

Conclusion and further research

In this thesis we described recently found flaw in the Smalgo-I and Smalgo-II algorithms for Swap Matching problem. Since the model used in these algorithms is very intuitive we devised an efficient algorithm called GSM which uses bit parallelism. This algorithm represents Swap Matching counterpart to Shift-And for standard pattern matching. Finally we have shown that thanks to Graph theoretic model we can easily generalize GSM to wildcards.

There is one known variant of Swap Matching problem called Approximate Swap Matching. In this variant we search for swap matches while counting how many swaps happened. It has been already studied in [6] where authors gave a solution to this problem.

During the time we studied this problem we came up with some variants of the Swap Matching problem which might be interesting for further research.

1. Swap Matching problem where only symbols which are k symbols apart can swap. This problem can be again divided into two different variants.
   a) swaps cannot overlap - symbols between swapped symbols must not be swapped
   b) swaps can overlap
2. Swap Matching problem where k adjacent symbols can swap to any permutation
3. Swap Matching variant closely related to approximate Swap Matching is when we are not allowed to swap only k number of times.

To solve 1b we can divide the patten into groups and run matching algorithm on each of them separately. The text is distributed into each group and when all groups give a match in a specific way we got a match.

We believe that approach through Graph theoretic model could be used to devise algorithms for other variants of Swap Matching problem since Graph theoretic model can be altered to given needs.

Bibliography

[1] Ahmed, P.; Iliopoulos, C. S.; Islam, A. S.; etc.: The swap matching problem revisited. *Theoretical Computer Science*, volume 557, no. 0, 2014: pp. 34–49, ISSN 0304-3975.

[2] Amir, A.; Aumann, Y.; Landau, G.; etc.: Pattern matching with swaps. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, Oct 1997, ISSN 0272-5428, pp. 144–153.

[3] Amir, A.; Cole, R.; Hariharan, R.; etc.: Overlap matching. *Information and Computation*, volume 181, no. 1, 2003: pp. 57–74, ISSN 0890-5401.

[4] Amir, A.; Landau, G. M.; Lewenstein, M.; etc.: Efficient special cases of Pattern Matching with Swaps. *Information Processing Letters*, volume 68, no. 3, 1998: pp. 125–132, ISSN 0020-0190.

[5] Campanelli, M.; Cantone, D.; Faro, S.: A New Algorithm for Efficient Pattern Matching with Swaps. In *Combinatorial Algorithms*, *Lecture Notes in Computer Science*, volume 5874, edited by J. Fiala; J. Kratochvíl; M. Miller, Springer Berlin Heidelberg, 2009, ISBN 978-3-642-10216-5, pp. 230–241.

[6] Campanelli, M.; Cantone, D.; Faro, S.; etc.: An Efficient Algorithm for Approximate Pattern Matching with Swaps. In *Proceedings of the Prague Stringology Conference 2009*, edited by J. Holub; J. Žďárek, Czech Technical University in Prague, Czech Republic, 2009, ISBN 978-80-01-04403-2, pp. 90–104.

[7] Cantone, D.; Faro, S.: Pattern Matching with Swaps for Short Patterns in Linear Time. In *SOFSEM 2009: Theory and Practice of Computer Science*, *Lecture Notes in Computer Science*, volume 5404, edited by M. Nielsen; A. Kučera; P. Miltersen; C. Palamidessi; P. Tůma; F. Valencia, Springer Berlin Heidelberg, 2009, ISBN 978-3-540-95890-1, pp. 255–266.

[8] Charras, C.; Lecroq, T.: *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004, ISBN 0954300645.

[9] Faro, S.: Swap Matching in Strings by Simulating Reactive Automata. Proceedings of the Prague Stringology Conference 2013, 2013, pp. 7–20.

[10] Iliopoulos, C.; Rahman, M.: A New Model to Solve the Swap Matching Problem and Efficient Algorithms for Short Patterns. In *SOFSEM 2008: Theory and Practice of Computer Science*, *Lecture Notes in Computer Science*, volume 4910, edited by V. Geffert; J. Karhumäki; A. Bertoni; B. Preneel; P. Návrat; M. Bieliková, Springer Berlin Heidelberg, 2008, ISBN 978-3-540-77565-2, pp. 316–327.

[11] Muthukrishnan, S.: New results and open problems related to non-standard stringology. In *Combinatorial Pattern Matching*, *Lecture Notes in Computer Science*, volume 937, edited by Z. Galil; E. Ukkonen, Springer Berlin Heidelberg, 1995, ISBN 978-3-540-60044-2, pp. 298–317.

Appendix A

Acronyms

GSM Graph Swap Matching

WGSM Wildcard graph Swap Matching

CS Cross sampling

BMA Basic matching algorithm

DAG Directed acyclic graph

FFT fast Fourier transform

BDM Backward directed acyclic word graph matching

Appendix **B**

Contents of enclosed CD

On attached CD you can find a implementation of Smalgo-I algorithm and GSM algorithm. You can try these algorithms by running the `example.out` or by compiling the sources yourself with `make` command.

The input should consist of two strings which are made of letters a, b, c and d. First string is the pattern and the second string is the text. To check how Smalgo-I give false positives we recommend trying input `abab aaba`.

The example output shows on which positions algorithms find their matches.

```
readme.txt ........................ the file with CD contents description
exe ...................................... the directory with executables
  example ...................... GSM and Smalgo-I example executable
src ...................................... the directory of source codes
  example .................. GSM and Smalgo-I implementation sources
  thesis .............. the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
  thesis.ps ............................. the thesis text in PS format
```