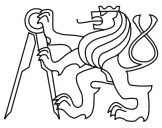
Insert here your thesis' task.

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



Master's thesis

Dead zone tree pattern matching in trees

Bc. Robin Obůrka

Supervisor: Ing. Jan Trávníček

22nd April 2016

Acknowledgements

I would like to thank the supervisor of my thesis for huge number of valuable tips, ideas, observations and comments.

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 accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 22nd April 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Obůrka, Robin. *Dead zone tree pattern matching in trees.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

V práci jsou představeny dva nové algoritmy pro vyhledávání ve stromech — sousměrný algoritmus (založený na algoritmu Morris–Pratt) a algoritmus na principu mrtvých zón. Algoritmy naleznou všechny výskyty daného stromového vzorku, které odpovídají vstupnímu stromu. Vzorek i vstupní strom jsou použity v linearizované podobě. Algoritmy používají podobné principy jako jejich řetězcové alternativy, které jsou podle potřeby modifikované. Velikost pomocné struktury, která je zkonstruovaná pro sousměrný algoritmus, je lineární vzhledem k velikosti vzorku. Algoritmus na principu mrtvých zón používá dvě pomocné struktury, jedna je opět lineární vzhledem k velikosti vzorku a druhá je lineární vzhledem k velikosti abecedy. Algoritmy jsou porovnány s doposud nejvýkonnějšími existujícími algoritmy, které jsou založeny na konečných stromových automatech, *stringpath* vyhledávání a s protisměrným algoritmem pro vyhledávání ve stromech. Měření ukazují, že dopředný algoritmus pro vyhledávání ve stromech tyto algoritmy výkonem překonává a algoritmus na principu mrtvých zón je s nimi srovnatelný. Jejich časová složitost je z teoretického úhlu pohledu o něco horší než u jejich řetězcových alternativ ale předpokládá se, že bude dále vylepšena. Pro sousměrný algoritmu může být během samotného vyhledávání počet porovnání symbolů v nejlepším případě lineární a v případě algoritmu na principu mrtvých zón dokonce sub-lineární.

Klíčová slova vyhledávání ve stromech, sousměrné vyhledávání, protisměrné vyhledávání, stromy, arbologie, Knuth-Morris-Pratt, vyhledávání na principu mrtvých zón

Abstract

A new Forward (Morris–Pratt–like) and a new Dead-zone tree pattern matching algorithms for ordered trees are presented. The algorithms find all occurrences of a single given tree pattern which match an input tree. They make use of linearisations of both the given pattern and the input tree. The algorithms use modified but similar approaches to their string equivalents. The size of the data structure constructed for the Forward tree pattern matching algorithm is linear in the size of the pattern tree. The Dead-zone tree pattern matching algorithm is using two data structures of sizes linear in the size of the alphabet and pattern tree, respectively. Algorithms were compared with best performing previously existing algorithms based on a (non-linearised) tree pattern matching using finite tree automata, stringpath matchers, and a Backward tree pattern matching algorithm. Measurements show that the Forward tree pattern matching algorithm outperforms these for single pattern matching and the Dead-zone tree pattern matching algorithm is comparable. Their time complexity properties are from the teoretical point of view decreased in coparison to their string equivalentsbut it is expected to improve. During matching, the number of symbol comparisons can be even linear in the size of the input tree in the best case in case of the Forward tree pattern matching algorithm and even sub-linear in case of the Dead-zone pattern matching algorithm.

Keywords tree pattern matching, backward pattern matching, forward pattern matching, trees, arbology, Knuth-Morris-Pratt, dead-zone matching

Contents

List of Figures

List of Tables

Introduction

This work belongs to a new algorithmic discipline called *Arbology*. The Arbology research group was set up in 2008 at Czech Technical University in Prague. The name *Arbology* is inspired by a Spanish word "Arbol" — the tree. The main point of interest of this group is applying the well-known stringology algorithms to trees; which is in fact possible since trees can be represented as string [1]. Stringology is a traditional discipline which deals with string processing. A lot of efficient algorithms for many basic tasks were developed already – for example pattern matching or compression. Therefore, the effort to apply these algorithms (with some care) to trees represented as strings sounds very reasonable.

The goal of my thesis is to find some way how to apply quiet recent Dead-zone algorithm to trees. The Dead-zone algorithm is using very elegant idea of minimizing the size of a live-zone (zone where an occurrence of a pattern can start) faster than traditional backward pattern matching algorithms. Every match attempt bring some information where an occurrence of a pattern can't start in a subject. In essence it is a framework into which you can plug any pattern matching algorithm. The better and efficient the algorithm, the quicker the progressing of live-zones minimization. It works with the naive pattern matching algorithm as well. Of course, there are still other approaches like Knuth–Morris–Pratt, Boyer–Moore and a other well-known algorithms.

The very immediate predecessor of my work is the article *Backward Linearised Tree Pattern Matching* published in LATA 2015 [2]. A Boyer–Moore-like algorithm for trees defined there and it is actually the first half of the solution of the Dead-zone algorithm. The second part — some forward pattern matching algorithm — must be defined and it is therefore a sub-goal of this thesis. Putting these two halves together is then straightforward. It will be somewhat technical but it is straightforward. Of course, details will be discussed later.

Chapter 1

Basic Notion

1.1 Alphabet, string

An *alphabet* is a finite nonempty set of *symbols*. A *ranked alphabet* is a finite nonempty set of symbols each of which has a unique nonnegative *arity* (or *rank*). Given a ranked alphabet $\mathcal{A}$, the arity of a symbol $a \in \mathcal{A}$ is denoted $Arity(a)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \ldots, p$ are called nullary (constants), unary, binary, ..., p-ary symbols, respectively. I assume that $\mathcal{A}$ contains at least one constant. In the examples I use numbers at the end of identifiers for a short declaration of symbols with arity. For instance, $a2$ is a short declaration of a binary symbol a. I use $|\mathcal{A}|$ notation for the size of set $\mathcal{A}$.

A *string* x is a sequence of i symbols $s_1 s_2 s_3 \ldots s_i$ from a given alphabet, where i is the size of the string. A sequence of zero symbols is called the *empty string*. The empty string is denoted by symbol ε. [2]

1.2 Tree, tree patterns

Based on concepts and notations from graph theory [3]:

An *graph* G is a pair (N, R), where N is a set of nodes and R is a set of edges such that each element of R is of the form (f, g), where $f, g \in N$. This element will indicate that, for node f, there is an edge between node f and node g.

A *directed graph* G is a graph, where each element of R of the form (f, g) indicates that, there is an edge leaving node f and entering node g. This edge is ordered from f to g. An *undirected graph* G is a graph in which no such ordering of edges is given.

A sequence of nodes $(f_0, f_1, \ldots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *labelling* of an ordered graph $G = (N, R)$ is a mapping of N

into a set of labels. In the examples we use a_f for a short declaration of node f labelled by symbol a.

A directed graph is *connected* if there exists a path from f_u to f_v for each pair of nodes (f_u, f_v), $u \neq v$, of the graph.

A *cycle* is a path $(f_0, f_1, \ldots, f_n)$ in which $f_0 = f_n$.

Given a node f of a directed graph, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in N$. By analogy, the *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in N$.

A *tree* is a connected directed graph without any cycle. The tree is assumed to have at least one node. A *rooted tree* t is a tree with a special node $r \in N$, called the *root*.

The rooted tree t can be also defined by following:

(1) $r \in N$ (root) has in-degree 0,
(2) all other nodes of t have in-degree 1,
(3) there is just one path from the root r to every $f \in N$, where $f \neq r$.

Nodes of a tree with out-degree 0 are called *leaves*.

A *labelled and rooted tree* is a tree with the additional property: (4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$, where $\mathcal{A}$ is an alphabet.

A node g is a *direct descendant* of node f if a pair $(f, g) \in R$.

An *ordered, labelled and rooted tree* is a labelled and rooted tree where direct descendants of a node f are ordered.

A *ordered, ranked, labelled and rooted tree* is a labelled and rooted tree labelled by symbols from a ranked alphabet and where the out-degree of a node f labelled by symbol $a \in \mathcal{A}$ equals $Arity(a)$. Nodes labelled by nullary symbols (constants) are leaves.

Throughout the text shorthand *ranked tree* will be used in context of ordered, ranked, labelled and rooted tree and *unranked tree* in context of ordered, labelled and rooted tree.

1.3 Linear notations and their properties

The *prefix notation* $pref(t)$ of a ranked tree t is defined as follows:

1. $pref(a) = a0$ if a is a leaf,
2. $pref(t) = an\ pref(b_1)\ pref(b_2) \ldots pref(b_n)$, where a is the root of tree t, $n = Arity(a)$ and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

The *prefix bar notation* $pref\_bar(t)$ of a unranked tree t is defined as follows:

1. $pref\_bar(a) = \ a \uparrow$ if a is a leaf,
2. $pref\_bar(t) = \ a\ pref\_bar(b_1)\ pref\_bar(b_2) \ldots pref\_bar(b_n) \uparrow$, where a is the root of tree t and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

The *postfix notation* $post(t)$ of a ranked tree t is defined as follows:

(a) Tree t_{1r} over a ranked alphabet (b) Tree t_{1u} over an unranked alphabet

Figure 1.1: Tree t_{1r} from Examples 1.3.1 and 1.3.2

1. $post(a) = a0$ if a is a leaf,
2. $post(t) = post(b_1)\ post(b_2) \ldots post(b_n)\ an$, where a is the root of tree t, $n = Arity(a)$ and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

The *postfix bar notation* $post\_bar(t)$ of a unranked tree t is defined as follows:

1. $post\_bar(a) = \ a \uparrow$ if a is a leaf,
2. $post\_bar(t) = \ a\ post\_bar(b_1)\ post\_bar(b_2) \ldots post\_bar(b_n) \uparrow$, where a is the root of tree t and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

Let $w = a_1 a_2 \ldots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the *arity checksum* $ac(w) = arity(a_1) + arity(a_2) + \ldots + arity(a_m) - m + 1 = \sum_{i=1}^{m} arity(a_i) - m + 1$. Let $pref(T)$ and w be a tree T in prefix notation and a substring of $pref(T)$, respectively. Then, w is the prefix notation of a subtree of T, if and only if $ac(w) = 0$, and $ac(w_1) \geq 1$ for each proper prefix w_1 of w (i.e. $w = w_1 x$, $x \neq \varepsilon$) [4].

Example 1.3.1. Consider a ranked alphabet $\mathcal{A} = \{a2,\ a1,\ a0\}$. Consider an ordered, ranked, labelled and rooted tree $t_{1r} = (\{a2_1,\ a2_2,\ a0_3,\ a1_4,\ a0_5,\ a1_6,\ a0_7\},\ R_{t1r})$ over alphabet $\mathcal{A}$, where $R_{t1r} = \{(a2_1, a2_2), (a2_1, a1_6), (a2_2, a0_3), (a2_2, a1_4), (a1_4,\ a0_5), (a1_6, a0_7)\}$. Tree t_{1r} in prefix notation is $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$. Trees can be represented graphically, as is done for tree t_{1r} in Figure 1.1a.

Example 1.3.2. Consider an unranked alphabet $\mathcal{A} = \{a\}$. Consider an ordered, labelled and rooted tree $t_{1u} = (\{a_1,\ a_2,\ a_3,\ a_4,\ a_5,\ a_6,\ a_7\},\ R_{t1u})$ over an alphabet $\mathcal{A}$, where $R_{t1u} = \{(a_1, a_2), (a_1, a_6), (a_2, a_3), (a_2, a_4), (a_4, a_5), (a_6, a_7)\}$. Tree t_{1u} in prefix bar notation is $pref\_bar(t_{1u}) = a\ a\ a\ \uparrow\ a\ a\ \uparrow\uparrow\ \uparrow\ a\ a\ \uparrow\uparrow\uparrow$. The tree t_{1u} is illustrated in Figure 1.1b.

Example 1.3.3. As another example; Consider a ranked alphabet $\mathcal{A} = \{a4,\ a0,\ b0\}$. Consider an ordered, ranked, labelled, rooted, and directed tree $t_{2r} = (\{a4_1, a4_2, a4_3, a0_4, b0_5,\ a0_6, a0_7,\ a0_8, b0_9, a0_{10}, a0_{11}, a0_{12}, b0_{13}\},\ R_{t2r})$ over an alphabet $\mathcal{A}$, where R_{t2r} is a set of the following ordered pairs:

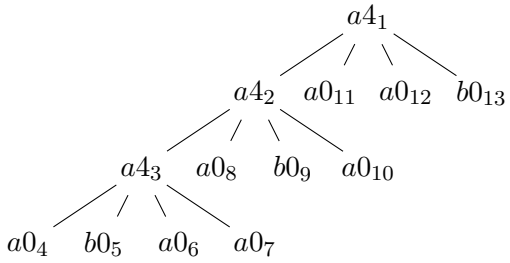


Figure 1.2: Tree t_{2r} from Example 1.3.3

$R_{t2r} = \{(a4_1, a4_2), (a4_1, a0_{11}), (a4_1, a0_{12}), (a4_1, b0_{13}), (a4_2, a4_3), (a4_2, a0_8), (a4_2, b0_9), (a4_2, a0_{10}), (a4_3, a0_4), (a4_3, b0_5), (a4_3, a0_6), (a4_3, a0_7)\}$.

Prefix notation of tree t_{2r} is $pref(t_{2r}) = a4_1a4_2a4_3a0_4b0_5a0_6a0_7a0_8b0_9a0_{10}a0_{11}$ $a0_{12}b0_{13}$. Tree t_{2r} is illustrated in Figure 1.2.

The height of a tree t, denoted by *Height(t)*, is defined as the length of the longest path leading from the root of t to a leaf of t. [2]

1.4 Subtree and pattern

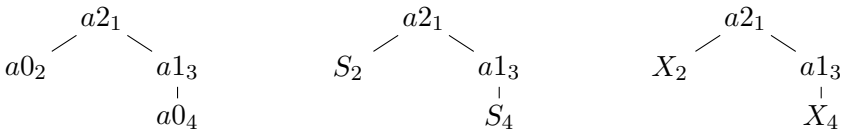
A *subtree* (a complete subtree) of a tree $t = (N, R)$ is any tree $t' = (N', R')$ such that:

1. N' is nonempty subset of N,
2. $R' = (N' \times N') \cap R$, and
3. No node of $N \setminus N'$ is a descendant of a node in N'.

To define a *tree pattern*, a special wild-card symbol $S \notin \mathcal{A}$, $Arity(S) = 0$ is used, which serves as a placeholder for any subtree. A tree pattern is defined as a labelled ordered tree over an alphabet $\mathcal{A} \cup \{S\}$. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. A tree pattern containing at least one symbol S will be called a *tree template*.

A tree pattern p with $k \geq 0$ occurrences of the symbol S *matches* a subject tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of t such that the tree p', obtained from p by substituting the subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, k$, is equal to the subtree of t_s rooted at n. Tree t_s is the *matched subtree* of tree T.

Let a tree pattern p match a subject tree t at node n and let m be the number of nodes in the matched subtree t_s. Let i be the index of node n in $pref(t) = a_1a_2 \ldots a_ia_{i+1} \ldots a_{i+m-1}a_{i+m} \ldots$. An *occurrence* of tree pattern p in subject tree t is a pair $(i, i + m)$. The pair $(i, i + m)$ is also an *occurrence* of substring $pref(t_s)$ in string $pref(t)$.



(a) Subtree p_{1r} from Example 1.4.1

(b) Tree pattern p_{2r} from Example 1.4.1

(c) Tree pattern p_{3r} from Example 1.4.1

Figure 1.3: Subtree, tree pattern, and nonlinear tree pattern (Example 1.4.1)

The *nonlinear tree pattern* also uses another special wild-card symbols $X, Y, \cdots$, not in alphabet $\mathcal{A}$. These symbols serve as placeholders for specific subtrees. Every occurrence of a symbol X, Y, $\cdots$ in a nonlinear tree pattern is matched with the same subtree. A nonlinear tree pattern has to contain at least one symbol from $\mathcal{A}$. A nonlinear tree pattern which contains at least two equal nonlinear variables will be called a *nonlinear tree template*.

A nonlinear tree pattern np with $k \geq 2$ occurrences of a nonlinear variable X *matches* a subject tree t at node n if there exists a subtree t_X of the tree t and subtrees $t_1, t_2, \ldots, t_m$ (not necessarily the same) of the tree t such that the tree np', obtained from np by substituting the subtree t_X for the i-th, $1 \leq i \leq k$, occurrences of X in np, and by substituting the subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, m$, is equal to the subtree of t rooted at n.

Example 1.4.1. Consider a ranked tree $t_{1r} = (\{a2_1,\ a2_2,\ a0_3,\ a1_4,\ a0_5,\ a1_6,\ a0_7\},\ R_{1r})$ from Example 1.3.1, which is illustrated in Figure 1.1a.

Consider a subtree p_{1r} over an alphabet $\mathcal{A}, p_{1r} = (\{a2_1, a0_2, a1_3, a0_4\},\ R_{p1r})$. Subtree p_{1r} in prefix notation is $pref(p_{1r}) = a2\ a0\ a1\ a0$ and $R_{p1r} = \{((a2_1, a0_2),\ (a2_1, a1_3)), ((a1_3, a0_4))\}$.

Consider a tree pattern p_{2r} over an alphabet $\mathcal{A} \cup \{S\}$, $p_{2r} = (\{a2_1,\ S_2,\ a1_3,\ S_4\},\ R_{p2r})$. Tree pattern p_{2r} in prefix notation is $pref(p_{2r}) = a2\ S\ a1\ S$ and $R_{p2r} = \{(a2_1, S_2), (a2_1, a1_3), (a1_3, S_4)\}$.

Consider a nonlinear tree pattern p_{3r} over an alphabet $\mathcal{A} \cup \{S, X\}$, $p_{3r} = (\{a2_1,\ X_2,\ a1_3,\ X_4\},\ R_{p3r})$. Nonlinear tree pattern p_{2r} in prefix notation is $pref(p_{2r}) = a2\ X\ a1\ X$ and $R_{p2r} = \{(a2_1, X_2), (a2_1, a1_3), (a1_3, X_4)\}$.

Tree patterns p_{1r}, p_{2r} and p_{3r} are illustrated in Figure 1.3. Tree pattern p_{1r} occurs once in tree t_{1r} — it matches at node 2 of t_{1r}. Tree pattern p_{2r} occurs twice in t_{1r} — it matches at nodes 1 and 2 of t_{1r}. Tree pattern p_{3r} occurs once in t_{1r} — it matches at nodes 2 of t_{1r}. [2]

Chapter 2

String Pattern Matching Approaches

Let me provide some basic overview of string patter matching.

The algorithms used in this thesis are just modifications of well-known algorithms that are usually used for string processing. The Dead-zone algorithm itself was originally designed for strings as well.

2.1 Introduction

String pattern matching is one of the subjects of study of *Stringology*. The name *Stringology* was coined in 1984 by computer scientist Zvi Galil for the issue of algorithms and data structures used for string processing. [5]

The classification of pattern matching *problems* is very complex. This classification could be found in [6]. There are several classifications of pattern matching *algorithms*. One classification method splits algorithms into Backward and Forward pattern matching algorithms. To be precise, there is a bidirectional algorithm too — the Dead-zone algorithm.

2.2 Backward string pattern matching algorithm

The symbols of the pattern and the text are compared in opposite direction to the shifting of the pattern in the backward string pattern matching. Matching time with the basic backward pattern matching algorithm is $O(m * n)$, where m is the size of the pattern and n is the size of the subject. No preprocessing of the pattern nor subject is needed. The basic backward pattern matching algorithm is a common base for many modifications which make it more efficient in practise.

Instead of a shift by 1 (as per line 11 of Algorithm 1), lager shifts can often be made. The length of shift depends on used heuristic.

Name: Basic backward pattern matching.
Input: A string *text* of size n and a string *pattern* of size m.
Output: Locations of the *pattern* in the *text*.

```
1  begin
2      i := 0
3      while i < n − m do
4          j := m
5          while j > 0 and pattern[j] = text[i + j] do
6              j := j − 1
7          end
8          if j = 0 then
9              OUTPUT(i + 1)
10         end
11         i := i + 1 {Length of the shift.}
12     end
13 end
```

Algorithm 1: Basic backward string pattern matching algorithm.

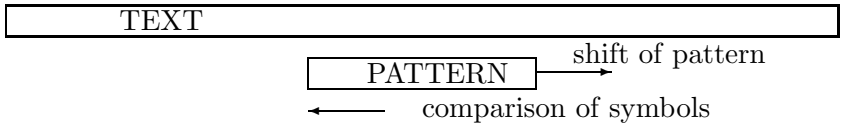


Figure 2.1: Graphical outline of algorithm 1

The algorithm 1 is modified for each heuristics by replacing statement on line 11. by some different statement.

For my thesis only the Bad character shift heuristic is important.

2.2.1 Bad character shift

This heuristic for larger shifts is used in the Boyer-Moore-Horspool algorithm [7]. Computing the length of the shift is based on one symbol aligned to the end of the pattern. This shift, a simplification of the one used by the original Boyer-Moore algorithm, has turned out to perform very well in practice.

The length of the shift is computed using the symbol aligned to the last symbol of the pattern. The shifts are stored in a *bad character shift table*.

Definition 2.2.1. Let $pattern[1..m]$ be over an alphabet $\mathcal{A}$. The bad character shift table BCS is defined as follows $BCS[a] = \min(\{m\} \cup \{j | pattern[m - j] = a \text{ and } j > 1\})$ for each $a \in \mathcal{A}$.

Example 2.2.1. Consider a pattern $p_4 = a2a1a0a1a0$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0\}$. The $BCS[a3] = 5$, $BCS[a2] = 4$, $BCS[a1] = 1$, $BCS[a0] = 2$.

Backward string matching algorithm with bad character shift is obtained by replacing statement on line 11 of algorithm 1 by statement:

$$11 \;\; i := i + BCS[text[i + m]].$$

The complexity of matching when the bad character shift is used is still $O(m * n)$ and the preprocessing time (construction of the bad character shift table) is $O(m + |\mathcal{A}|)$, where m is the size of the pattern, n is the size of the subject, and $|\mathcal{A}|$ is the size of the alphabet. The size of the bad character shift table is $\Theta(|\mathcal{A}|)$.

2.2.2 Match shift

The match shift is performed when an occurrence is found. The pattern is shifted so that it is aligned using the longest nontrivial border.

Definition 2.2.2. Let $pattern[1..m]$ be over an alphabet $\mathcal{A}$. The match shift $MS = m - length(Border(pattern))$.

2.3 Forward string pattern matching algorithms

In this case, the symbols of the pattern and the text are compared in the same direction as is the direction of shifting of the pattern. Time complexity is also the same as in case of Forward pattern matching algorithm — $O(m*n)$, where m is the size of the pattern and n is the size of the subject.

The basic forward pattern matching algorithm is in essence the same as Naive algorithm for string pattern matching generally. For shift of length 1 as on line 11 of algorithm 2 we get the Naive algorithm. Longer shifts are also possible. It depends on the used heuristic.

2.3.1 Knuth–Morris–Pratt

This algorithm uses a precomputed table to determine the length of shift. The table stores an information about the longest prefixes that are also suffixes for all suffixes or better — *borders* of all suffixes.

Definition 2.3.1. Let $x \in A^*$ be a string of length n. If border of length β is found, then $x[1..\beta] = x[n - \beta + 1..n]$. So, *border* u of string x is any proper prefix of x that equals a suffix of x.

Example 2.3.1. See Figure 2.2. Consider an attempt at a left position j, that is when the window is positioned on the text factor $y[j..j + m - 1]$. Assume that the first mismatch occurs between $x[i]$ and $y[i+j]$ with $0 < i < m$. Then, $x[0..i - 1] = y[j..i + j - 1] = u$ and $a = x[i] \neq y[i + j] = b$. [8]

Name: Basic forward pattern matching.
Input: A string *text* of size n and a string *pattern* of size m.
Output: Locations of the *pattern* in the *text*.

```
1  begin
2      i := 0
3      while i <= n − m do
4          j := 0
5          while j < m and pattern[j + 1] = text[i + j + 1] do
6              j := j + 1
7          end
8          if j = m then
9              OUTPUT(i + 1)
10         end
11         i := i + 1 {Length of the shift.}
12     end
13 end
```

Algorithm 2: Basic forward string pattern matching algorithm.

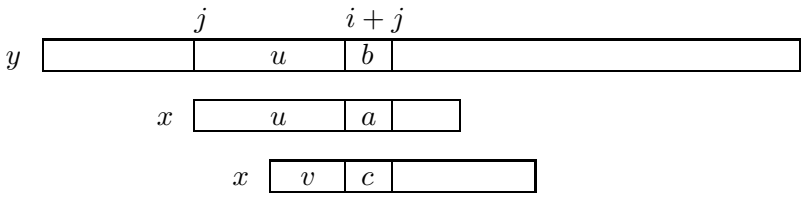


Figure 2.2: Shift in the Knuth–Morris–Pratt algorithm

Shift is based on expectation that a prefix v of the pattern matches some suffix of the u of the text. If we want to avoid another immediate mismatch, the character following the prefix v in the pattern must be different from a. The longest such a prefix v is called the *border* of u. [8]

Now, see algorithms 3 and 4.

Preprocessing function takes $O(m)$ time in respect to the size m of the pattern. It is one time process before searching phase itself.

The searching phase of Knuth–Morris–Pratt algorithm has time complexity $O(m + n)$. The algorithm makes at most $2n - 1$ comparisons during the search phase. [9, 8].

2.4 Dead-zone string pattern matching algorithm

Dead-zone algorithm works on the principle of limiting possible range of indexes in the subject where the beginnings of occurrences of the pattern can be. Dead-zone algorithm uses two important terms — dead-zone and live-zone.

Name: Preprocessing function for Knuth–Morris–Pratt
Input: A string p (pattern) of size m.
Output: Table of integers $kmpNext$ representing shifts

```
1  begin
2      i := 0
3      j = kmpNext[0] = −1
4      while i < m do
5          while j > −1 and p[i] ≠ p[j] do
6              j := kmpNext[j]
7          end
8          i := i + 1
9          j := j + 1
10         if p[i] = p[j] then
11             kmpNext[i] := kmpNext[j]
12         end
13         else
14             kmpNext[i] := j
15         end
16     end
17 end
```

Algorithm 3: Preprocessing function for Knuth–Morris–Pratt.

Name: Knuth–Morris–Pratt matching function
Input: A string p (pattern) of size m, a string s (text) of size n and $kmpNext$ array (result of preprocessing).
Output: A list of matches.

```
1  begin
2      i := j := 0
3      while j < n do
4          while i > −1 and p[i] ≠ s[j] do
5              i := kmpNext[i]
6          end
7          i := i + 1
8          j := j + 1
9          if i ≥ m then
10             OUTPUT(j − i)
11             i := kmpNext[i]
12         end
13     end
14 end
```

Algorithm 4: Knuth–Morris–Pratt matching function (different representation than algorithm 2.

Definition 2.4.1. Range of indexes in the subject where it is still possible to find the beginnings of occurrences of the pattern is called a *live-zone*.

Definition 2.4.2. The range of indexes in the subject where it is not possible to find the beginnings of occurrences of the pattern is called a *dead-zone*.

Name: Dead-zone abstract matcher
Input: A string p (pattern) of size m, a string s (text) of size n and the boundary of live zone *live_low* and *live_high*.
Output: A list of matches.

```
1  begin
2  |  Procedure dzmat(live_low, live_high)
3  |  |  if live_low ≥ live_high then
4  |  |  |  return
5  |  |  end
6  |  |  j := ⌊(live_low + live_high)/2⌋
7  |  |  i := 0
8  |  |  while (i < m) and (p[i] = s[j + i]) do
9  |  |  |  i := i + 1
10 |  |  end
11 |  |  if i = m then
12 |  |  |  OUTPUT(j)
13 |  |  end
14 |  |  if i < m then
15 |  |  |  return
16 |  |  end
17 |  |  new_dead_left := j − shift_left(i, j) + 1
18 |  |  new_dead_right := j + shift_right(i, j)
19 |  |  dzmat(live_low, new_dead_left)
20 |  |  dzmat(new_dead_right + 1, live_high)
21 end
```

Algorithm 5: Dead-zone abstract matching function.

The Dead-zone algorithm could be implemented as a simple "abstract" recursive algorithm. See algorithm 5.

Let m be the length of the pattern and let n be the length of the subject. At the beginning, live-zone is the range of indexes from the beginning of the subject to the index $n - m + 1$. The first invocation of *dzmat* is $dzmat(0, n - m + 1)$.

The rule is simple: Occurrence cannot be found on indexes too close to the end of the subject where the pattern would overflow the subject.

In the next step pick some position in the live-zone and try to found an occurrence. Regardless whether it fails, the dead-zone will emerge in the tested

area. This process generates two smaller live-zone and they are processed recursively.

The algorithm 5 is presented as a "framework". It just defines how to manipulate with dead-zones (and live-zones respectively). Shifting heuristics is hidden in the functions $shift\_left()$ and $shift\_right()$. These functions could be both the naive shifts by 1 or some heuristic of Backward or Forward pattern matching algorithms. Of course, every implementation should aim to the longest possible shifts. The faster the dead-zone grows the better performance the *dzmat* algorithm gets. [10]

Please, note some important things about algorithm 5:

Algorithm as presented works for string where the first symbol has index 0.

This is simplified version. The original version presented in [10] is using so called mapping functions to generalize order of symbols comparison while checking for occurrence. If you are interested in details, see the original article [10].

2.4.1 Example of *dzmat* algorithm's execution

Let me provide some example of *dzmat* algorithm's execution for better understanding. Example comes from [10].

Example 2.4.1. Searching the pattern $p =$ "*abracadabra*" of size $m = 11$ in the subject $s =$ "*The quick brown fox jumped over the lazy dog*" of size $n = 44$ with shifts based on *Horspool's algorithm*.

1. First invocation with a live-zone $[0, 34)$. Match attempt at 17:

   ```
   The quick brown fox jumped over the lazy dog
                   abracadabra
   ```

   There is mismatch at $i = 0$. Left/right shift by 11/11. New dead-zone is $[7, 28)$. Two live-zones will be $[0, 7)$ and $[28, 34)$.

2. Invoked with live-zone $[0, 7)$ — the left one. Match attempt at 3:

   ```
   The quick brown fox jumped over the lazy dog
      abracadabra
   ```

   There is mismatch at $i = 0$. Left/right shift by 11/11. New dead-zone is $[-7, 14)$. Two live-zones will be $[0, -7)$ (will be skipped by first condition) and $[14, 7)$ (will be skipped too).

3. Invoked with live-zone $[28, 34)$ — the right one. Match attempt at 31:

   ```
   The quick brown fox jumped over the lazy dog
                                     abracadabra
   ```

There is mismatch at $i = 0$. Left/right shift by 11/4. New dead-zone is $[21, 35)$. Two live-zones will be $[28, 21)$ (will be skipped by first condition) and $[35, 34)$ (will be skipped too).

Chapter 3

Tree Pattern Matching

The tree pattern matching is a similar problem to the string pattern matching. Trees can be represented in linear notation — i.e. as strings. An interesting fact is that is not necessary to build linear representation explicitly. It can be obtained by sequential tree traversal "on fly". The patterns in linear notation are represented by substrings of trees in the linear notation. They can contain "gaps" given by a special wild-card symbol S, which serves as a placeholder for any subtree.

The string pattern matching algorithm couldn't be used as is. The wild-card symbol S doesn't represent simple linear sequence. It represents subtree and matched subtrees may be possibly nested. The wild-card symbol S need some special care. Also, the tree pattern matching problem is more complex than the string version of this problem. There is at most n^2 distinct substrings of string of size n, whereas there is at most $2^{n-1} + n$ distinct tree patterns which match a tree of size n. [2]

There are some relevant previous results on the field of tree pattern matching problems. Many of them use some kind of tree automata. If you are interested in details, see chapter *Introduction* in [2].

3.1 Backward Tree Pattern Matching

Examples, definitions and core ideas in this section are used from [2].

This method sees the tree pattern matching problem as matching of connected subgraphs in trees. The basic idea of backward tree pattern matching for tree patterns is the same as in the string case: moving the pattern in one direction and matching symbols of tree pattern and subject tree in the opposite direction. Wild-card S occurrences must be handled in a special way. A prefix ranked bar notation of the tree is used for the purpose of extending of shifts.

Definition 3.1.1. The *prefix ranked bar notation* $pref\_ranked\_bar(t)$ of a tree t is defined as follows:

1. $pref\_ranked\_bar(S) = S \uparrow S$
2. $pref\_ranked\_bar(a) = a0 \uparrow 0$ if a is a leaf,
3. $pref\_ranked\_bar(t) = an\ pref\_ranked\_bar(b_1)\ pref\_ranked\_bar(b_2) \ldots pref\_ranked\_bar(b_n) \uparrow n$, where a is the root of the tree t, $n = Arity(a)$ and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

Definition 3.1.2. Let $\uparrow n$, where $n \geq 0$ be bar symbols of arity n. The *bar set* $\mathcal{A}_\uparrow$ is the set of all bar symbol $\uparrow n$.

3.1.1 Bad character shift table

Definition 3.1.3. Let $pattern[1..m]$ be a $pref\_ranked\_bar$ notation of a tree pattern p over an alphabet $\mathcal{A}$. The *bad character shift table* $BCS(pattern[1..m])$ for backward tree pattern matching is defined for each $a \in \mathcal{A}$:
$BCS(pattern[1..m])[a] = \min($

$$\{m\} \cup \{j : pattern[m-j] = a \text{ and } m > j > 0\} \cup$$
$$\{j + Arity(a) * 2 : pattern[m-j] = S \text{ and } m > j > 0 \text{ and } a \notin \mathcal{A}_\uparrow\} \cup$$
$$\{j - 1 : pattern[m-j] = S \text{ and } m > j > 1 \text{ and } a \in \mathcal{A}_\uparrow\})$$

The wild-card S is not in shift table BCS because this symbol cannot occur in the subject tree.

Items of the BCS table are computed as the minimum value from four formulas shown in Definition 3.1.3 where the formulas are separated by the union operation.

The first formula makes sure that the shift is not longer than the size of the pattern m. The size of a subtree hidden in wild-card S is considered as the smallest possible one, i.e. 2: one nullary symbol $a0 \uparrow 0$.

The second formula defines the minimal safe shift for symbols that occur in the pattern. The minimal safe shift for a symbol a is the distance j of the closest occurrence of the symbol a from the end of the pattern. Wild-card symbol S is considered to correspond to the smallest possible subtree again.

The third and fourth formulas define the shift for cases when a symbol a is expected to be in a subtree t_e that corresponds to wild-card S. The location of the last wild-card S from the end of the pattern is used to define the base shift length j and this shift can be prolonged by some number depending on the arity of the symbol a, see the second part of the definition. The smallest subtree t_e that contains the symbol a is rooted by a and its direct descendants are nullary symbols $b0$. For each symbol $b0$ in the subtree t_e there is also one symbol $\uparrow 0$. The base shift j is then prolonged by $2 * Arity(a)$. Any symbol from the set $\mathcal{A}_\uparrow$ can occur as the last symbol of a subtree t_e, i.e. it can be matched with $\uparrow S$. Therefore, the base shift of each bar is shortened by 1, see fourth part of the definition. The shift cannot be zero and in that case the base shift is not shortened. Note that this case would occur only for pattern $S \uparrow S$.

See Example 3.1.1.

Example 3.1.1. Consider a tree pattern p_{3r} in prefix ranked bar notation $pref\_ranked\_bar(p_{3r}) = a2\ a1\ S\ {\uparrow}S\ {\uparrow}1\ a1\ a0\ {\uparrow}0\ {\uparrow}1\ {\uparrow}2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0, {\uparrow}S\}$. Algorithm 3.1.1 constructs the following items of the BCS table.
$BCS[a3] = \min(\{10\} \cup \emptyset \cup \{13\}) = 10$, $BCS[a2] = \min(\{10\} \cup \{9\} \cup \{11\}) = 9$, $BCS[a1] = \min(\{10\} \cup \{4, 8\} \cup \{9\}) = 4$, $BCS[a0] = \min(\{10\} \cup \{3\} \cup \{7\}) = 3$,
$BCS[{\uparrow}3] = \min(\{10\} \cup \emptyset \cup \{6\}) = 6$, $BCS[{\uparrow}2] = \min(\{10\} \cup \emptyset \cup \{6\}) = 6$,
$BCS[{\uparrow}1] = \min(\{10\} \cup \{1, 5\} \cup \{6\}) = 1$, $BCS[{\uparrow}0] = \min(\{10\} \cup \{2\} \cup \{6\}) = 2$.

BCS table is constructed by algorithm 6. Firstly, algorithm finds the location of the last wild-card S. Then, the BCS table for all symbols of the alphabet is initialised to the size of the pattern. The length of the shift for all symbols of the alphabet is possibly shortened with the use of the information on the position of the last wild-card S. The arity of symbols is used to make this part of the shift function longer according to Definition 3.1.3. Finally, the length of the shift is again possibly shortened by the actual positions of symbols in the pattern.

3.1.2 Subtree jump table

The backward tree pattern matching algorithm uses operation "skip subtree" corresponding to the wild-card symbol S. Comparing symbols (labels) every time is very time consuming and inefficient. There is another one structure that works as optimisation of this operation. The structure is called SJT or *subtree jump table*.

The SJT structure contains two kinds of positions for each subtree r of a tree t. The first kind of position is the position of the first symbol of the subtree r in $pref\_ranked\_bar(r)$ notation in the $pref\_ranked\_bar(t)$ notation of the tree t as an index and the position one after the last symbol of the subtree r in $pref\_ranked\_bar(r)$ notation as a value. The second one is the position of the last symbol of the subtree r in $pref\_ranked\_bar(r)$ notation in the $pref\_ranked\_bar(t)$ notation of the tree t as an index and the position one before the first symbol of the subtree r in $pref\_ranked\_bar(r)$ notation as a value. SJT structure has the same size as the $pref\_ranked\_bar$ notation and it is constructed by algorithm 7.

Definition 3.1.4. Let t and $pref\_ranked\_bar(t)$ of length n be a tree and its prefix ranked bar notation, respectively. A *subtree jump table* $SJT(pref\_ranked\_bar(t))$ is defined as a mapping from set of integers $\{1..n\}$ into a set of integers $\{0..n{+}1\}$. If $pref\_ranked\_bar(t)$ $[i..j]$ is the prefix ranked bar notation of a subtree of tree t, then $SJT(pref\_ranked\_bar(t))[i] = j + 1$ and $SJT(pref\_ranked\_bar(t))[j] = i - 1$, $1 \leq i < j \leq n$.

Name: ConstructBCS
Input: Tree *pattern* in prefix ranked bar notation $pref\_ranked\_bar(pattern)$ of size m over alphabet $\mathcal{A}$ of the *subject* tree.
Output: The bad character shift table $BCS(pref\_ranked\_bar(pattern))$.

```
1  begin
2  |  s := m
3  |  for i := 1 to m do
4  |  |  if pref_ranked_bar(pattern)[i] = S then s = m − i;
5  |  end
6  |  foreach x ∈ A do BCS[x] = m;
7  |  foreach x ∈ A do
8  |  |  if x ∉ A↑ then shift := s + Arity(x) ∗ 2;
9  |  |  else if s >= 2 then shift := s − 1;
10 |  |  else shift := s;
11 |  |  if BCS[x] > shift then BCS[x] := shift;
12 |  end
13 |  for i := 1 to m − 1 do
14 |  |  if pref_ranked_bar(pattern)[i] ∉ {S, ↑S} and
   |  |  BCS[pref_ranked_bar(pattern)[i]] > (m − i) then
   |  |  BCS[pref_ranked_bar(pattern)[i]] := m − i ;
15 |  end
16 end
```

Algorithm 6: Construction of BCS table

Example 3.1.2. Consider a tree t_{2r} and its representation in prefix ranked bar notation $pref\_ranked\_bar(t_{2r}) = a2\ a2\ a0\ {\uparrow}0\ a0\ {\uparrow}0\ {\uparrow}2\ a2\ a0\ {\uparrow}0\ a0\ {\uparrow}0\ {\uparrow}2\ {\uparrow}2$ over alphabet $\mathcal{A} = \{a3, a2, a1,\ a0,\ {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0\}$. A visualisation presented in Table 3.1 is of the $SJT(pref\_ranked\_bar(t_{2r}))$.

3.1.3 Backward linearised tree pattern matching algorithm

The backward tree pattern matching algorithm is an extension of the string backward pattern matching algorithm shown as an algorithm 1.

The modification of the string backward matching algorithm is based on the principle that the algorithm performs also tests for wild-cards S in the pattern. The modification is in line 10 of algorithm 8, where a part of the subject tree representing a subtree is skipped when a wild-card S, represented as $S\ {\uparrow}S$, is processed. Also, two indexes, one to the pattern and the other one to the text, are needed because subtrees (which need to be skipped) are often longer than two symbols.

Name: ConstructSJT
Input: Tree t in prefix notation $pref\_ranked\_bar(t)$ of length n, index of current node $rootIndex$ default is 1, reference to an empty subtree jump table $SJT(pref\_ranked\_bar(t))$ of length n
Output: index $exitIndex$, subtree jump table $SJT(pref\_ranked\_bar(t))$

```
1  begin
2  |  index := rootIndex + 1
3  |  for i = 1 to Arity(pref_ranked_bar(t)[rootIndex]) do
4  |  |  index :=
   |  |  ConstructSJT(pref_ranked_bar(t), index, SJT(pref_ranked_bar(t)))
5  |  end
6  |  index := index + 1
7  |  SJT(pref_ranked_bar(t))[rootIndex] = index
8  |  SJT(pref_ranked_bar(t))[index − 1] = rootIndex − 1
9  |  return index
10 end
```

Algorithm 7: Construction of subtree jump table

Table 3.1: Subtree jumping table $SJT(pref\_ranked\_bar(t_{2r}))$ of tree t_{2r}

1	2	3	4	5	6	7	8	9	10	11	12	13	14
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$
15	8	5	2	7	4	1	14	11	8	13	10	7	0

Example 3.1.3. Consider a tree pattern p_{4r} in the prefix ranked bar notation $pref\_ranked\_bar(p_{4r}) = a2\ S\ \uparrow S\ S\ \uparrow S\ \uparrow 2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow S\}$ and a tree t_{2r} in the prefix ranked bar notation $pref\_ranked\_bar\ (t_{2r}) = a2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$. The $BCS[a3] = 6$, $BCS[a2] = 5$, $BCS[a1] = 4$, $BCS[a0] = 2$, $BCS[\uparrow 3] = 1$, $BCS[\uparrow 2] = 1$, $BCS[\uparrow 1] = 1$, $BCS[\uparrow 0] = 1$. A run of algorithm 8 is depicted in Table 3.2. Longer subtrees in place of wild-cards S are denoted by $S\rightarrow\ \leftarrow S$.

The run of algorithm 8 for Example 3.1.3 starts at position 6 of the $pref\_ranked\_bar(t_{2r})$. Mismatch of $\uparrow 2$ and $\uparrow 0$ results in subsequent shift by 1 symbol to align $\uparrow 0$ with position of the end of the last wild-card S in the $pref\_ranked\_bar(p4r)$. The algorithm recognises pattern match on positions 2 to 7 and shift is by 1 symbol to align $\uparrow 2$ again with the end of the last wild-card S in $pref\_ranked\_bar(p4r)$. Mismatch of $\uparrow$ 2 and $a2$ results in a shift by 5 symbol where $a2$ is not only aligned with $a2$ but also with position closes to the end of the pattern where $a2$ can be as a part of the last wild-card S. Another match is recognised and the shift is by 1 symbol where another occurrence is recognised and subsequent shift is to outside of the $pref\_ranked\_bar(t_{2r})$

Name: BackwardLTPM.
Input: The *subject* tree in $pref\_ranked\_bar(subject)$ notation of size n, the tree *pattern* in $pref\_ranked\_bar(pattern)$ notation of size m, $SJT(pref\_ranked\_bar(subject))$, and $BCS(pref\_ranked\_bar(pattern))$.
Output: Locations of occurrences of the pattern *pattern* in the tree *subject*.

```
1  begin
2      i := 0
3      while i <= (n − m) do
4          j := m
5          position := i + j
6          while j > 0 and position > 0 do
7              if pref_ranked_bar(subject)[position] =
               pref_ranked_bar(pattern)[j] then
8                  position := position − 1
9              else if pref_ranked_bar(pattern)[j] = ↑S and
               pref_ranked_bar(subject)[position] ∈ A↑ then
10                 position := SJT(pref_ranked_bar(subject))[position]
11                 j = j − 1 {Subtree skip}
12             else break;
13             j := j − 1
14         end
15         if j = 0 then output(position + 1);
16         i := i + BCS[pref_ranked_bar(subject)[i + m]]
17     end
18 end
```

Algorithm 8: Backward tree pattern matching algorithm

Table 3.2: Trace of the run of algorithm 8 for subject tree t_{2r} and tree pattern p_{4r}

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t_{2r})$
14	6	2	2	2	2	6	6	2	2	2	2	6	14	$subtree\_sizes(t_{2r})$
					$\uparrow 2$									$\uparrow 0 \neq \uparrow 2$, $shift = 1$
	$a2$	$S\rightarrow$	$\leftarrow S$	$S\rightarrow$	$\leftarrow S$	$\uparrow 2$								match, $shift = 1$
							$\uparrow 2$							$a2 \neq \uparrow 0$, $shift = 5$
							$a2$	$S\rightarrow$	$\leftarrow S$	$S\rightarrow$	$\leftarrow S$	$\uparrow 2$		match, $shift = 1$
$a2$	$S\rightarrow$				$\leftarrow S$	$S\rightarrow$						$\leftarrow S$	$\uparrow 2$	match, $shift = 1$

resulting in the end of the run of the algorithm 8.

3.1.4 Analysis

The BCS table is the only data structure needed for the algorithm and its size is $\Theta(\mathcal{A})$, where $\mathcal{A}$ is the alphabet size. The preprocessing time is $O(m + \mathcal{A})$, where m is the pattern length and $\mathcal{A}$ is the alphabet size.

Backward string pattern matching is known to perform sublinear number of comparisons of symbols on average. The modification to backward tree pattern matching requires the input tree to be read in prefix ranked bar notation. However, the algorithm still performs $\Omega(\frac{n}{m})$ comparisons of symbols, where n is the size of the input subject tree and m is the size of the given tree pattern and $O(n * m)$ comparisons of symbols as in the case of the backward string pattern matching. The lengths of the shifts depend on the position of the last wild-card S in the pattern p – the closer to the end of the pattern the last occurrence of symbol S is, the longer are the shifts performed.

Chapter 4

Solution Introduction

In this chapter, I will provide explanation and outline of the steps leading to satisfy goals of this Thesis.

4.1 Backward pattern matching directions

There is an important fact about backward pattern matching algorithm that wasn't explained yet and in not noticeable from algorithm 1.

The core idea of backward pattern matching algorithm is use of different directions of matching and shifting the pattern. Usually, it doesn't matter if the pattern is being shifted to the left with matching direction to the right or other way round. The modification of algorithm 1 is straightforward and the heuristics are working in opposite direction with small changes — some kind of inversion.

4.2 Two-anchor problem

Details about Dead-zone algorithm are provided in section 2.4 or see the original paper [10].

Dead-zone algorithm is trying to optimise a number of necessary match attempts to minimum. It uses so called live-zones. The live-zone is an area where it is possible to find the start of the pattern's occurrence. The opposite term to the live-zone is a dead-zone. The dead-zone is an area where it is *not* possible to find the start of the pattern's occurrence. As the dead-zones are growing, the live-zones are shrinking.

The speed of growing of the dead-zones is affected by the left and right shifts of the pattern. This is obvious from algorithm 5. Longer shifts means smaller live-zone and smaller live-zone reduces number of necessary match attempts. Obviously, the algorithm's efficiency strongly depends on the quality of shift functions.

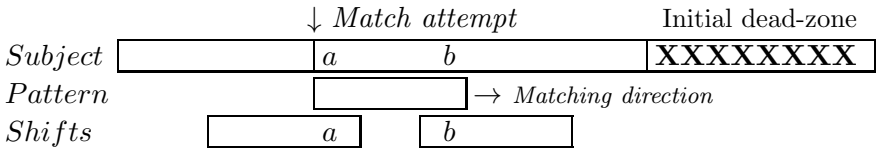


Figure 4.1: Two anchored Dead-zone

The Dead-zone algorithm typically uses some heuristic from forward or backward pattern matching algorithms as shift function. One of the original ideas in Dead-zone algorithm is using bad-character shift. See illustration on figure 4.1. If mismatch occurs while current match attempt (right direction), the algorithm performs a shift to the left according to the bad-character shift table. At the same moment the algorithm uses an inverse version of the bad-character shift table to make the shift to the right. As an anchor for the second shift the algorithm uses the end of the pattern.

So, the original Dead-zone algorithm makes two shifts in opposite directions but based on the (same, in principle) heuristics. These shifts are using two anchors — the beginning and the end of the pattern. It is possible to use this idea in *exact* string matching problem. Anyway, the same is complicated in trees with wild-card symbol S. The pattern is essentially "elastic". There is no easy way how to transpose the shift and subsequent dead-zone from the end of the pattern to the beginning of the pattern.

4.3 One-anchor solution

As I explained in previous section 4.2, the rest of the text will only focus on description of an algorithm that is using only one anchor for shifts. The beginning of the match attempt is chosen, though the end can be chosen as well.

The solution is based on two algorithms, one forward and one backward. Consider matching direction from the left to the right. Some forward pattern matching algorithms (pattern is shifted and matched in the same direction) provides the shift to the right based on its heuristics. Some backward pattern matching algorithm (pattern is shifted and matched in opposite directions) provides the shift to the left. The both algorithms need only one anchor as a base of their shifts.

There is one disadvantage. In preprocessing phase is necessary to compute two auxiliary structures for two different shift functions. On the other hand, there are still a lot of advantages. This concept is still framework, so it doesn't matter which algorithm is used exactly. There is good chance to keep efficiency

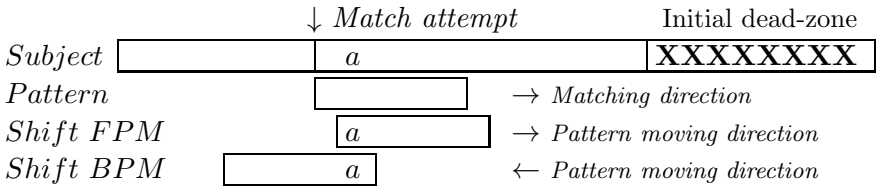


Figure 4.2: One anchored Dead-zone

of Dead-zone algorithm on trees. And finally, this is the way leading to the solution. See small demonstration on figure 4.2.

4.4 A “Battle plan”

Let me summarize the facts. Dead-zone algorithm for linearised trees, as designed in this thesis, needs one backward and one forward pattern matching algorithm. The backward one was introduced in the article *Linearised Backward Tree Pattern Matching* [2]. At this moment there is no forward matching algorithm for linearised trees. So, the first step is to design some suitable forward tree pattern matching algorithm. After that, things can be put together and Dead-zone algorithm for trees can be designed, hence Thesis goals will be satisfied.

Chapter 5

Forward Tree Pattern Matching

Let me to introduce a *Forward linearised tree pattern matching algorithm* that will be used in tree version of Dead-zone algorithm.

5.1 Basic idea

The algorithm is inspired by Knuth–Morris–Pratt algorithm for strings. Basic explanation of Knuth–Morris–Pratt algorithm is provided in section 2.3.1.

Knuth–Morris–Pratt algorithm has a heuristic based on borders (see definition 2.3.1). The algorithm's auxiliary structure is called a *border array*. In algorithms 3 and 4 is border array represented by array *kmpNext*.

5.1.1 Simplification

Algorithms 3 and 4 represent the Knuth–Morris–Pratt algorithm. They are able to avoid another immediate mismatch and they actually use so called *tagged border*. If you are interested in details, see [8].

For better illustration, I would like to show more naive variant of preprocessing function. It represents "raw" border array construction and it turns Knuth–Morris–Pratt algorithm to Morris–Pratt algorithm. See algorithm 9 (originally comes from [11]).

5.1.2 Explanation

Note that algorithm in this example indexes string from 0. All tree pattern matching algorithms are indexed from 1.

Example 5.1.1. Consider pattern $p = ABACABDE$. Table 5.1 shows the result of algorithm's 9 invocation on pattern p.

Name: Preprocessing function for Morris–Pratt
Input: A string p (pattern) of size m.
Output: Table of integers ba

```
1  begin
2      len := 0
3      i := 1
4      ba[0] := 0
5      while i < m do
6          if p[i] = p[len] then
7              len := len + 1
8              ba[i] := len
9              i := i + 1
10         end
11         else
12             if len ≠ 0 then
13                 len := ba[len − 1]
14             end
15             else
16                 ba[i] := 0
17                 i := i + 1
18             end
19         end
20     end
21 end
```

Algorithm 9: preprocessing function for Morris–Pratt.

Table 5.1: Result of Morris–Pratt preprocessing function on pattern from example 5.1.1

0	1	2	3	4	5	6	7
A	B	A	C	A	B	D	E
0	0	1	0	1	2	0	0

Table 5.2: Visualisation of shifts based on border array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
subject	D	A	A	B	A	C	A	B	F	G	D	D	E	D
match attempt			A	B	A	C	A	B	D	E				
next attempt							A	B	A	C	A	B	D	E

Example 5.1.2. Consider a pattern p from example 5.1.1 and a subject $s = DAABACABFGDDED$. Table 5.2 shows the execution of Morris–Pratt algorithm – match attempts and shifts based on border array.

Let me explain how the shifts based on border array work. See examples 5.1.1 and 5.1.2.

The table 5.2 shows a mismatch at index 8. The longest matching sequence is at index 7 and it corresponds to an index 5 in pattern. The table 5.1 shows for a mismatch at index 5 a value 2. This value represent the length of the border and the index of the next attempt in the pattern.

The next attempt starts with a comparison at next position in the subject and at position 2 in the pattern. It is not necessary to compare symbols on indexes 0 and 1 — they will match anyway. And this is what the border array encodes. Border array is just an auxiliary data structure which provides the answer in constant time (after preprocessing phase, of course).

If one wants to design Morris–Pratt–like algorithm for trees, one needs to define a mechanism that is similar to borders and border arrays.

5.2 Theoretical background

This algorithm must work together with the backward tree pattern matching algorithm introduced in section 3.1. Naturally, the algorithms must be compatible at lowest level. They must share the same linearization mechanism and used notation. So, see the important definitions 3.1.1 and 3.1.2 in section 3.1.

Also, It needs the *Subtree jump table* (section 3.1.2).

5.3 Linearised tree border

Linearised tree border is substitution of string borders for Morris–Pratt–like algorithm on trees.

Definition 5.3.1. The relation *matches* is recursively defined for strings S_1 and S_2 where $S_1 = pref\_ranked\_bar(p)$, hence the whole pattern, and a factor

the whole pattern of S_2:

$$
\begin{array}{lll}
S_1 = xS'_1 & S_2 = xS'_2 & and \;\; S'_1 \; matches \; S'_2, \\
S_1 = SS'_1 & S_2 = SS'_2 & and \;\; S'_1 \; matches \; S'_2, \\
S_1 = x_1...x_mS'_1 & S_2 = SS'_2 & and \;\; ac(x_1...x_m) = 0 \\
 & & and \;\; \forall k, 1 < k < n, ac(x_1...x_k) \not\geq 0 \\
 & & and \;\; S'_1 \; matches \; S'_2, \\
S_1 = SS'_1 & S_2 = x_1...x_mS'_2 & and \;\; ac(x_1...x_m) = 0 \\
 & & and \;\; \forall k, 1 < k < n, ac(x_1...x_k) \not\geq 0 \\
 & & and \;\; S'_1 \; matches \; S'_2, \\
S_1 = SS'_1 & S_2 = x_1...x_m & and \;\; ac(x_1...x_m) > 0 \\
 & & and \;\; \forall k, 1 < k < n, ac(x_1...x_k) \neq 0, \\
S_1 = \varepsilon \; or \; S_2 = \varepsilon & &
\end{array}
$$

Definition 5.3.2. Consider a *pattern* in $pref\_ranked\_bar()$ notation. The linearised tree border array β contains values:

$$\beta[i] = i - min(j : pattern \; matches \; pattern[j..i + j - 1])$$

The linearised tree border array is constructed by algorithm 11. The relation *matches* is determined by algorithm 10.

The basic idea is the same as in case of strings. If the mismatch occurs after several symbol comparisons, the linearised tree border allows to skip the part of subject where it is sure that the pattern can't start. The shift function is designed in a way that it even shifts between subtrees.

Example 5.3.1. Consider a tree pattern p and its prefix ranked bar notation $pref\_ranked\_bar(p) = a2 \; a2 \; a0 \uparrow 0 \; a2 \; S \uparrow S \; a1 \; a0 \uparrow 0 \uparrow 1 \uparrow 2 \uparrow 2 \; a0 \uparrow 0 \uparrow 2$. Table 5.3 shows the array of linearised tree border values for pattern p.

5.4 Forward linearised tree pattern matching algorithm

At this point, it is possible to define Forward linearised tree pattern matching algorithm. The algorithm 12 is just straightforward modification of common pattern matching algorithm with Morris–Pratt style heuristics. The algorithm must additionally also handle wild-card symbol S.

The algorithm uses one pointer into the *pattern* (j) and two pointers into the *subject* (i and *offset*). The pointer i holds position of the current attempt and *offset* holds the position of currently compared symbol. Pointer *offset* is needed due to the "elasticity" of the pattern. While loop at line 3 ensures that the whole subject will be scanned. While loop at line 6 represents a current match attempt. Tests (if and else-if) at lines 7 and 11 perform comparison of a single symbol. There are two variants, one for an alphabet symbol (line 7)

Name: Matches
Input: A string p (pattern) of size m, subtree jump table sjt as a vector of integers and integers *offset* and *stop* (the length of tested match).
Output: Boolean value

```
1  begin
2      i := 1
3      while offset ≤ stop and i ≤ m do
4          if p[i] = p[offset] then
5              i := i + 1
6              offset := offset +1
7          end
8          else if p[i] = S or p[offset] = S then
9              i := sjt[i]
10             offset := sjt[offset]
11
12         end
13         else
14             return false
15         end
16     end
17     return true
18 end
```

Algorithm 10: Linearised tree border — procedure *matches*.

and one for a wild-card symbol S (line 11). In the case of comparison of the wild-card symbol S is necessary to skip the whole subtree in the subject using SJT (line 12) and skip the wild-card symbol S (a pair $S{\uparrow}S$) itself (line 13). Break in else branch (line 16) ends current match attempt in addition to the guard in the respective while. After that (line 19) there are only two options. There was or was not a match. A match is detected and reported (lines 19 and 20). In both cases, the shift is performed according to the border array values.

5.5 Proof

Theorem 5.5.1. Given a tree pattern p in *prefix ranked bar notation* (result of $pref\_ranked\_bar()$) and shift table $ba(p)$ constructed by algorithm 11 and algorithm 12 correctly computes the locations of all occurrences of the pattern p in an input tree t.

Proof. The forward tree pattern matching algorithm is an extension of the Morris–Pratt string pattern matching algorithm. It is to be proved that shift-

Name: Compute linearised tree border array
Input: A string p (pattern) of size m, subtree jump table sjt as a vector of integers.
Output: Table of integers ba

```
1  begin
2  |  ba is a vector of length m initialised to 0
3  |  ba[0] := −1
4  |  for i := 1 to i ≤ m do
5  |  |  min := i
6  |  |  for j := 2 to j ≤ i do
7  |  |  |  if matches(p, sjt, stop = i, offset = j) then
8  |  |  |  |  min = j − 1
9  |  |  |  |  break
10 |  |  |  end
11 |  |  end
12 |  |  ba[i] := i − min
13 |  end
14 end
```

Algorithm 11: Linearised tree border — compute border array.

Table 5.3: Array for linearised tree border from example 5.3.1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	$a2$	$a2$	$a0$	$\uparrow 0$	$a2$	S	$\uparrow S$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$a0$	$\uparrow 0$	$\uparrow 2$
-1	0	1	0	0	1	2	3	3	4	5	6	7	8	9	10	11

ing using $ba(p)$ cannot skip any occurrences of the tree pattern p. Assume that the match attempt was able to match first $j-1$ symbols of the pattern, therefore the shift according to the linearised tree border array is $j - ba[j-1] - 1$. Assume that there is a shorter shift by i symbols where there is another occurrence. Formally for i it holds that $0 < i < j - ba(p)[j-1] - 1$. It must therefore be possible to match a factor of the pattern – $p[i..j]$ with the pattern itself. However according to the definitions 5.3.2 and 5.3.1, the shift for j correctly matched symbols is derived from tests whether the factors of the pattern – $p[k..j]$, where $0 < k < j$, matches the pattern itself. It must also hold that $i \leq k$ since $ba(p)[j-1] \geq 0$. Therefore $p[i..j]$ was already tried whether it matches the pattern itself and since the shift is minimal possible it is clear that the $p[i..j]$ didn't match the pattern itself, hence there are no occurrences on shorter shift. □ □

Name: ForwardLTPM.
Input: The *subject* tree in $pref\_ranked\_bar(subject)$ notation of size n, the tree *pattern* in $pref\_ranked\_bar(pattern)$ notation of size m, $SJT(pref\_ranked\_bar(subject))$, and $BA(pref\_ranked\_bar(pattern))$.
Output: Locations of occurrences of the pattern *pattern* in the tree *subject*.

```
1  begin
2      i := 1
3      while i ≤ n − m + 1 do
4          offset := i
5          j := 1
6          while j ≤ m and offset ≤ n do
7              if pattern[j] = subject[offset] then
8                  j := j + 1
9                  offset := offset +1
10             end
11             else if pattern[j] = S then
12                 offset := sjt[offset]
13                 j := j + 2
14             end
15             else
16                 break
17             end
18         end
19         if j > m then
20             OUTPUT(i)
21         end
22         i := i + j − ba[j − 1] − 1
23     end
24 end
```

Algorithm 12: Forward tree pattern matching algorithm

Table 5.4: Result of preprocessing function (algorithm 11) on the pattern p from example 5.6.1.

0	1	2	3	4	5	6	7	8	9	10	11	12
	a_2	a_0	$\uparrow_0$	a_2	S	$\uparrow_S$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$
-1	0	0	0	1	2	3	3	4	5	6	7	8

Table 5.5: Trace of the run of algorithm 12 for the subject s and the pattern p from example 5.6.1.

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	a_2	a_0	$\uparrow_0$	a_2	a_2	a_0	$\uparrow_0$	a_2	a_0	$\uparrow_0$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$
1	a_2	a_0	$\uparrow_0$	a_2	S											$\uparrow_S$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$
2					a_2	a_0	$\uparrow_0$	a_2	S	$\uparrow_S$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$						
3									a_2													
4										a_2												
5											a_2											

5.6 Example

Example 5.6.1. Consider a pattern p and a subject s with their respective representations in prefix ranked bar notation $pref\_ranked\_bar(p) = a_2\ a_0\ \uparrow_0\ a_2\ S\ \uparrow_S\ a_1\ a_0\ \uparrow_0\ \uparrow_1\ \uparrow_2\ \uparrow_2$ and $pref\_ranked\_bar(s) = a_2\ a_0\ \uparrow_0\ a_2\ a_2\ a_0\ \uparrow_0\ a_2\ a_0\ \uparrow_0\ a_1\ a_0\ \uparrow_0\ \uparrow_1\ \uparrow_2\ \uparrow_2\ a_1\ a_0\ \uparrow_0\ \uparrow_1\ \uparrow_2\ \uparrow_2$. Table 5.4 shows the result of preprocessing — the linearised tree border array. Table 5.5 show the trace of the matching function's run. Explanation:

1. match, $j = 13$, shift by $13 - ba[13-1] - 1 = 4$ to position 5
2. match, $j = 13$, shift by $13 - ba[13-1] - 1 = 4$ to position 9
3. $a_2 \neq a_0$, $j = 1$, shift by $1 - ba[1-1] - 1 = 1$ to position 10
4. $a_2 \neq \uparrow_0$, $j = 1$, shift by $1 - ba[1-1] - 1 = 1$ to position 11
5. $a_2 \neq a_1$, $j = 1$, shift by $1 - ba[1-1] - 1 = 1$ to position 12, but it breaks condition $i \leq n - m + 1$ because $22 - 12 + 1 = 11$ and $12 > 11$

5.7 Time complexity

Consider a pattern of length m and a subject of length n. The time complexity of Forward linearised tree pattern matching algorithm is $O(m^3 + m * n)$ including both preprocessing and matching itself.

Relation *matches* from definition 5.3.1 could be verified in $O(m)$ time where m is the length of the pattern. Computation of linearised tree border takes $O(m^2)$ tests (verifications of relation *matches*). Thus, it is $O(m^3)$ for preprocessing phase.

Matching itself takes $O(m * n)$ time. The linear time complexity is not reached because of the behavior of wild-card symbols S. Anyway, the real case is expected to be better. The number of symbol comparisons is linear in the size of the input tree in the best case.

Chapter 6

Dead-zone

The Dead-zone linearised tree pattern matching algorithm will be introduced in this chapter. This algorithm is the main goal of the thesis.

6.1 Introduction

This objective is actually straightforward. It is just about "gluing" the forward and backward tree pattern matching together.

The Dead-zone algorithm was described in section 2.4. It uses a recursive procedure *dzmat* (algorithm 5). The *dzmat* procedure is able to try a single match attempt and then make a left and right shift according to some heuristics. After the shifts, the area of possible occurrences of the pattern is restricted and the algorithm is a little bit closer to finishing the searching in the whole subject.

The main idea is described in the chapter 4 — Solution introduction. In the case of string pattern matching problem it is possible to use one heuristics in one direction and the "inversed" form of heuristic's function in the opposite direction. This is possible because string has fixed length and translation of dead-zone computed according to the end of the pattern is easily projectable to the beginning of the pattern. In terminology of this thesis: it has two anchors. (Details in section 4.2.)

In the case of the tree pattern matching, the patterns are elastics. The patterns have the beginning fixed but it is not possible to use the dead-zone computed according to the end position of the pattern. Thus, the solution must rely on one anchor. There must be one heuristic in one direction and second heuristic in the opposite direction. To be precise, it is necessary to have one forward and one backward tree pattern matching algorithm. (Details in section 4.3.)

The Backward linearised tree pattern matching algorithm exists and was introduces in [2]. Important details are discussed in the section 3.1.

The Forward linearised tree pattern matching algorithm is important gain of this thesis and Morris–Pratt–like algorithm for linearised trees is introduced in chapter 5.

There is the last step. It is necessary to modify the common form of *dzmat* recursive procedure.

6.2 Theoretical background

This algorithm uses a lot of previously introduced techniques. It must be compatible at lowest level with forward and backward tree pattern matching algorithm. They share the same linearization mechanism and used notation. See definitions 3.1.1 and 3.1.2 in section 3.1.

It needs *Subtree jump table* as well (section 3.1.2).

6.3 Dead-zone linearised tree pattern matching algorithm

The modification of *dzmat* recursive algorithm (algorithm 5) is straightforward.

It is necessary to modify the part representing a single match attempt. This part needs to deal with wild-card symbol S.

The second modification is necessary in the part where the shifts are performed. Shift functions defined by forward and backward tree pattern matching algorithms are used instead of the "anonymous" functions $shift\_left$ and $shift\_right$ for simplicity.

See algorithm 13.

Wild-card symbol S is handled by a while loop starting at line 9. The necessary modification is in the same fashion as in the Forward linearised tree pattern matching algorithm. The algorithm keeps matching until a mismatch is found (handled by else at line 18). It matches an alphabet symbol (if at line 10) or wild-card symbol S (else-if at line 14). In the case of testing of wild-card symbol S it is necessary to skip whole subtree of subject using SJT (line 15). The two pointers into the subject must be used because of the "elastic" behavior as in the case of the Forward linearised tree pattern matching algorithm (see explanation in section 5.4). After the break in else branch at line 19 which ends the current match attempt the prospective match is handled by if at line 22.

The second modification is present at lines 25 and 26. The "abstract" shift functions are replaced by shift functions from appropriate forward and backward algorithms. However with a presence of another shifting heuristics the algorithm can be modified to use them.

The proof of the algorithm correctness is not necessary in this case. Correctness of $shift\_left()$ i.e. shift function of Backward linearised tree pattern

Name: Dead-zone linearised tree pattern algorithm
Input: The *subject* tree in $pref\_ranked\_bar(subject)$ notation of size n, the tree *pattern* in $pref\_ranked\_bar(pattern)$ notation of size m, $SJT(pref\_ranked\_bar(subject))$, $BCS(pref\_ranked\_bar(pattern)$ and $BA(pref\_ranked\_bar(pattern))$.
Output: Locations of occurrences of the pattern *pattern* in the tree *subject*.

```
1  begin
2      Procedure dztpmrec(low, high)
3          if low ≥ high then
4              return
5          end
6          j := ⌊(low + high)/2⌋
7          offset := j
8          i := 1
9          while i ≤ m do
10             if pattern[i] = subject[offset] then
11                 i := i + 1
12                 offset := offset +1
13             end
14             else if pattern[i] = S then
15                 offset = SJT[offset]
16                 i := i + 2
17             end
18             else
19                 break
20             end
21         end
22         if i > m then
23             OUTPUT(j)
24         end
25         new_dead_left := j − BCS[j] + 1
26         new_dead_right := j + i − BA[i − 1] − 1
27         dztpmrec(low, new_dead_left)
28         dztpmrec(new_dead_right, high)
29 end
```

Algorithm 13: Dead-zone linearised tree pattern matching — recursive function.

matching algorithm was proved in [2] and correctness of $shift\_right()$ i.e. shift function of Forward linearised tree pattern matching algorithm was proved in section 5.5.

6.4 Example

Please, read the explanation points in example 6.4.1 carefully. The visualisation shows informations that may not be clear from algorithm itself.

Example 6.4.1. Consider a pattern p and a subject s with their respective representations in prefix ranked bar notation $pref\_ranked\_bar(p) = a_2\ a_0 \uparrow_0 a_2\ S \uparrow_S a_1\ a_0 \uparrow_0 \uparrow_1 \uparrow_2 \uparrow_2$ and $pref\_ranked\_bar(s) = a_2\ a_0 \uparrow_0 a_2\ a_2\ a_0 \uparrow_0 a_2\ a_0 \uparrow_0 a_1\ a_0 \uparrow_0 \uparrow_1 \uparrow_2 \uparrow_2 a_1\ a_0 \uparrow_0 \uparrow_1 \uparrow_2 \uparrow_2$. Both are the same input as in example 5.6.1. Table 6.1 shows the bad character shift table. Table 6.2[1] shows the linearised tree border. Table 6.3 show the trace of the matching function's run. Explanation:

1. Initial dead-zone from $n - m - 1 = 22 - 12 + 2$ to n.
2. First invocation with $low = 1$, $high = 12$, match attempt at position 6, *miss*.
3. BLTPM shift to position 6, FLTPM to position 7, DZ grows by index 6.
4. Invocation with $low = 1$, $high = 6$, match attempt at position 3, *miss*.
5. BLTPM shift to position 2, FLTPM to position 4, DZ grows by index 3.
6. Invocation with $low = 1$, $high = 2$, match attempt at position 1, *match*.
7. BLTPM shift to position -1, FLTPM to position 5, DZ grows by indexes 1–4. Anyway, the invocation with $low = 4$ and $high = 6$ is prepared on stack and it will be executed!
8. Invocation with $low = 4$, $high = 6$, match attempt at position 5, *match*.
9. BLTPM shift to position 3, FLTPM to position 9, DZ grows by indexes 7–8. Anyway, the invocation with $low = 7$ and $high = 12$ is prepared on stack and it will be executed!
10. Invocation with $low = 7$, $high = 12$, match attempt at position 9, *miss*.
11. BLTPM shift to position 9, FLTPM to position 10, DZ grows by index 9.
12. Invocation with $low = 7$, $high = 9$, match attempt at position 8, *miss*. This invocation is redundant.
13. BLTPM shift to position 6, FLTPM to position 11, DZ grows by index 10.
14. Invocation with $low = 10$, $high = 12$, match attempt at position 11, *miss*.

[1] Please note that this table as the same as table 5.4. This copy should improve readability of example.

Table 6.1: Result of preprocessing function (algorithm 6) on the pattern p from example 6.4.1.

a_0	a_1	a_2	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$
1	4	3	2	7	9

Table 6.2: Result of preprocessing function (algorithm 11) on the pattern p from example 6.4.1.

0	1	2	3	4	5	6	7	8	9	10	11	12
	a_2	a_0	$\uparrow_0$	a_2	S	$\uparrow_S$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$
-1	0	0	0	1	2	3	3	4	5	6	7	8

Table 6.3: Trace of the run of algorithm 13 for the subject s and the pattern p from example 6.4.1.

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	a_2	a_0	$\uparrow_0$	a_2	a_2	a_0	$\uparrow_0$	a_2	a_0	$\uparrow_0$	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
2						a_2																
3	a_2	a_0	$\uparrow_0$	a_2	a_2	~~a_0~~	$\uparrow_0$	a_2	a_0	$\uparrow_0$	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
4			a_2																			
5	a_2	~~a_0~~	~~$\uparrow_0$~~	a_2	a_2	~~a_0~~	$\uparrow_0$	a_2	a_0	$\uparrow_0$	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
6	a_2	a_0	$\uparrow_0$	a_2	S											$\uparrow_S$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$
7	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	a_2	~~a_0~~	$\uparrow_0$	a_2	a_0	$\uparrow_0$	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
8					a_2	a_0	$\uparrow_0$	a_2	S	$\uparrow_S$	a_1	a_0	$\uparrow_0$	$\uparrow_1$	$\uparrow_2$	$\uparrow_2$						
9	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	a_0	$\uparrow_0$	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
10									a_2													
11	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_0~~	$\uparrow_0$	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
12								a_2	a_0	$\uparrow_0$												
13	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	a_1	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~
14											a_2											
15	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_2~~	~~a_0~~	~~$\uparrow_0$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~	~~a_1~~	~~a_0~~	~~$\uparrow_0$~~	~~$\uparrow_1$~~	~~$\uparrow_2$~~	~~$\uparrow_2$~~

15. BLTPM shift to position 8, FLTPM to position 12, DZ grows by index 11. Whole text was marked as dead-zone.

6.5 Time complexity

Consider a pattern of length m and a subject of length n. The time complexity of Dead-zone linearised tree pattern matching algorithm is $O(m * |\mathcal{A}| + m^3 + m * n)$ including both preprocessing and matching itself.

The addends $m * |\mathcal{A}|$ and m^3 represent preprocessing phase of Backward and Forward linearised tree pattern matching algorithms, respectively. See sections 3.1.4 and 5.7 for details.

$O(m*n)$ time is the matching phase itself. This phase is not linear because of the behavior of wild-card symbol S. Again, the real case should be better. The number of symbol comparisons is even sub-linear in the size of the input tree in the best case.

Chapter 7

Implementation Details

Let me describe some aspects of implementation and clarify some implementation decisions.

7.1 Implementation language

As implementation language was chosen Java. My professional opinion is that Java is one of the worst possible option. Anyway, the choice has reasonable justification and the justification is the existence of Forest FIRE toolkit.

The Forward linearised tree pattern matching algorithm has a time complexity that is comparable to the Backward linearised tree pattern matching algorithm which behaves well in practise. So the value of time complexity itself is insufficient to get image about algorithm's quality. I need some real data sets and some other algorithms to compare. The Forest FIRE toolkit gives me both.

7.2 Forest FIRE

Forest FIRE toolkit and accompanying FIRE Wood GUI was introduced in [12]. This toolkit already implemented many tree pattern matching algorithms and constructions of automata used in them. E.g. algorithms like DFRTA (deterministic frontier-to-root (bottom-up) tree automaton) or an algorithm based on Aho-Corasick automaton. The original version of toolkit did not contain linearisation of both the pattern tree and the subject tree. The toolkit was extended with linearisation of them for Backward linearised tree pattern matching algorithm [2].

Also, the original implementation of Backward linearised tree pattern matching algorithm was added as well so it is present in Forest FIRE toolkit. My work is based on this extended version.

Forest FIRE contains a rich data set for testing purposes. This data set was obtained by taking the Mono project's X86 instruction set grammar and, for each grammar production, taking the tree in the production's right hand side, and replacing any nonterminal occurrences by wild-card symbol occurrences. The resulting pattern set consists of 460 tree patterns of varying sizes. [2]

Personally I perceive this toolkit imperfect in many ways. It has a too rich object design. Even the most basic operations have resulted in the walk trough deep structure of inherited classes. Adding new algorithm involves into many copy & paste operations during programming. Files defining data sets has unnecessarily complicated structure. There is no direct mechanism for data selection, result evaluation, debugging and so on.

Regardless Forest FIRE toolkit is a good choose for the testing purposes.

7.3 Implementation

Every implemented algorithm is represented by its own class that implements interface *IMatcher*. Instance is constructed with a set of patterns. Interface prescribes method *match* that takes one subject. The method should edit the given tree and mark it with so called annotations.

An annotation comes from Forest FIRE toolkit and it is incompatible with typical approach. There is a "translation" mechanism — method *annotateTree* that takes tree and a set of matches and annotates the tree. This mechanism is very important because it allows to write algorithms in much more readable and portable form.

All auxiliary calculations and structures of algorithm are realised as private members of appropriate class.

No additional requirements are imposed to implementation inside of Forest FIRE toolkit.

7.4 Implemented algorithms

I extended Forest FIRE toolkit by two new algorithms.

Class $KMPMatcher$[2] is implementation of Morris–Pratt–like Forward linearised tree pattern matching algorithm. The algorithm's logic is implemented in private method $matchKMP()$ and it corresponds to algorithm 12.

The necessary auxiliary structure — linearised tree border array — is computed in private method $computeBA()$ (algorithm 11). The method is called in constructor because the object is constructed for some set of pat-

[2] Algorithm itself is variation to Morris–Pratt string pattern matching algorithm. Knuth–Morris–Pratt in the name of class and methods comes from earlier phase of design. This details was unclear at the beginning. It is a little bit misleading. I know it but I hope that it is just trifle.

terns. The method needs algorithm 10 that is implemented by private method *ba_matches*().

Class $DZMatcher$ is an implementation of Dead-zone linearised tree pattern matching algorithm. The algorithm's logic is implemented by private recursive method *dz_rec*() and it corresponds to algorithm 13. First invocation of recursive algorithm is done by private method $matchDZ()$. The necessary auxiliary structures — the linearised tree border array and the bad character shift table — are taken from $KMPMatcher$ and from $BTPMatcher$. Class $BTPMatcher$ is an implementation of the Backward linearised tree pattern matching algorithm and it comes from [2]. The algorithms for auxiliary structures are obtained by copy & paste method from the original classes. [3]

Both matchers use prefix ranked bar notation.

I also designed conceptually same algorithms for prefix ranked notation. The conversion itself is straightforward and it was already present in the toolkit. The Forward linearised tree pattern matching algorithm can process prefix ranked notation with only trivial changes. The Backward linearised tree pattern matching algorithm for prefix ranked notation already exists in the Forest FIRE toolkit. And the Dead-zone linearised tree pattern matching algorithm is again only compilation of the two previously mentioned algorithms.

Classes $PrefixDZMatcher$ and $PrefixKMPMatcher$ represent version of $DZMatcher$ and $KMPMatcher$ for prefix ranked notation. The different classes are necessary due to different alphabet type.

[3] It is not important to achieve a pure object design. Copy & paste of necessary pieces of code is absolutely sufficient.

Chapter 8

Measurements

This chapter contains report about algorithms performance.

8.1 Data

Forest FIRE toolkit contains a pattern set consisting of 460 tree patterns of varying sizes. There are also two subject tree sets. A set of 150 trees of approximately 500 nodes each and a set of 500 trees of approximately 150 nodes each. This data set was obtained by taking the Mono project's X86 instruction set grammar.

The pattern set was previously used for benchmarking Forest FIRE toolkit itself. [2]

Some other details are provided in section 7.2.

8.2 Compared algorithms

I compared my Forward and Dead-zone linearised tree pattern matching algorithms with the Backward linearised tree pattern matching algorithm. It is the closest competitor because it also uses the linearisation mechanism. The algorithms are compared in both available notations: prefix ranked bar notation and an prefix ranked notation.

The algorithms and their variants ($2 * 3 = 6$) are also compared with DFRTA (deterministic frontier-to-root (bottom-up) tree automaton) algorithm and with an algorithm based on Aho-Corasick automaton. These two algorithms are notation independent.

8.3 Setup

All test runs were conducted on a computer with Intel Core i7 at 2.1 GHz (with Turbo Boost technology up to 3.3 GHz) with 4MB cache and 16 GB of

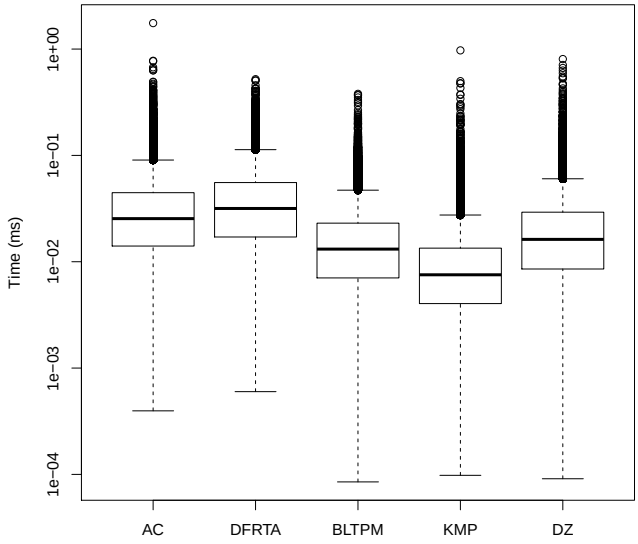


Figure 8.1: Distributions of times for the compared algorithms — prefix ranked bar notation, 500 trees data set.

RAM running Debian GNU/Linux 8 “Jessie” using OpenJDK Java in version 7.

8.4 Results

The results are presented in pictures 8.1, 8.2, 8.3, 8.4 as boxplots on logarithmic scale.

The figures show that Morris–Pratt–like Forward linearised tree pattern matching algorithm outperforms the best algorithms using typical approaches. It even outperforms the Backward linearised tree pattern matching algorithm based on linearisation mechanism.

The Dead-zone linearised tree pattern matching algorithm has slightly worse performance than Backward linearised tree pattern matching algorithm. The algorithm uses the same shift functions. So without any further enhancements the Dead-zone algorithm should be comparable or better than the Morris–Pratt–like Forward linearised tree pattern matching algorithm. But it is not. I found one performance trouble in the current implementation of the dead-zone idea and I think that it is the reason. See Conclusion for details.

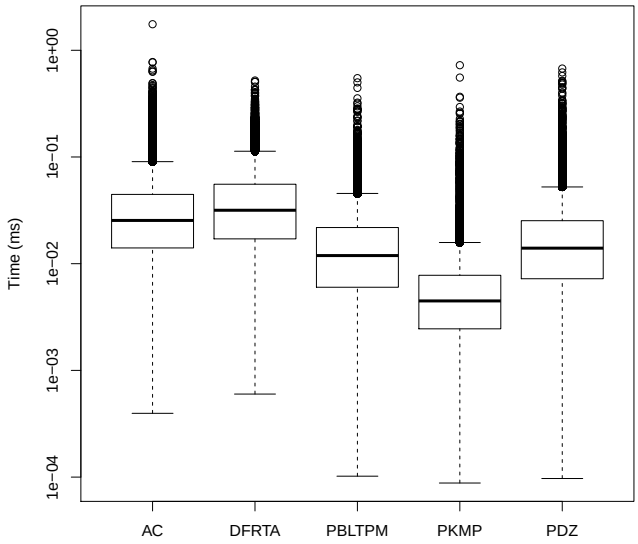


Figure 8.2: Distributions of times for the compared algorithms — prefix notation, 500 trees data set.

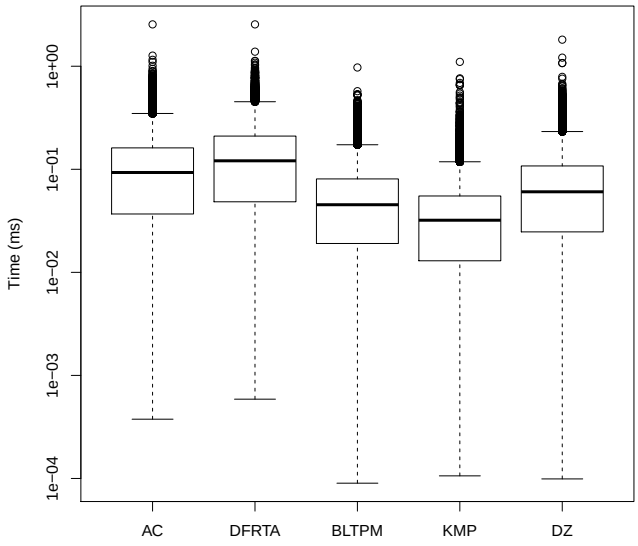


Figure 8.3: Distributions of times for the compared algorithms — prefix ranked bar notation, 150 trees data set.

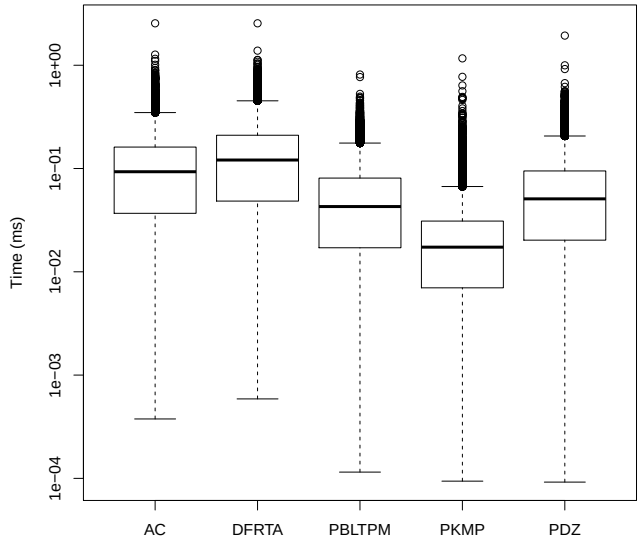


Figure 8.4: Distributions of times for the compared algorithms — prefix notation, 150 trees data set.

Conclusion

At this point is necessary to evaluate the thesis's goals.

The main goal was to adapt the Dead-zone algorithm for pattern matching problem for trees. It was shown in the chapter 4 that it is necessary to use some Forward and Backward tree pattern matching algorithm together. The Backward one exists and was introduced in [2].

Therefore the first step was to design the Forward linearised tree pattern matching algorithm. This algorithm was introduced in the chapter 5. It is a modification of the Morris–Pratt string pattern matching algorithm with shifts based on string borders and border arrays. The analogy of borders for trees was introduced in the same chapter. With borders for linearised trees, the shift function behaves in the same way as in case of string pattern matching. A major success is that this algorithm significantly outperforms the best currently known algorithms.

With the Morris–Pratt–like algorithm it was possible to design the Dead-zone algorithm itself. This algorithm is a modification of the string pattern matching algorithm's version. It was necessary to use the matching loop that is characteristic for the tree pattern matching problem and to use the shift functions from previously mentioned algorithms.

Unfortunately, the linearised tree pattern matching version of Dead-zone algorithm — in current state — is not a significant breakthrough.

I experienced performance issues arising in the original Dead-zone algorithm. The algorithm splits currently explored live-zone to two smaller areas an recursively goes into them. The both recursive calls, to the left and to the right, are prepared within the call stack at the same time. The recursive invocation on the left live-zone could generate shift that will increase dead-zone so much that it affects the right live-zone. The invocation on the original right live-zone actually than may explore in fact a dead-zone because it has out-dated information about dead and live zones. Probability of this happening increases with decreasing distances between consecutive live-zones. It means a lot of unnecessary match attempts. It is a serious impact to the

algorithm's performance.

The main goal of thesis was clearly fulfilled. The Dead-zone algorithm was design. The algorithm works satisfactorily despite the issues in original design. Also another algorithm that improves the current state-of-the-art was presented in this thesis as a step towards the Dead-zone algorithm.

Finally, let me talk about the future work. I have two more ideas how to improve both algorithms.

The Forward linearised tree pattern matching algorithm is inspired by Morris–Pratt algorithm. There is well known improvement of this algorithm on strings and it is what makes the Knuth–Morris–Pratt algorithm. An improvement could be achieved by applying the same idea to the Morris–Pratt–like algorithm.

The second idea concerns to the Dead-zone algorithm. Previously mentioned performance issue has — in my opinion — easy solution. Every invocation could return its knowledge about the right most boundary of a current dead-zone. The information will be aggregated as a maximum and passed to the following recursive invocation. This modification guarantees the elimination of match attempt duplicities.

A similar improvement was presented by the authors of the original Dead-zone algorithm. In their case it lead to a significant speedup and something similar should happen in case of trees. The original algorithm was implemented since it is a algorithm's base version.

Bibliography

[1] Janoušek, J. Arbology [online]. [2016-02-10]. Available from: `http://arbology.fit.cvut.cz/`

[2] Trávníček, J.; Janoušek, J.; Melichar, B.; et al. Linearised Backward Tree Pattern Matching. *LATA 2015*, 2015.

[3] Aho, A. V.; Ullman, J. D. *The theory of parsing, translation, and compiling*. Prentice-Hall, 1972.

[4] Melichar, B.; Janoušek, J.; Flouri, T. Arbology: trees and pushdown automata. *Kybernetika*, volume 48, No.3, 2012: pp. 402–428.

[5] Holub, J. The Prague Stringology Club [online]. [2016-02-19]. Available from: `http://www.stringology.org/`

[6] Bořivoj, M.; Jan, H.; Tomáš, P. Text Searching Algorithms. Volume I: Forward String Matching. 2005, available from `http://stringology.org/athens/TextSearchingAlgorithms/`.

[7] Horspool, R. N. Practical fast searching in strings. *Software Practice and Experience*, volume 10, 1980: pp. 501–506.

[8] Charras, C.; Lecroq, T. EXACT STRING MATCHING ALGORITHMS [online]. [2016-02-28]. Available from: `http://www-igm.univ-mlv.fr/~lecroq/string/node8.html`

[9] D.E., K.; J.H., M. J.; V.R., P. Fast pattern matching in strings. *SIAM Journal on Computing*, volume 6, 1977: pp. 323–350.

[10] Watson, W., B.; Kourie, G., D.; Strauss, T. A Sequential Recursive Implementation of Dead-Zone Single Keyword Pattern Matching. *Combinatorial Algorithms*, volume 23rd International Workshopa, 2012: pp. 236–248.

[11] Searching for Patterns — Set 2 (KMP Algorithm) [online]. [2016-03-17]. Available from: http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

[12] Cleophas, L. Forest FIRE and FIRE Wood: Tools for Tree Automata and Tree Algorithms. 2008: pp. 191–198.

Appendix A

Acronyms

PM Pattern Matching

BCS Bad character shift

SJT Subtree jump table

GUI Graphical user interface

Appendix B

Contents of enclosed CD

- `bare` ..... Git bare repositories of projects
- `src` ..... the directory of source codes
  - `masters_thesis` ..... the directory of LaTeX source codes of the thesis
  - `forestfirewood` ..... implementation sources
- `DP_Oburka_Robin.pdf` ..... the thesis text in PDF format
- `DP_Oburka_Robin.ps` ..... the thesis text in PS format