CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Implementation of a repetition searching algorithm in trees
Student:	Aleksandr Shatrovskii
Supervisor:	Ing. Jan Trávníček
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2017/18

Instructions

Analyze the algorithm of computing all subtree repeats in trees [1].
Analyze the internal representation of trees in Automata library [2].
Implement the algorithm of computing all subtree repeats in Automatalibrary.
Test the implementation using appropriate randomly generated andpredefined trees.

References

[1] Christou, M., Crochemore, M., Flouri, T., Iliopoulos, C.S., Janoušek, J., Melichar, B., Pissis, S.P.: Computing all subtree repeats in ordered trees. Information Processing Letters 112(24), 958–962 (2012)
[2] Plachy Stepan: Automatová knihovna - Stromové automaty a algoritmy nad stromy. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2015.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague December 14, 2016

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

Bachelor's thesis

Implementation of a Repetition Searching Algorithm in Trees

Aleksandr Shatrovskii

Supervisor: Ing. Jan Trávníček

15th May 2017

Acknowledgements

I would like to thank my supervisor, Ing. Jan Trávníček, for his guidance and my dear parents for their support and encouragement.

Declaration

In Prague on 15th May 2017 . . . .

Citation of this thesis

Shatrovskii, Aleksandr. *Implementation of a Repetition Searching Algorithm in Trees*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Předmětem této práce je implementace algoritmu hledání repetic v ohodnocených označených uspořádaných stromech. V práci je podrobně analyzován a popsán efektivní algoritmus, předloženy Michalisem Christou a dalšími, je prozkoumaná vnitřní reprezentace stromových struktur v Automatové knihovně. Algoritmus a další potřebné podpůrné struktury jsou začleněny do Automatové knihovny. Implementace je otestována pomocí předdefinovaných a náhodně generovaných stromů.

Klíčová slova ohodnocený označený uspořádaný strom, repetice ve stromu, knihovna, arbologie

Abstract

This thesis is concerned with implementation of the algorithm which computes all subtree repeats in a ranked labeled ordered tree. The efficient algorithm proposed by Michalis Christou et al. is analyzed in detail and is presented in this thesis. Internal representation of tree structures in the Automata library is explored. The algorithm and necessary data structures are implemented as part of the Automata library project. The implementation is tested with both predefined and randomly generated trees.

Keywords ranked labeled ordered tree, subtree repeats, library, arbology

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Tree is a common data structure in Computer science. It has a wide range of applications from data organization to computational biology. Arbology is an algorithmic discipline that focuses on algorithms on trees [1]. Detecting subtree repeats in trees is one of the problems this discipline is concerned with. Systems that heavily rely on trees, such as compilers and the process of code optimization, can benefit from new findings in this field of study.

My thesis focuses on one of such algorithms. It is used to find all subtree repeats in ranked labeled ordered trees in linear time and space and was proposed by M. Christou et al. [2].

Goals

The main goal of this thesis is to implement the aforementioned efficient algorithm as part of the Automata library. To accomplish it I will have to analyze the algorithm, analyze the existing implementation of tree structures in the Automata library and test the implementation afterwards, using both predefined and randomly generated trees.

Thesis structure

Chapter 1 covers theory, basic definitions and properties of trees that are required for understanding of the inner workings of the algorithm. The algorithm is then presented and discussed in Chapter 2. In the following Chapter 3 I describe the Automata library project and analyze the existing implementation of tree structures in the library. Chapter 4 focuses on my own implementation of the new algorithm, design choices, additional structures and modifications that I had to make in order to accommodate the new functionality. Finally, in Chapter 5 I describe how the implementation has been tested. Conclusion is the closing chapter of this work.

Chapter 1

Theory

1.1 Strings

Definition 1.1.1. *Alphabet* (Σ) is a finite and non-empty set of elements.

Definition 1.1.2. The elements of the alphabet are called *symbols*.

In this thesis curly braces will be used to denote sets and round braces will be used to denote collections where order is significant.

Example 1.1.3. Set {a} is a minimal alphabet made up of a single element *a*.

Definition 1.1.4. *Concatenation* of two given symbols x and y is denoted by the two symbols joined: xy.

Definition 1.1.5. *String* x is a possibly empty, finite sequence of symbols $(x_1, x_2, \ldots, x_n)$, where $n \geq 1$ and $x_i \in \Sigma, 1 \leq i \leq n$.

Example 1.1.6. A letter “č”, a word “computer” or a fragment of programming code are all considered to be strings according to Definition 1.1.5.

Definition 1.1.7. *Length* is a property of a string, that corresponds to the number of elements that make up the string. For a given string x it is denoted by $|x|$.

Example 1.1.8. Let x be a string with the value *tree*, where each character is a symbol. The length of $x = |x| = |(t, e, x, t)|$ equals to 4.

Definition 1.1.9. *Empty string* is a zero-length string. In writing it is denoted by the symbol ε.

Definition 1.1.10. *Concatenation* of two given strings $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ results in a string $xy = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$.

Definition 1.1.11. *Factor* (or a *substring*) w of a string x is a range $x[i..j]$, $1 \leq i \leq j \leq |x|$ of symbols in a string x, that will form the initial string x when concatenated with two other possibly empty strings $y = (x_1, \ldots, x_{i-1})$ and $z = (x_{j+1}, \ldots, x_{|x|})$.

Example 1.1.12. The word *knowledge* contains two words in itself: *know* and *ledge*. Both of them are factors of the original string.

Definition 1.1.13. *Ranked alphabet* $\mathcal{A}$ is a couple (Σ, φ), where Σ is the alphabet as introduced in Definition 1.1.1 and φ is a mapping from a finite set of symbols to a set of natural numbers (including zero) $\varphi : \Sigma \to \mathbb{N}$ [3].

Definition 1.1.14. *Arity or rank* of a symbol is denoted as $\varphi(a)$, where $a \in \Sigma$. It reflects the number of children of a node that has a as its symbol.

Example 1.1.15. The mapping $\varphi : \Sigma \to \mathbb{N}$ means that every symbol of the alphabet gets an integer value assigned to it. If some hypothetical alphabet comprises the names of functions that exist in the scope of a given program file, then the rank of each symbol of such alphabet could represent the number of parameters that can be passed to the function: $\{(multiply\_two\_numbers, 2)$, $(fibonacci, 1)\}$.

Definition 1.1.16. Each couple of the ranked alphabet can be *enumerated* with a mapping $\mu : \Sigma \times \mathbb{N} \to \mathbb{N}$.

Example 1.1.17. Mapping μ essentially assigns a unique number to every couple $(\alpha, \varphi(\alpha))$ of the ranked alphabet $\mathcal{A}$. Possible enumeration of a ranked alphabet consisting of selected logical operators may be represented as follows:

- $(not, 1) \longmapsto 1$
- $(or, 2) \longmapsto 2$
- $(and, 2) \longmapsto 3$

1.2 Trees

Definition 1.2.1. *In-degree* of a node is the number of edges that are directed to the given node. Accordingly, *out-degree* of a node is the number of edges directed away from the given node.

Definition 1.2.2. *Rooted ordered tree* is a data structure made up of nodes connected by directed edges that does not contain cycles, where *root node* (exactly one) is the node which has in-degree 0 while all other nodes have in-degree 1 and *leaf node* is the node (one or more) that has out-degree 0.

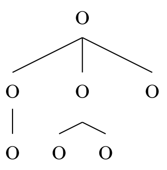

Figure 1.1: Rooted tree

Example 1.2.3. As can be seen in Figure 1.1, a rooted ordered tree has exactly one node with in-degree of 0 — that is the root node. Ordering means, that the order of subtrees of each node is significant and predefined. Changing the order of children of some node will produce a different tree.

Note, that in this and all subsequent tree representations in this thesis directed edges are represented by lines, where the node below the line is the destination node.

Definition 1.2.4. *Labeled tree* is a tree in which every node has a symbol from an alphabet assigned to it.

Definition 1.2.5. *Rank* is the characteristic of a tree that signifies that for every node out-degree, that reflects the arity of the symbol associated with the node, is given.

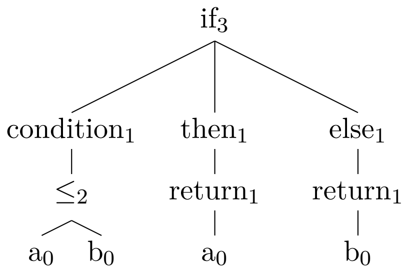

Figure 1.2: Rooted ordered ranked labeled tree

Example 1.2.6. Figure 1.2 serves as illustration of a rooted ordered ranked labeled tree. It is a pseudo-representation of the abstract syntax tree that holds "*if*" statement that returns the smaller of two values. Rank of each node is written after the respective symbol in subscript.

Definition 1.2.7. *The number of nodes* in a tree t is denoted by $|t|$.

Example 1.2.8. The number of nodes in a tree in Figure 1.2 is 11.

Definition 1.2.9. *Height of a tree* t is the maximal length of a path from its root to a leaf expressed in the number of edges.

Example 1.2.10. The height of a tree in Figure 1.1 is 2, while the height of a tree in Figure 1.2 is 3.

1.3 Postfix notation of a tree

Definition 1.3.1. *Postfix notation* of a tree t is a linear representation *post(t)* of a tree t that can be obtained by applying the recursive Algorithm 1.3.1 to the root node.

Algorithm 1.3.1 Tree-To-Postfix

```
Input: Triplet (S, ℓ, ac)
1: function POSTFIX(node)
2:     for each child in node.children do          ▷ In order of appearance
3:         POSTFIX(child)
4:     OUTPUT(node)                                ▷ printf(), for example
```

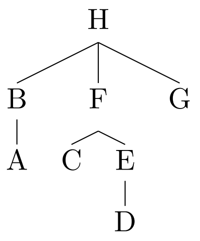

Figure 1.3: Tree for the illustration of a postfix notation

Example 1.3.2. Nodes of the tree in Figure 1.3 will be listed in the alphabetical order after the application of the Algorithm 1.3.1 to the root node *H*.

Definition 1.3.3. Trees are *equal* if and only if their postfix notations form the same strings and these strings are constructed from the same ranked alphabet.

Definition 1.3.4. Let t be a tree, let x be its representation in postfix notation *post(t)*, let p be a subtree of the tree t, let w be the representation of p in postfix notation *post(p)*, then *a subtree repeat* p in a tree t is a tuple $M_{x,w} = (\{i_1, i_2, \dots, i_r\}; |p|)$, where $r \geq 2$, $i_1 < i_2 < \dots < i_r$, and $w = x[i_1..i_1+|p|-1] = x[i_2..i_2+|p|-1] = \dots = x[i_r..i_r+|p|-1] = post(p)$ [2].

Definition 1.3.5. *Complete* subtree repeat $M_{x,w}$ is a tuple $M_{x,w}$ that includes every occurrence of w in x.

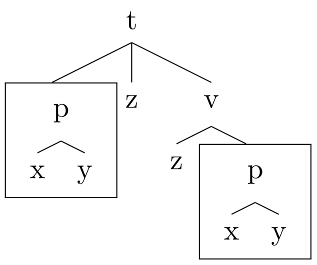

Figure 1.4: Tree with highlighted subtree repeats

Example 1.3.6. It is possible to demonstrate Definition 1.3.4 on the tree in Figure 1.4. In this case, $post(t) = (x, y, p, z, z, x, y, p, v, t)$ and $post(p) = (x, y, p)$, tuple of subtree repeat is $M_{x,w} = (\{1, 6\}; 3)$. In this case, it is also complete.

1.4 Properties of trees in postfix notation

Tree structures when represented by their postfix notation have some unique properties that the algorithm for computing all subtree repeats is based upon.

Lemma 1.4.1. The postfix notations of all subtrees of a tree t over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$ are factors of the postfix notation $post(t)$ of t.

Example 1.4.2. Subtree repeats have been demonstrated in the previous Example 1.3.6. Notice, that the postfix notation of a subtree p, $post(p) = (x, y, p)$ appears twice in the postfix notation of the original tree and that it is always a substring of the $post(t) = (\mathbf{x}, \mathbf{y}, \mathbf{p}, z, z, \mathbf{x}, \mathbf{y}, \mathbf{p}, v, t)$.

Definition 1.4.3. Let $w[1..m], m \geq 1$ be a string over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$. Then, the *arity checksum* $ac(w) = \varphi(a_1) + \varphi(a_2) + \ldots + \varphi(a_m) - m + 1 = \sum_{i=1}^{m} \varphi(a_i) - m + 1$.

Example 1.4.4. In other words, to compute this property, one has to find the sum of out-degrees of all the nodes involved, subtract the number of those nodes and add 1. For the tree t in Figure 1.4, $x = post(t)$, $ac(x) = 0$.

Every valid tree structure will always produce zero checksum. The arity checksum of a graph that contains disconnected valid tree structures will be negative. The arity checksum of a graph that does not contain some subtree of otherwise valid tree structure will be positive.

Lemma 1.4.5. Let $post(t)$ be a tree t in postfix notation and let w be a factor of $post(t)$ over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$. Then w is the postfix notation of a subtree of t if and only if all of the following conditions are met:

- $ac(w) = 0$
- $\varphi(w[1]) = 0$

- no subtree rooted at $w[l]$, where $1 \leq l \leq |w|$, has leftmost node that appears before $w[1]$ in $post(t)$

Example 1.4.6. I will refer to the tree in Figure 1.4 again. The arity checksum of w is $(2 + 0 + 0) - 3 + 1 = 0$, which means that the first condition is met for t. The second condition is also not violated: $w[0] = (x)$, x is a leaf of p and t in both cases. The tree conforms to the third condition as well, because postfix representations of all three simple subtrees are contained within w and are factors of it.

Chapter 2

Algorithm

The algorithm [2][4] that is analyzed and implemented in this thesis consists of two phases. In the first one all the necessary auxiliary arrays are computed. It is known as the *preprocessing phase.* Results obtained in this phase will then be used throughout the second *computation phase*, where the subtree repeats will be calculated.

The problem that it aims to solve is "computing all complete subtree repeats of a rooted ordered labeled ranked tree t consisting of n nodes" [2].

For demonstration purposes in this chapter let $\mathcal{A} = (\Sigma, \varphi(\Sigma))$ be the ranked alphabet, where $\Sigma = \{a, b, c, d, e, m\}$ and $\varphi(\Sigma) = \{0, 0, 2, 2, 2, 2\}$. Let t be a tree, where each node is labeled by a symbol from $\mathcal{A}$. Figure 2.1 depicts such a tree. Arity of each node is written in subscript.

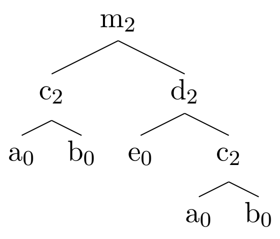

Figure 2.1: Demonstration tree

In this chapter for illustrative purposes contents of queues will be written inside parentheses with the element in front of the queue (first to be popped) being the leftmost.

2.1 Approach

The algorithm utilizes a bottom-up approach. It traverses the tree from its leaves to the root node. This traversal resembles postfix representation $x = post(t)$ of a tree t, which is why postfix representation is used.

The foundation of all subtree repeats that may be discovered in a tree are leaves. Complying with the bottom-up approach, leaves are the first nodes that need to be grouped into sets by the labels assigned to them according to mapping μ. The number of sets by labels that will be computed at this stage is the maximum number of tree repeats that will be found at any other level of the tree, because all other subtrees will be defined in the algorithm by their leftmost node and the leftmost node of any subtree is a leaf (Lemma 1.4.5).

This algorithm runs in linear time and space — it takes the number of iterations equal to the height of a tree to finish, because at each step i all subtree repeats of height $i-1$ will have already been computed.

It follows the principles of dynamic programming by breaking the problem down into smaller subproblems, solving them and then reusing the results. The smallest possible subproblem in this case is a leaf node.

2.2 Preprocessing phase

To improve efficiency of calculation, an input tree has to be preprocessed. As a result, the following auxiliary arrays will be produced:

- Integer parent array P will store index of the parent node for each node of t except for the root node, as it has an in-degree of 0 (Definition 1.2.2).
- Integer height array H will store the height of each subtree of t. For every subtree p there will be an integer value stored in $H[i]$, if the root node of p appears at $post(t)[i]$. As the first node that appears in the postfix representation of a tree is always a leaf, array H will always have 0 at the first index ($H[0] = 0$), as well as at every other index of $post(t)$ that corresponds to a position of a leaf.
- Binary array FC will indicate whether a node is the leftmost child of its parent with a positive value (1 or "true") at a respective index in the array or a negative (0 or "false") if the condition does not hold. It does not include the root node since it does not have a parent node.

These arrays can be computed in linear time and space. Algorithms 2.2.1, 2.2.2 and 2.2.3 produce arrays P, H and FC respectively [5, p. 47-48, 127].

Table 2.1 is an example of the contents of the arrays after they have been constructed for the tree t in Figure 2.1. Note that numbering starts from 0 in this example, just as it will in real implementation.

All the other arrays and variables are initialized to zero.

2.3 Additional data structures

Apart from the variables described in the previous section, there are other temporary data structures which the algorithm uses in calculation. Their types

Algorithm 2.2.1 Node-Parents-Array

Input: $post(t) = x[1..n]$
Output: Array P where element i denotes the parent of $x[i]$

```
1:  $R \leftarrow$ New-Stack
2:  for $i \leftarrow 1$ to $n$ do
3:      for $j \leftarrow 1$ to $\varphi(x[i])$ do
4:          $r \leftarrow$ Pop($R$)
5:          $P[r] \leftarrow i$
6:      Push($R, i$)
```

Algorithm 2.2.2 Subtree-Height-Array

Input: $post(t) = x[1..n]$
Output: Array H where element i is the height of subtree rooted at $x[i]$

```
1:  $R \leftarrow$ New-Stack
2:  for $i \leftarrow 1$ to $n$ do
3:      if $\varphi[x_i] = 0$ then
4:          Push($R, 0$)
5:          $H[i] \leftarrow 0$
6:      else
7:          $r \leftarrow 0$
8:          for $j \leftarrow 1$ to $\varphi(x[i])$ do
9:              $r \leftarrow \max(r,$Pop(R)$)$
10:         $H[i] \leftarrow r + 1$
11:         Push($R, r + 1$)
```

Algorithm 2.2.3 First-Child-Array

Input: $post(t) = x[1..n]$
Output: Binary array F such that $F[i] = 1$ if $x[i]$ is a first child

```
1:  $R \leftarrow$ New-Stack
2:  for $i \leftarrow 1$ to $n$ do
3:      if $\varphi[x_i] = 0$ then
4:          Push($R, i$)
5:      else
6:          for $j \leftarrow 1$ to $\varphi(x[i]) - 1$ do
7:              $r \leftarrow$ Pop($R$)
8:              $F[r] \leftarrow 0$
9:          $r \leftarrow$ Pop($R$)
10:         $F[r] \leftarrow 1$
11:         Push($R, i$)
```

$Index$	0	1	2	3	4	5	6	7	8
x	a	b	c	e	a	b	c	d	m
$\varphi(x)$	0	0	2	0	0	0	2	2	2
$\mu(x, \varphi(x))$	0	1	2	3	0	1	2	4	5
P	2	2	8	7	6	6	7	8	-
H	0	0	1	0	0	0	1	2	3
FC	1	0	1	1	1	0	0	0	-

Table 2.1: Auxiliary arrays for the tree in Figure 2.1

vary from simple integer variables, arrays and queues to arrays of queues. They are presented below:

- Array μ stores a unique number from a sequence ranging from 1 to $n = |t|$ for each couple $(\Sigma, \varphi(\Sigma))$ of the alphabet thus making identification of each symbol and subsequently the subtree it defines easier, because the type of the alphabet elements must not necessarily be enumerable. What's more, different nodes in a tree with different number of children can share the same label. Mapping helps to solve this issue by dividing this nodes into separate sets.

- Integer variable sc is used to identify and also count every subtree repeat that has been found so far. By the end of execution of the algorithm it will be set to the number of total subtree repeat tuples that have been discovered in a particular tree.

- Array T stores identifier sc of a found subtree repeat at index i, where i is the first position in the postfix representation of the corresponding subtree.

- Array TL is used in conjunction with the array T. If the subtree identifier is saved in T at index i, the length of the corresponding subtree is stored at $TL[i]$.

- Triplets (S, ℓ, ac) are used to denote all the occurrences of some factor w of $x = post(t)$. S is a set of starting positions of w inside x. ℓ is the length of w and ac is the arity checksum that conforms to Definition 1.4.3. Every subtree can be defined with a triplet.

- Level array LA is an array of queues of triplets. Triplets will be placed into queue i of LA inside of function Assign-Level if they define a subtree that is a child of a node of height i or, alternatively, if the height of their tallest sibling is $i - 1$. This behavior makes LA act as a temporary storage for results of calculation of smaller subproblems.

Example 2.3.1. Let t be the tree in Figure 2.2. Its postfix representation is (b, c, a). Subtrees that will be found during the run of the algorithm are (b), (c) and (b, c, a) with the identifier sc incremented and set for each subtree repeat in order of their appearance. Contents of the arrays T and TL for tree t are presented in Table 2.2.

Figure 2.2: Small tree with three subtree repeats

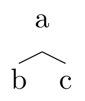

Table 2.2: T and TL for Tree 2.2

	Index	0	1	2
$h = 0$	T	1	2	0
	TL	1	1	0
$h = 1$	T	3	2	0
	TL	3	1	0

Example 2.3.2. The subtree rooted at node b in Figure 2.2 can be described with the triplet $(\{0\}, 1, 0)$. The original tree itself rooted at c can be described with the triplet $(\{0\}, 3, 0)$. Note, that the only thing that differs between them is the length of w. This is the reason why $T[0]$ has been overwritten in Table 2.2.

Example 2.3.2 illustrates how just by changing the length parameter a taller subtree can be obtained, but for it to be valid, it must adhere to requirements of Lemma 1.4.5.

2.4 Computation phase

The computation phase of the proposed algorithm consists of three functions: Subtree-Repeats, Assign-Level and Partition. Each function is listed in Algorithms 2.6.2, 2.6.1 and 2.6.3 respectively in a dedicated Section 2.6.

The starting point of execution is function Subtree-Repeats. It gets the postfix representation of a tree over a ranked alphabet as input x.

Its first part between lines 2 and 14 is devoted to allocating the auxiliary arrays described in Section 2.2 and initialization of other simple data structures that will be used throughout this function. Variables that have not been previously mentioned are initialized with zero values.

Line 16 is where the process of grouping leaves into sets by their labels starts. The *for* loop traverses through each node of a tree in its postfix representation. All the inner nodes are skipped, because they can not be used in a triplet (S, ℓ, ac) to describe a subtree as that would violate the requirements of Lemma 1.4.5 (nodes inside S must be leaves). The indices inside $post(t)$ that correspond to positions of leaves with the same identifier are placed into a set $A_{\Sigma}[k]$ where k is an integer value assigned to a symbol by mapping μ.

Apart from the array of queues A_{Σ}, other structures involved in the computation are as follows:

- Variable k is used to store the integer assigned to the node being processed by mapping μ.
- Queue Q_5 holds the types of nodes (μ) that have been encountered among leaves in order of their appearance.
- Boolean array B_{Σ} tracks the type of nodes that have been added to Q_5 to avoid duplicates.
- Integer array C_{Σ} stores a sequentially incremented number for each node type in Q_5 that will later be used to identify a subtree repeat starting at this node.

Example 2.4.1. Contents of the array A_{Σ} for Tree 2.1 after the loop ends are presented in Table 2.3. The contents of queue Q_5 are $(0, 1, 3)$. Ascending order here is purely coincidental because the leaves were processed in the same order in which mapping μ was assigned to their labels.

Table 2.3: Contents of array A_{Σ} at line 31

	i	0	1	2	3	4	5	6	7	8
$A_{\Sigma}[i]$	Queue front	0	1		3					
	Queue back	4	5							

At this point all leaves have been processed and placed into a set according to their labels. Complete subtree repeats of height 0 have been found and ready to be output or saved for later processing depending on the implementation. This is done inside a *while* loop that starts at line 31 by popping every element of Q_5 and using it as an index in A_{Σ} to retrieve appropriate subtree repeats indices. In the same loop, these sets are being encapsulated in set S which is a part of triplet (S, ℓ, ac). ℓ is set to one (size of a one-node tree) and the arity checksum is set to 0, because every leaf is a valid subtree of a tree, therefore it can be repeated.

The triplet is then sent to function Assign-Level that will check if the root of a subtree, represented by each element of S together with length ℓ, is the first child of its parent. If that is true, then the subtree can be joined with its siblings and a parent node to form a taller subtree (Section 2.1). Formally, this process can be summed up with an expression

$A_n[k] = \{i | i \in S \wedge FC[i + \ell - 1] = 1 \wedge H[P[i + \ell - 1]] = k\}$. Afterwards, triplets from A_n get pushed into a queue h of the level array LA where h is the height of the parent node of a given subtree.

Other structures involved are described below:

- *root* is the position of the root node of a subtree in x that is calculated from the factor length $|w| = \ell$ and position of $w[0]$ inside x.
- Queue Q_4 is used to track the heights of the parent nodes of the subtrees.
- Array B_n aides avoidance of duplicates in Q_4.

Example 2.4.2. The contents of the level array LA after the function call ends are presented in Table 2.4.

Table 2.4: Contents of array LA after the first call to ASSIGN-LEVEL

i	0	1	2	3
LA		$((0,4),1,0)$	$((3),1,0)$	

Indices 1 and 5 have not been included in any set because they can not define a subtree with their height being greater than 0. One of the triplets have been assigned to the second queue in LA, because it has a sibling of height $2 - 1 = 1$.

After the level assignment has finished, SUBTREE-REPEATS can proceed. At line 36 the *for* loop starts. It iteratively progresses through array LA, retrieves a triplet assigned to the height being processed at a given cycle and sends it to function PARTITION as an argument.

The purpose of the recursive function PARTITION is to *partition* set S into smaller sets and at the same time expand the structure defined by the triplet that it gets as an argument until that triplet represents a valid tree.

Inside the loop at line 2 the algorithm iteratively dequeues a node from set S and calculates the index of the next node r that will immediately follow this subtree in the postfix notation x. Condition inside the *if* statement at line 5 checks if any other subtree has been previously discovered at $x[r]$. It would mean that node r has already been used inside some set S to define some subtree of t. If that is true, new triplet needs to be formed to define those two neighboring subtrees. In this case, ℓ will predictably be increased by the size of the subtree at $x[r]$ (stored in $TL[r]$) and the arity checksum will be decreased by 1. To make ac 0 again and consequently validate the tree structure a node needs to be discovered later that will act as a parent node for the subtrees $x[i..r-1]$ and $x[r..r + TL[r] - 1]$.

If the subtree at $x[r]$ has not been processed yet, that means that node r is a root node of subtree $x[i..i + l]$. The newly constructed triplet will have its length ℓ increased by 1 and ac set to 0, which means that the structure described by it has become a valid subtree again.

The pseudocode between lines 20 and 29 allows for handling of two aforementioned cases. It moves the triplets from queues Q_1 and Q_2 into a universal triplet queue Q_3.

Inside the final *while* loop at line 30 arity checksum of each triplet stored in Q_3 is checked. If $ac = 0$, which means that no more processing of this triplet is required, the algorithm outputs or saves the subtree repeat, updates values of $T[i]$ and $TL[i]$ for each i inside set S with an incremented value of the subtree repeat counter and sends the triplet to function Assign-Level to check, if it can be used again. If $ac \neq 0$ the triplet will be sent to function Partition for further expansion.

The rest of the variables used in function Partition are described below:

- Triplet arrays E_n and E_Σ are used to store triplets inside Partition. They differ in size, because the former is used when the neighboring node has been processed and has got some value of the subtree counter sc ussigned to it, unlike the latter that has to be indexable by the range of values that μ can take.
- Arrays B_n and B_Σ compliment arrays E_n and E_Σ by aiding duplicate avoidance.

Table 2.5 illustrates how the contents of arrays T and TL change throughout the execution of the algorithm. Table 2.6 serves the same purpose for array LA. Zero values are omitted to assist readability.

Table 2.5: Changes in arrays T and TL during the algorithm execution for the tree in Figure 2.1

	i	0	1	2	3	4	5	6	7	8
$h = 0$	T	**1**	**2**		**3**	**1**	**2**			
	TL	**1**	**1**		**1**	**1**	**1**			
$h = 1$	T	**4**	2		3	**4**	2			
	TL	**3**	1		1	**3**	1			
$h = 2$	T	4	2		**5**	4	2			
	TL	3	1		**5**	3	1			
$h = 3$	T	**6**	2		5	4	2			
	TL	**9**	1		5	3	1			

Finally, the algorithm has reached its end. The output format has not been explicitly specified, but one of the many possible ways to present the results of computation is presented in Figure 2.3. Each subtree repeat is output on a separate line with node IDs in curly brackets and the length of the subtree repeat appearing after the comma.

Table 2.6: Changes in array LA during the algorithm execution for the tree in Figure 2.1

	i	0	1	2	3
$h = 0$	LA		$((0,4),1,0)$	$((3),1,0)$	
$h = 1$	LA			$((3),1,0)$	$((0),3,0)$
$h = 2$	LA				$((0),3,0)$
$h = 3$	LA				

```
#1 {0,4}, 1
#2 {1,5}, 1
#3 {3}, 1
#4 {0,4}, 3
#5 {3}, 5
#6 {0}, 9
```

Figure 2.3: Possible output of the algorithm

2.5 Observations

During the analysis of the article [2] and the algorithm proposed in it, I have discovered an inconsistency between the definition of a subtree repeat 1.3.4 and the pseudocode of the algorithm, that solves the problem of finding them. Unlike the definition that appears in the same article and requires at least two strings corresponding to a subtree to be found at different positions in $post(t)$, the algorithm will output the starting position and the length of some factor of $post(t)$ even if the subtree it represents is not repeated anywhere in the same tree.

I have also discovered a flaw in Function 2.6.3. The call to Function 2.6.1 at line 38 is inside the *for* loop that iterates through the contents of set S, but set S itself does not change between the calls, yet the function gets called with the same argument S multiple times. This causes array LA to get filled with duplicates. The problem can be solved by moving the function call out of the loop. A simple typographic mistake is the apparent cause of this issue. My suspicion was confirmed by another earlier article [4] describing the same algorithm that does not have this flaw in it. I have corrected this mistake in the pseudocode in Section 2.6.

2.6 Pseudocode

The algorithm proposed by M. Christou et al. [2] is split into three functions. Pseudocode for each one of them is presented in this section.

Algorithm 2.6.1 Assign-Level

Input: Triplet (S, ℓ, ac)

```
 1: $Q_4 \leftarrow$ New-Queue
 2: while not empty S do
 3:     $i \leftarrow$ Dequeue($S$)
 4:     $root \leftarrow i + \ell - 1$
 5:     if $FC[root] = 1$ then
 6:         $k \leftarrow H[P[root]]$
 7:         Enqueue($A_n[k], i$)
 8:         if $B_n[k] = 0$ then
 9:             $B_n[k] \leftarrow 1$
10:             Enqueue($Q_4, k$)
11: while not empty $Q_4$ do
12:     $k \leftarrow$ Dequeue($Q_4$)
13:     Enqueue($LA[k], (A_n[k], \ell, 0)$)
14:     $B_n[k] \leftarrow 0$
15:     $A_n[k] \leftarrow$ Clear-List
```

Algorithm 2.6.2 SUBTREE-REPEATS

Input: $x[1..n] = post(t)$ over ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$

Output: Sets of starting positions of factors of $post(t)$ and their lengths, representing subtrees from t

```
▷ Initialize global variables
$sc \leftarrow 0$
$A_\Sigma[1..|\Sigma|] \leftarrow$ NEW-QUEUE-ARRAY
$A_n[1..n] \leftarrow$ NEW-QUEUE-ARRAY
$B_\Sigma[1..|\Sigma|] \leftarrow$ NEW-BIT-ARRAY
$B_n[1..n] \leftarrow$ NEW-BIT-ARRAY
$C_\Sigma[1..|\Sigma|] \leftarrow$ NEW-INTEGER-ARRAY
$E_\Sigma[1..|\Sigma|] \leftarrow$ NEW-TRIPLET-ARRAY
$E_n[1..n] \leftarrow$ NEW-TRIPLET-ARRAY
$Q_5 \leftarrow$ NEW-QUEUE
$LA[1..h(t)] \leftarrow$ NEW-QUEUE-ARRAY
$FC[1..n] \leftarrow$ COMPUTE-FIRST-CHILD-ARRAY
$H[1..n] \leftarrow$ COMPUTE-NODE-HEIGHT-ARRAY
$P[1..n] \leftarrow$ COMPUTE-NODE-PARENTS-ARRAY
▷ Start of the algorithm
for $i \leftarrow 1$ to $n$ do
    if $\varphi(x[i]) = 0$ then
        $k \leftarrow \mu(x[i], \varphi(x[i]))$
        if $B_\Sigma[k] = 0$ then
            $B_\Sigma[k] \leftarrow 1$
            ENQUEUE($Q_5, k$)
        ENQUEUE($A_\Sigma[k], i$)
        if $C_\Sigma[k] = 0$ then
            $sc \leftarrow sc + 1$
            $C_\Sigma[k] \leftarrow sc$
        $T[i] \leftarrow C_\Sigma[k]$
        $TL[i] \leftarrow 1$
    else
        $T[i] \leftarrow 0$
        $TL[i] \leftarrow 0$
while not empty $Q_5$ do
    $k \leftarrow$ DEQUEUE($Q_5$)
    $B_\Sigma[k] \leftarrow 0$
    OUTPUT($A_\Sigma[k], 1$)
    ASSIGN-LEVEL(($A_\Sigma[k], 1, 0$))
for $i \leftarrow 1$ to $H[n]$ do
    while not empty $LA[i]$ do
        PARTITION(DEQUEUE($LA[i]$), $x$)
```

Algorithm 2.6.3 Partition

Input: Triplet (S, ℓ, ac), $x[1..n] = post(t)$

```
1:  $Q_1, Q_2, Q_3 \leftarrow$ New-Queue
2:  while not empty $S$ do
3:      $i \leftarrow$ Dequeue($S$)
4:      $r \leftarrow i + \ell$
5:      if $T[r] \neq 0$ then
6:          Enqueue($E_n[T[r]].S, i$)
7:          if $B_n[T[r]] = 0$ then
8:              $B_n[T[r]] \leftarrow 1$
9:              $E_n[T[r]].\ell \leftarrow \ell + TL[r]$
10:             $E_n[T[r]].ac \leftarrow ac - 1$
11:             Enqueue($Q_1, T[r]$)
12:     else
13:         $v \leftarrow \mu(x[r], \varphi(x[r]))$
14:         Enqueue($E_\Sigma[v].S, i$)
15:         if $B_\Sigma[v] = 0$ then
16:             $B_\Sigma[v] \leftarrow 1$
17:             $E_\Sigma[v].\ell \leftarrow \ell + 1$
18:             $E_\Sigma[v].ac \leftarrow ac + \varphi(x[r]) - 1$
19:             Enqueue($Q_2, v$)
20: while not empty $Q_1$ do
21:     $k \leftarrow$ Dequeue($Q_1$)
22:     Enqueue($Q_3, E_n[k]$)
23:     $E_n[k] \leftarrow$ Clear-Triplet
24:     $B_n[k] \leftarrow 0$
25: while not empty $Q_2$ do
26:     $k \leftarrow$ Dequeue($Q_2$)
27:     Enqueue($Q_3, E_\Sigma[k]$)
28:     $E_\Sigma[k] \leftarrow$ Clear-Triplet
29:     $B_\Sigma[k] \leftarrow 0$
30: while not empty $Q_3$ do
31:     $(S, \ell, ac) \leftarrow$ Dequeue($Q_3$)
32:     if $ac = 0$ then
33:         Output($S, \ell$)
34:         $sc \leftarrow sc + 1$
35:         for each $j \in S$ do
36:             $T[j] \leftarrow sc$
37:             $TL[j] \leftarrow \ell$
38:         Assign-Level($(S, \ell, ac)$)
39:     else
40:         Partition($(S, \ell, ac), x$)
```

Chapter 3

Representation of trees in Automata library

Automata library is a large project, whose development is lead by Ing. Jan Trávníček. Contributions are made by students of Czech Technical University in Prague as part of their theses'. It includes implementations of different data structures, related to but not limited by trees, regular expressions, automata and grammars [6]. Most importantly, it includes algorithms that can be applied to those structures.

The library is written in C++11 and consists of executable applications and dynamic libraries. It follows Unix philosophy and employs principles of modularity and reusability. Each application is designed to do a single task well. Multiple binaries can be chained using Unix pipes to solve a more complex task. Applications of the project utilize XML (Extensible Markup Language) format implemented by Martin Žák [7] for communication.

This thesis is in particular concerned with ranked labeled ordered trees that were implemented by Štěpán Plachý in the scope of his bachelor's thesis[8].

In the following sections I will analyze the internal representation of trees in Automata library.

3.1 Basic trees

Trees in general are implemented as an extension to the `std` namespace. The source code of the implementation is stored in file `tree.hpp` in `alib2std/src/extensions` directory, along with other extensions that are irrelevant to this thesis but nevertheless make development process more convenient. It is a part of the `libalib2std.so` dynamic library.

`std::tree` is a template class. It is up to a programmer to chose the datatype of the contents `m_data` of tree nodes. Every `std::tree` also holds a pointer `m_parent` to its parent node that can be `NULL` in case of a root node. Children, which are objects of the `std::tree` class as well, are stored inside a simple *vector*, which is a part of STL (Standard Template Library).

Class `std::tree` provides postfix, prefix and infix iterators for traversing the structure in different orders.

`std::tree` is not designed to be used in the algorithms directly, instead there are other tree classes that use `std::tree` to hold their inner structure.

3.2 Tree types

Implementations of different tree types that are used in the algorithm are built on top of the `std::tree::TreeBase` class located in the `alib2data/src/tree` and are part of the `libalib2data.so` dynamic library. Trees in the project are split into two groups : *ranked* and *unranked*. The difference was explained in Chapter 1. Every class in each group is a subclass that inherits from either `std::tree::RankedTreeBase` or `std::tree::UnrankedTreeBase`, which in turn are subclasses of the aforementioned `std::tree::TreeBase`.

A simplified class diagram shown in Figure 3.1 gives a straightforward representation of the class hierarchy.

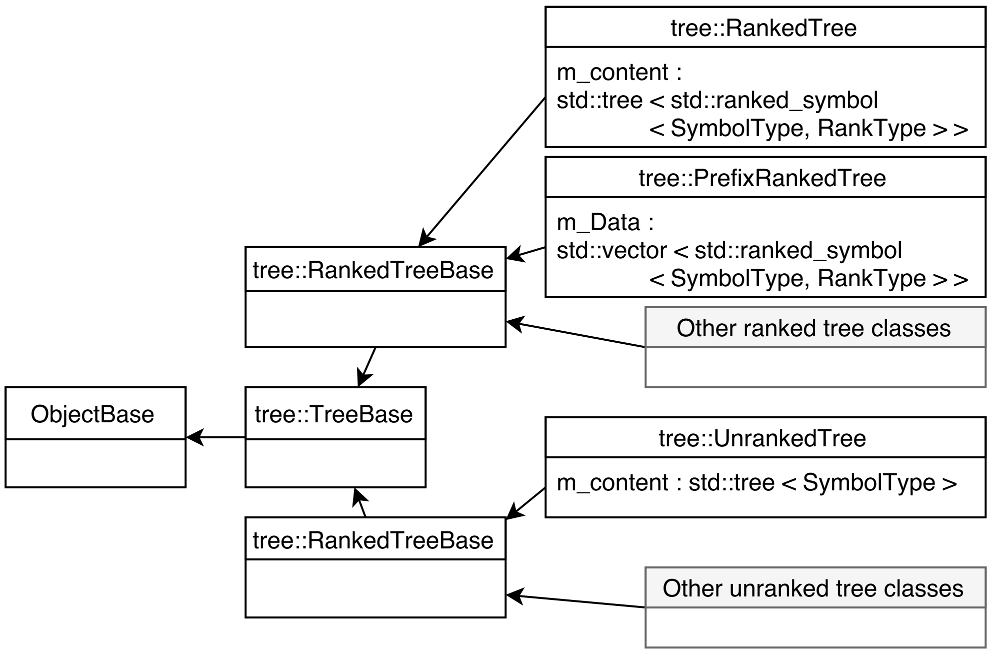
tree::RankedTree
m_content :
std::tree < std::ranked_symbol
< SymbolType, RankType > >
tree::PrefixRankedTree
m_Data :
std::vector < std::ranked_symbol
< SymbolType, RankType > >
tree::RankedTreeBase
Other ranked tree classes
ObjectBase
tree::TreeBase
tree::UnrankedTree
m_content : std::tree < SymbolType >
tree::RankedTreeBase
Other unranked tree classes

Figure 3.1: Simplified class diagram for trees

These classes introduce their own member variables to hold their structure. Hierarchical trees hold their inner structure inside `std::tree`, while prefix trees use *vectors*, because prefix trees have linear structure.

Each class offers methods for their basic structure manipulation and validation.

Postfix trees are not yet implemented in the library, because other algorithms rely on different notations. `tree::PostfixRankedTree` class will be required for the algorithm discussed in Chapter 2 to function.

3.3 Conversion

Class `tree::TreeAuxiliary`, which is located in directory `alib2data/src/tree/common`, provides methods for conversion of trees in cases that were not covered by `std::trees`. For example, as I previously mentioned in Section 3.1, `std::trees` provides iterators that traverse a given tree in post-order, pre-order and in-order. Other conversions, such as a cast from an unranked tree to a ranked one, are implemented in class `tree::TreeAuxiliary`.

3.4 Existing algorithm implementation

The library already contains a naive recursive implementation of a subtree repeat searching algorithm. It works by building a *vector* of nodes in the subtree of every node. Whenever it encounters a node that has not been assigned a subtree repeat number yet, it tries to look it up in a structure that holds each ranked symbol together with a vector of its subtree nodes.

This approach is inefficient, because a collection of subtree nodes need to be built for every node and whenever the algorithm needs to decide whether the subtree in question is a repeat subtree or not, it needs to compare these collections.

3.5 Relevant tools

Automata library offers multiple applications for interacting with tree algorithms. Some of them are:

- `aarbology2` — applies algorithms on trees.
- `arand2` — generates random trees.
- `acast2` — converts a tree from one type to another.

Combined they can be used to test the future implementation and compare obtained results to the output of a preexisting naive recursive algorithm that has already been extensively tested, according to my supervisor.

However, the output of `aarbology2` when it is used to find the subtree repeats in a tree differs from what I initially expected from the algorithm. It does not output distinct subtree repeats in plain text line by line, instead it renames the nodes of the tree — same symbols are assigned to the root nodes of subtrees, only if those subtrees belong to the same subtree repeat. This approach makes further processing of a tree possible.

Figure 3.2 demonstrates how the naive algorithm maps subtree repeats' numbers onto the tree nodes.

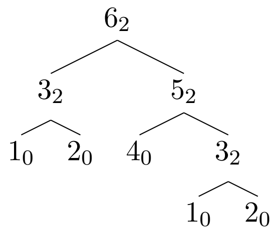

Figure 3.2: Output of the naive algorithm

Chapter 4

Implementation

My goal was to implement the algorithm described in Chapter 2. It had to be integrated into Automata library, that is why the analysis of the library and internal representations of tree structures (Chapter 3) was a crucial part of this undertaking.

4.1 Postfix ranked tree

The first issue that I had to solve was the incompatibility of existing tree types with the algorithm. It requires a tree in postfix notation to operate.

A new class `PostfixRankedTree` had to be created to hold a linear postfix representation of a tree. It belongs to the `tree` namespace and offers functionality similar to the existing class `tree::PrefixRankedTree`. I used postfix iterators offered by the `std::tree` class to implement the constructor of postfix ranked tree class.

4.2 Postfix tree transformation

The algorithm does not change the type and structure of the tree, it outputs its postfix representation, which was not supported by the library before and can not be processed further. To make the algorithm useful to the users of the library, conversion mechanism for `tree::PostfixRankedTrees` to be transformed back into `tree::OrderedRankedTree` have been provided. Finally, class named `tree::TreeAuxiliary` has been modified to include the appropriate method.

4.3 Renaming tree nodes

Different algorithms for computing subtree repeats may assign different IDs to otherwise exact same subtree repeats. To alleviate this problem I have implemented a method that simply renames the nodes of the tree according to a map that it constructs in the process. It preserves subtree repeats and makes comparison of trees produced by different repeat searching algorithms possible. I have placed it into a separate `tree::NormalizeTreeLabels` class.

4.4 Extension of a naive algorithm

I have extended the list of representations supported by the naive algorithm to include `PostfixOrderedTree`, by implementing a method inside `tree::properties::ExactSubtreeRepeatsNaive` class. It functions like its prefix counterpart with changes accounting for obvious differences in tree representations.

4.5 The new algorithm

The most important part of the implementation process, this algorithm has been added to the `tree::properties` namespace, put into a new class `tree::properties::ExactSubtreeRepeats` and has been made available as one of numerous options in the `aarbology2` tool. It is named `exactSubtreeRepeats`.

```
automata-library/bin-release$ ./aarbology2 \
-s $INPUTPATH -a exactSubtreeRepeats
```

Due to the fact that the algorithm takes postfix representation of a tree as its argument, other representations are not supported and need to be converted by the `acast2` application before being passed to `aarbology2`.

A nested class `ExactSubtreeRepeatsAux` that holds auxiliary structures have been created inside `ExactSubtreeRepeats`.

The code follows pseudocode (Section 2.6) as closely as possible. Like other parts of the library it may become a subject of study for other students, that is why the focus was made on readability and explicitness. STL components were used instead of arrays with manually allocated memory for the same reason, as they are easy and safe to use and debug.

Comments have been added to methods and class members. Code has been automatically formatted with *Uncrustify* and *clang-format* tools using configuration files supplied with the library.

Chapter **5**

Tests

In this chapter I will describe the testing process and bugs that it helped me to discover and fix in my code.

To test the implementation during the development phase I have reused the tree in Figure 2.1 from Chapter 2. Unfortunately, the library does not have a GUI (Graphical User Interface), thus the tree had to be transformed into XML format [7] manually (Appendix C). Resulting file *repeats.test1.xml* has been saved in the directory `examples2/tree` following the naming convention established by the previous contributors.

As manual editing is the only way to predefine a tree, the command below can be used to check the XML validity and also visually check the tree structure:

```
automata-library/bin-release$ ./atniceprint \
-i ../examples2/tree/repeats.test1.xml
```

The following predefined test trees have been added to the `examples2/tree` folder:

- `repeats.test1.xml` — the tree I have used to demonstrate the execution of the algorithm in Chapter 2.
- `repeats.test2.xml` — a larger tree used in the article [2].
- `repeats.test3.xml` — a long tree where every node has a different label and a maximum of 1 child.
- `repeats.test4.xml` — a wide tree where every node has a different label and is a leaf except for the root node.
- `repeats.test5.xml` — a long tree where every node has the same label and a maximum of 1 child to test mapping μ.

- `repeats.test6.xml` — a wide tree where every node has the same label and is a leaf except for the root node to test mapping μ.

Example tables and figures in Chapter 2 have been confirmed to be correct by the tests. Trees 5 and 6 which have the same symbol assigned to every node have been processed without a mistake.

Based on the assumption that existing code of the library has been tested and produces correct output, I wrote a *Bash* shell script for automated testing of implemented algorithms and placed it into the root directory of the library source codes. It can be executed by the following command:

```
automata-library$ ./tests.repeats.sh release
```

I chose to write the tests as a script, because this way all parameters of the tests can be easily changed without recompilation, shortening the time it takes to make adjustments to the testing process. The library already contained test files similar to what I have intended to create, so I followed the established code style and design to maintain consistency of the project.

In every case progress of the testing process is printed on the standard output. When a test fails, detailed description of the problematic trees is written to the log file `bin_release/log_tests.txt`. The tests first supply the algorithms with predefined trees from the `examples2` folder and then switch to randomly generated trees.

The script checks operations that have been affected by my implementation. The first one is the conversion of ranked trees to their postfix representation and back. I compare original `RankedTree` to the one that has been converted to `PostfixRankedTree` and then type-cast back to `RankedTree`. This test helped me to discover a flaw in the conversion algorithm I have originally used, that have caused the tree to become vertically mirrored. The bug caused by the children nodes being pushed to the wrong side of the node vector has been fixed.

The second one is the algorithm analyzed in this thesis together with the normalization (renaming) of tree node labels. To test this use case I compare representations of the same tree processed in two different ways that should yield identical results. The first transformation is done by the naive algorithm and the second one is done by the studied algorithm. Labels of both trees are renamed before comparisons to avoid situations when the subtrees are identified correctly but different IDs are assigned to them by different algorithms.

Tests of the algorithm by random trees helped me to discover variables that were named incorrectly because of their similar names that are based on the original article. The problem was undetectable on predefined trees because of their small size and simple structure. One of the trees that highlighted the bug has been added to the examples folder under number 7. The flaw has been fixed.

The final test is similar to the second one and checks the correctness of the postfix extension of naive algorithm. The same tree is being processed in its different representations and then compared.

Indentation issue described in Section 2.5 has been confirmed to be a mistake.

Conclusion

Analysis of the algorithm and tree structures internal representation, implementation of a repetition searching algorithm as part of Automata library and its testing were my goals in this thesis. All these goals were successfully accomplished.

The implementation passes tests by predefined and random trees and produces expected results. As part of the implementation due to the lack of necessary conversion mechanisms additional classes were added to the code of the project apart from the main algorithm. Naive algorithm is left in place for testing purposes and completeness reasons.

During the implementation I have followed pseudocode from [2] as closely as possible to make future inspection, comparison and modification of the code easier. I have respected code conventions of the library to make the process of integration as smooth as possible.

The library is still being developed. My implementation could be extended with support for trees in prefix and other notations. Other students are working right now to make Automata library cover more algorithms and get a GUI. When it is publicly released, it will be used by students of *Algorithms* or *Automata and Grammar* courses to better understand the theory they cover.

Bibliography

[1] Melichar, B.; Janoušek, J.; et al. Arbology: Trees and pushdown automata. *Kybernetika*, volume 48, no. 3, 2012: pp. 402–428.

[2] Christou, M.; Crochemore, M.; et al. Computing all subtree repeats in ordered trees. *Information Processing Letters*, volume 112, no. 24, 2012: pp. 958–962.

[3] Madani, K.; Correia, A.; et al. *Computational Intelligence: Revised and Selected Papers of the International Joint Conference, IJCCI 2013, Vilamoura, Portugal, September 20-22, 2013*. Studies in Computational Intelligence, Springer International Publishing, 2015, ISBN 9783319233925, 142 pp.

[4] Christou, M.; Crochemore, M.; et al. *Computing All Subtree Repeats in Ordered Ranked Trees*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 978-3-642-24583-1, pp. 338–343, doi:10.1007/978-3-642-24583-1_33. Available from: `http://dx.doi.org/10.1007/978-3-642-24583-1_33`

[5] Flouri, T. *Pattern matching in tree structures*. Dissertation thesis, Czech Technical University, Prague, Czech Republic, 9 2012.

[6] Ing. Jan Trávníček / Automata library · GitLab. `https://gitlab.fit.cvut.cz/travnja3/automata-library`, [Online, Accessed on 05/04/2017].

[7] Žák, M. *Automatová knihovna - vnitřní a komunikační formát*. Bacherlor's thesis, Czech Technical University in Prague, 2014.

[8] Plachý, Š. *Automatová knihovna - Stromové automaty a algoritmy nad stromy*. Bacherlor's thesis, Czech Technical University in Prague, 2015.

Appendix **A**

Acronyms

XML	Extensible Markup Language
GUI	Graphical User Interface
STL	Standard Template Library

Appendix B

User manual

B.1 Requirements

The library should support most Linux distributions. Work described in this thesis has been done in a Lubuntu operating system. Automata library requires the following applications and binaries for successful compilation:

- `make` (version 3.9 or later)
- `g++` (version 4.8 or later) or `clang++` (version 3.5 or later)
- `libcppunit-dev`
- `libtclap-dev`
- `libxml2-dev`

`doxygen` and `graphviz` are optional and are needed for generation of documentation.

Please refer to the manual of your package manager for detailed installation procedures. In most cases, `sudo apt-get install` with the name of a respective application will suffice.

B.2 Installation

The code of entire project is compiled by typing the command:

```
automata-library$ make release
```

Each application or dynamic library inside the project can be recompiled separately. For example, to recompile the `libalib2algo.so`, one would execute the command inside `alib2algo` folder:

```
automata-library/alib2algo$ make release
```

The next command will gather binaries of each application inside a single folder `bin-release`:

```
automata-library$ make install-release
```

Now the applications can be executed from folder `bin-release`.

B.3 Execution

The process of detecting subtree repeats in a tree can be split into multiple steps.

1. Generate or load a tree from a file.
2. Obtain postfix representation of the tree.
3. Find subtree repeats.
4. *(Optional)* Transform the tree back to its original form.
5. *(Optional)* Output the tree in a human readable format.

`acast2` is able to combine the first two steps. The goal can be achieved by chaining the following commands:

```
automata-library/bin-release$ ./acast2 -t PostfixRankedTree -i \
 ../examples2/tree/repeats.test1.xml \
 | ./aarbology2 -a exactSubtreeRepeats \
 | ./acast2 -t RankedTree \
 | ./atniceprint
```

Appendix C

Example tree in XML

Following is the XML representation of a tree used for illustration of the algorithm in Chapter 2.

```
<?xml version="1.0"?>
<RankedTree>
  <alphabet>
    <RankedSymbol>
      <Character>a</Character>
      <Unsigned>0</Unsigned>
    </RankedSymbol>
    <RankedSymbol>
      <Character>b</Character>
      <Unsigned>0</Unsigned>
    </RankedSymbol>
    <RankedSymbol>
      <Character>c</Character>
      <Unsigned>2</Unsigned>
    </RankedSymbol>
    <RankedSymbol>
      <Character>d</Character>
      <Unsigned>2</Unsigned>
    </RankedSymbol>
    <RankedSymbol>
      <Character>e</Character>
      <Unsigned>0</Unsigned>
    </RankedSymbol>
    <RankedSymbol>
      <Character>m</Character>
      <Unsigned>2</Unsigned>
    </RankedSymbol>
  </alphabet>
```

```
<content>
  <RankedSymbol>
    <Character>m</Character>
    <Unsigned>2</Unsigned>
  </RankedSymbol>
  <Children>
    <RankedSymbol>
      <Character>c</Character>
      <Unsigned>2</Unsigned>
    </RankedSymbol>
    <Children>
      <RankedSymbol>
        <Character>a</Character>
        <Unsigned>0</Unsigned>
      </RankedSymbol>
      <RankedSymbol>
        <Character>b</Character>
        <Unsigned>0</Unsigned>
      </RankedSymbol>
    </Children>
    <RankedSymbol>
      <Character>d</Character>
      <Unsigned>2</Unsigned>
    </RankedSymbol>
    <Children>
      <RankedSymbol>
        <Character>e</Character>
        <Unsigned>0</Unsigned>
      </RankedSymbol>
      <RankedSymbol>
        <Character>c</Character>
        <Unsigned>2</Unsigned>
      </RankedSymbol>
      <Children>
        <RankedSymbol>
          <Character>a</Character>
          <Unsigned>0</Unsigned>
        </RankedSymbol>
        <RankedSymbol>
          <Character>b</Character>
          <Unsigned>0</Unsigned>
        </RankedSymbol>
      </Children>
    </Children>
    </Children>
  </content>
</RankedTree>
```

Appendix D

Contents of enclosed CD

```
│ readme.txt .......................... a brief description of CD contents
├─ src ............................................ source code directory
│  ├─ automata-library ............ Automata library source code directory
│  └─ thesis .................... LaTeX source code of the thesis directory
└─ text ........................................... compiled text directory
   └─ thesis.pdf ............................. this thesis in PDF format
```