Pattern matching in tree structures

Tomáš Flouri
Department of Theoretical Computer Science, Faculty of Information Technology
Czech Technical University

A thesis submitted to Czech Technical University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The 17th of September 2012
Thesis Supervisor:
Prof. Ing. Borivoj Melichar, DrSc.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9/550, 160 00 Prague 6, Czech Republic
Abstract

In this thesis, we design and analyse algorithms on tree structures mostly based on pushdown automata (PDA).

Motivated by the elegant and intuitive solutions to various string pattern matching problems provided by finite automata in the area of stringology, we were intrigued to transfer those solutions into the tree domain, by linearising (or transforming) a tree structure to a string, and performing tree pattern matching in similar fashion as in the string domain. Due to the fact that trees are generated by context-free grammars, the PDA seemed as an appropriate model to be used for providing solutions for various tree pattern matching problems resembling the finite-automata-based solutions from stringology. In 2008, together with Borivoj Melichar and Jan Janoušek, we founded a new algorithmic discipline, which was given the name arbology from the spanish word árbol, meaning tree, and uses the PDA as its basic computational model. Moreover, a paper by Melichar and Janoušek [69] proved that the PDA is, indeed, an appropriate model of computation as it can accept a superclass of the class of regular tree languages which are accepted by finite tree automata, a popular tool used for tree processing.

This dissertation thesis presents some basic arbology results and principles. First, the linear (string) notations used for describing an ordered tree — ranked or unranked — are presented at the beginning of the thesis and some properties of those notations are given and proved. We then use these linear notations to construct the so-called search PDA, a PDA-based version of the KMP finite automaton, which is used in stringology for locating the occurrences of a pattern within a given string. The search PDA can be constructed from the linear notation $x$ of some tree pattern $p$ (which does not consist of any wildcard symbols), and can find all occurrences of $x$ in some subject tree $t$ in time linear to the size of $t$. Moreover, the size of the search
PDA is linear to the size of the pattern \( p \). The search PDA is subsequently extended to work with multiple tree patterns. Given a set of tree patterns \( p_1, p_2, \ldots, p_r \) (again with no wildcard symbols), we show a method, analogous to the Aho-Corasick finite automaton, to construct a search PDA that can locate all occurrences of all patterns \( p_1, p_2, \ldots, p_r \) in a given subject tree \( t \) in time linear to the size of \( t \). The space required for constructing such an automaton is linear to the sum of sizes of the tree patterns it was constructed for.

The next problem we consider is tree template matching in ranked ordered trees. Given the postfix notation \( x \) of a tree template \( p \), we present a method on how to construct a PDA for \( x \) that can read the postfix notation of a subject tree \( t \) and locate all occurrences of subtrees of \( t \) matching the template \( p \) in time linear to the size of \( t \). We also prove that for certain cases, the size of the constructed PDA may be exponential to the size of the tree template. However, the PDA is constructed once, and can be used arbitrary many times in locating the occurrences of \( p \) in different subject trees. In the subsequent section, we give an algorithm for solving the tree template matching problem for unranked trees. The time and space complexities are the same as in the ranked case.

Another interesting problem we are dealing with is tree indexing and the computation of repeating subtrees in ranked tree structures. Two new kinds of acyclic PDA for trees in prefix notation are presented. First, the subtree PDA constructed over the prefix notation of some ranked tree \( t \) and which accepts all subtrees of tree \( t \). Second, the tree pattern PDA which accepts all tree patterns (including templates) that match some subtree of \( t \). The presented PDA are input-driven and therefore can be transformed to equivalent determinised PDA. Given a tree \( t \) with \( n \) nodes, the deterministic subtree and the deterministic tree pattern PDA represent a complete index of the tree, and the search phase, i.e. locating all occurrences of a subtree or a tree pattern of size \( m \), respectively, is performed in time linear to \( m \) and not depending on \( n \). This is faster than the time required by existing pattern matching algorithms which depend on \( n \). The total size of the deterministic subtree PDA is linear in \( n \). The total size of the deterministic tree pattern PDA may, however, in some cases be exponential to \( n \).

Finally, we consider the problem of computing all subtree repeats in
a given ranked ordered tree. We transform the tree to a string representing its postfix notation, and then present an algorithm based on the bottom-up technique to solve it. The proposed algorithm consists of two phases: the preprocessing phase and the phase where all subtree repeats are computed. Its linear time and space complexity are important parts of its quality, making it efficient in practice.
Acknowledgements

First and foremost I want to thank my supervisor Bořivoj (Bob) Melichar. It has been an honor to be his PhD student. He has been a great source of inspiration and provided me with an insight on how to become a self-motivated researcher. He also struggled to teach me to make simple presentations that even people outside of the field would understand, a thing I yet have to master. I will never forget about his explanation of the finite automaton via a diagram where the states described the status of a person’s life, and the transitions represented the causes that led from one status to another.

I want also to acknowledge Jan Janoušek — my informal supervisor. He has helped me countless times not only in academic issues but also during tough times in my personal life. He and his wife, Kateřina Janoušková, have treated me neither as a student or colleague, but as a close friend, and for this I am indebted to them.

I am also grateful to several colleagues. The first person I should mention is Jan Holub, the head of my department at the Czech Technical University in Prague. He has helped me several times both in my academic and personal life. He has appreciated my work and rewarded me financially through research grants. He has also supported my travels at various conferences and even let me travel with him at times I was not really eligible for travelling. I should also mention Ladislav Vagner from whom I learned a lot about proper C/C++ testing and keeping things really simple when coding. Another source of inspiration is Jan Žďárek, a TeX and Unix guru. I shall also mention Michal Voráček who led my first steps in the world of string algorithms.

Solon P. Pissis deserves a special acknowledgment. The writing of this thesis would not have been possible, in the form in which it has happened, without his friendship, support, and crucial contribution to some of the papers presented in this thesis. It has been a pleasure to cooperate with him and I am looking forward in continuing our cooperation at the Heidelberg Institute in Germany.
Special thanks go also to Michalis Christou for his friendship and contribution to Chapter 6 of this thesis.

I should also mention Costas S. Iliopoulos for his kindness, for inviting me and supporting my travels for the StringMasters and London Stringology Days conferences, and for inviting me to do a Postdoc under his supervision. I want to also thank Maxime Crochemore for his kindness and for the interesting discussions we had in several meetings in Rouen, Blois and London.

I am also grateful to Antonis Alexopoulos, Marta Nováková and René Marek for their unconditional friendship, moral support and patience they have shown during hard times.

Long were my studies, and long are the acknowledgments. Over the years I have come to realise the truth in the statement that the world we live in is composed of more than just work and research. Family is, and has always been, a safe harbour from the world’s turbulences. I feel these last words should belong to my family.

I am grateful to my two best friends — in alphabetical order — Christodoulos Charalambous and Marios Schizas, not only for their unconditional friendship throughout the time we have known each other, but for their concern about me and for treating me as a brother. I consider these two friends as family members.

Final thanks are reserved for my parents Véra and Panikos and my sister Christina, without whom, and their unceasing support and care, I would have never reached this point. I am deeply grateful that you are who you are, that you have always supported me in my decisions and provided me with real parental guidance and a model of life. As an insignificant repayment, I dedicate my work to you.
To my family
Contents

1 Introduction
   1.1 Related work and applications .......................... 20
   1.2 Dissertation structure ................................ 22

2 Preliminaries
   2.1 Alphabet and strings ...................................... 23
   2.2 Trees, tree patterns, tree templates ...................... 25
   2.3 Language, finite automata, pushdown automata ............ 28
   2.4 LR(0) parsing ............................................ 32
   2.5 Asymptotic notation ..................................... 33
   2.6 Elementary data structures ............................... 34
      2.6.1 Stacks and queues ................................ 34
      2.6.2 Linked lists ....................................... 37
      2.6.3 Tries ............................................... 39

3 Linear notations of tree structures ......................... 42
   3.1 Ranked trees ............................................. 42
      3.1.1 Definitions and properties ...................... 43
      3.1.2 Algorithms for working with ranked linear notations . 46
   3.2 Unranked trees .......................................... 49
      3.2.1 Definitions and properties ...................... 50
      3.2.2 Algorithms for working with the unranked linear notations . 53
   3.3 Grammars and pushdown automata for particular linear notations of trees ................................................. 55
      3.3.1 Ranked trees ....................................... 56
### CONTENTS

3.3.2 Unranked trees ........................................... 58

4 Tree Pattern Matching ........................................ 62

4.1 Subtree Matching ........................................... 63
  4.1.1 A naive approach ...................................... 64
  4.1.2 Subtree matching by pushdown automata .............. 65
  4.1.3 Multiple subtree matching by pushdown automata ... 74

4.2 Tree template matching in ranked trees .................... 79
  4.2.1 A naive approach ...................................... 81
  4.2.2 Tree template matching algorithm .................... 82
    4.2.2.1 Match-sets .................................... 82
    4.2.2.2 Computing match-sets ............................ 86
    4.2.2.3 PDA for ranked tree template matching .......... 86

4.3 Tree template matching in unranked trees ................. 92
  4.3.1 Problem definition .................................... 92
  4.3.2 A Naive approach ..................................... 93
  4.3.3 Tree template matching algorithm ................... 94
    4.3.3.1 Match-sets .................................... 94
    4.3.3.2 Computing match-sets ............................ 96
    4.3.3.3 Constructing the action table .................. 97
  4.3.4 Searching phase ...................................... 100

5 Tree indexing .................................................. 104

5.1 Introduction ............................................... 104

5.2 Text indexing structures ................................... 105
  5.2.1 Suffix trees ......................................... 106
  5.2.2 Suffix automaton .................................... 107
  5.2.3 Suffix array ......................................... 109

5.3 Ranked tree indexing structures ........................... 109
  5.3.1 Subtree pushdown automata ............................ 110
  5.3.2 Tree pattern pushdown automaton ..................... 114
  5.3.3 Subtree PDA and repeats in trees .................... 119
  5.3.4 Conclusion ........................................... 123

6 Computing repeats in ranked tree structures ............... 124

6.1 Introduction ............................................... 124

6.2 Algorithm ................................................ 126
  6.2.1 Preprocessing phase .................................. 127
  6.2.2 Computing subtree repeats in unlabeled trees ....... 127
  6.2.3 Example ............................................. 133

6.3 Experimental results ....................................... 135
7 Conclusion
  7.0.1 Contributions ........................................ 137
  7.0.2 Future work ........................................... 139

Bibliography .................................................. 142

Relevant publications of the author ..................... 152

Other publications of the author ......................... 154
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Tree from Example 2</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Singly-linked list</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>Tree $a_2(a_0, a_1(a_0)), a_1(a_0)$ in (a) and its subtrees (b)</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Transition diagram of the basic pushdown automata for ranked alphabet for trees in (a) prefix and (b) postfix notation</td>
<td>57</td>
</tr>
<tr>
<td>3.3</td>
<td>Transition diagram of the basic pushdown automata for unranked alphabet for trees in (a) prefix bar and (b) postfix bar notation</td>
<td>60</td>
</tr>
<tr>
<td>4.1</td>
<td>Graphical representation of the subtree matching problem. Marked with dashed lines are the subtrees of $t$ which are matched by the subtree patterns $p$</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Transition diagram of the (deterministic) PDA accepting tree $t$ in prefix notation $\text{pref}(t) = a_2a_2a_0a_1a_0a_1a_0$ from Example 10</td>
<td>66</td>
</tr>
<tr>
<td>4.3</td>
<td>Transition diagram of the nondeterministic SMPDA accepting language $L = {xa_2a_2a_0a_1a_0a_1a_0</td>
<td>x \in \Sigma^+, ac(x) &gt; 0} \cup {a_2a_2a_0a_1a_0a_1a_0}$</td>
</tr>
<tr>
<td>4.4</td>
<td>Transition diagram of the deterministic SMPDA accepting language $L = {xa_2a_2a_0a_1a_0a_1a_0</td>
<td>x \in \Sigma^+, ac(x) &gt; 0} \cup {a_2a_2a_0a_1a_0a_1a_0}$</td>
</tr>
<tr>
<td>4.5</td>
<td>Transition diagram of the deterministic string matching finite automaton accepting language $L = \Sigma^*{a_2a_2a_0a_1a_0a_1a_0}$</td>
<td>72</td>
</tr>
<tr>
<td>4.6</td>
<td>Set of subtree patterns</td>
<td>75</td>
</tr>
<tr>
<td>4.7</td>
<td>Transition diagram of the (deterministic) PDA accepting the prefix notations of trees $p_1, p_2$ and $p_3$ from Example 13</td>
<td>76</td>
</tr>
<tr>
<td>4.8</td>
<td>Transition diagram of the (deterministic) PDA accepting the prefix notations of trees $p_1, p_2$ and $p_3$ from Example 14</td>
<td>77</td>
</tr>
<tr>
<td>4.9</td>
<td>Transition diagram of deterministic SMPDA constructed for trees $p_1, p_2$ and $p_3$ from Example 15</td>
<td>79</td>
</tr>
<tr>
<td>4.10</td>
<td>Graphical representation of trees $p_1, p_2, p_3, p_4$ with postfix notations $\text{post}(p_1) = a_0Sa_2$, $\text{post}(p_2) = b_0Sa_2$, $\text{post}(p_3) = Sb_0a_2$ and $\text{post}(p_4) = SSa_2$ and trees $t_1, t_2, t_3$ with postfix notations $\text{post}(t_1) = a_0a_0a_2$, $\text{post}(t_2) = b_0b_0a_2$ and $\text{post}(t_3) = a_0b_0a_2$</td>
<td>84</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.11</td>
<td>Graphical representation of proof of Lemma 11</td>
<td>85</td>
</tr>
<tr>
<td>4.12</td>
<td>Graphical representation of (a) tree template $p$ from Example 17 and (b) pushdown store symbols in accordance to the corresponding subtrees of $p$</td>
<td>88</td>
</tr>
<tr>
<td>4.13</td>
<td>Transition diagram of nondeterministic tree template matching PDA from Example 17</td>
<td>89</td>
</tr>
<tr>
<td>4.14</td>
<td>Graphical representation of trees $p_1, p_2, p_3, p_4$ with postfix bar notations $pstb(p_1) = \uparrow \uparrow a \uparrow S$, $pstb(p_2) = \uparrow \uparrow b \uparrow S$, $pstb(p_3) = \uparrow \uparrow S \uparrow bc$ and $pstb(p_4) = \uparrow \uparrow S \uparrow S$ and trees $t_1, t_2, t_3$ with postfix bar notations $pstb(t_1) = \uparrow \uparrow a \uparrow ac$, $pstb(t_2) = \uparrow \uparrow b \uparrow bc$ and $pstb(t_3) = \uparrow \uparrow a \uparrow bc$</td>
<td>96</td>
</tr>
<tr>
<td>4.15</td>
<td>Tree template $p$ from Example 1 having bar notation $pstb(p) = \uparrow \uparrow S \uparrow S \uparrow b a \uparrow \uparrow S \uparrow b \uparrow S \uparrow S a f$</td>
<td>100</td>
</tr>
<tr>
<td>4.16</td>
<td>Diagram of the subject tree $t$ having bar notation $pstb(t) = \uparrow \uparrow b \uparrow \uparrow a \uparrow a \uparrow a \uparrow b a \uparrow b a f \uparrow b a \uparrow b \uparrow a a \uparrow b \uparrow a a \uparrow b \uparrow a a f a f$</td>
<td>102</td>
</tr>
<tr>
<td>5.1</td>
<td>The suffix tree for string $x = banana$</td>
<td>107</td>
</tr>
<tr>
<td>5.2</td>
<td>Suffix automaton (DAWG) for string $x = banana$</td>
<td>108</td>
</tr>
<tr>
<td>5.3</td>
<td>Compacted Suffix automaton (CDAWG) for string $x = banana$</td>
<td>109</td>
</tr>
<tr>
<td>5.4</td>
<td>Transition diagram of non-deterministic subtree PDA $M$ from Example 21 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$</td>
<td>112</td>
</tr>
<tr>
<td>5.5</td>
<td>Transition diagram of the deterministic PDA $M$ from Example 22 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$</td>
<td>113</td>
</tr>
<tr>
<td>5.6</td>
<td>Transition diagram of the deterministic Subtree PDA $M$ from Example 22 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$ after eliminating the transitions that do not lead to any valid prefix notation</td>
<td>113</td>
</tr>
<tr>
<td>5.7</td>
<td>Transition diagram of deterministic tree PDA $M$ from Example 23 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$</td>
<td>115</td>
</tr>
<tr>
<td>5.8</td>
<td>Transition diagram of non-deterministic tree pattern PDA $M$ from Example 24 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$</td>
<td>117</td>
</tr>
<tr>
<td>5.9</td>
<td>Transition diagram of the deterministic Subtree PDA $M$ from Example 25 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$ after eliminating the unnecessary transitions</td>
<td>118</td>
</tr>
<tr>
<td>5.10</td>
<td>Graphical representation of proof of Lemma 18</td>
<td>120</td>
</tr>
<tr>
<td>5.11</td>
<td>Tree $t$ from Example 26</td>
<td>122</td>
</tr>
<tr>
<td>5.12</td>
<td>Transition diagram of the deterministic Subtree PDA $M$ from Example 26 constructed for $x = a_2a_2a_0a_1a_0a_2a_0a_1a_0$ after eliminating the unnecessary transitions</td>
<td>123</td>
</tr>
<tr>
<td>6.1</td>
<td>Tree $t$ from Example 28</td>
<td>133</td>
</tr>
<tr>
<td>6.2</td>
<td>An overview of the partitioning carried out by Algorithm 41</td>
<td>134</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3</td>
<td>Number of operations performed by Algorithm 41 against the number of nodes</td>
<td>136</td>
</tr>
<tr>
<td>7.1</td>
<td>Subject tree $t$ from Example 29</td>
<td>141</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Trace of the (deterministic) PDA $M$ from Example 10 for tree $t$ in prefix notation $\text{pref}(t) = a_2a_2a_0a_1a_0a_1a_0 \ldots$ 66
4.2 Trace of deterministic subtree PDA $M$ from Example 12 for an input tree $t$ with prefix notation $a_2a_2a_0a_1a_0a_0a_1a_1a_2a_0a_0 \ldots$ 73
4.3 Transition table of nondeterministic PDA constructed in Example 17 88
4.4 List of computed match-sets resulting from the transformation of the nondeterministic PDA from Example 17 to a deterministic PDA 90
4.5 State $q_I$ denotes the initial state of the deterministic PDA while state $q_F$ denotes the final state. The transitions are shared between the two states. The tree template is matched when taking either of the two transitions $q_F|Y_6Y_8 \mapsto Y_9$ or $q_F|Y_7Y_8 \mapsto Y_9$ 91
4.6 The sequences of match-sets obtained by lines 7-10 of Algorithm 35. Note that for the sake of saving space, the symbol $*$ represents an arbitrary match-set 99
4.7 The action table constructed by lines 12-14 of Algorithm 35 99
4.8 Dry run of the searching phase 103
5.1 Basic subtree repeat table $BSRT(t)$ from Example 26 122
5.2 Extended subtree repeat table $ESRT(t)$ from Example 26 122
6.1 The arrays $P$, $H$, and $F$ computed during the preprocessing phase 133
6.2 Triplets placed in the level array $L$ during the computation of repeating subtrees for Example 28 135
6.3 Postfix representation of indexed subtrees 136
<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Is-Stack-Empty</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>Is-Stack-Full</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>Push</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Pop</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Is-Queue-Empty</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Is-Queue-Full</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>Enqueue</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>Dequeue</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>List-Search</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>List-Insert</td>
<td>38</td>
</tr>
<tr>
<td>11</td>
<td>List-Delete</td>
<td>39</td>
</tr>
<tr>
<td>12</td>
<td>Trie-Insert</td>
<td>40</td>
</tr>
<tr>
<td>13</td>
<td>Trie-Search</td>
<td>40</td>
</tr>
<tr>
<td>14</td>
<td>Trie-Delete</td>
<td>41</td>
</tr>
<tr>
<td>15</td>
<td>Is-Trie-Empty</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>Subtree-Size-Array-Postfix</td>
<td>47</td>
</tr>
<tr>
<td>17</td>
<td>Node-Parents-Array-Postfix</td>
<td>47</td>
</tr>
<tr>
<td>18</td>
<td>Node-Parents-Array-Prefix</td>
<td>48</td>
</tr>
<tr>
<td>19</td>
<td>Subtree-Height-Array-Postfix</td>
<td>48</td>
</tr>
<tr>
<td>20</td>
<td>Postfix-To-Prefix-Transform</td>
<td>49</td>
</tr>
<tr>
<td>21</td>
<td>Prefix-To-Postfix-Transform</td>
<td>49</td>
</tr>
<tr>
<td>22</td>
<td>Transform-Postfix-Bar-To-Postfix</td>
<td>54</td>
</tr>
<tr>
<td>23</td>
<td>Subtree-Size-Array-Postfix-Bar</td>
<td>55</td>
</tr>
<tr>
<td>24</td>
<td>Subtree-Matching-Naive</td>
<td>64</td>
</tr>
<tr>
<td>25</td>
<td>Tree-Match-PDA</td>
<td>65</td>
</tr>
<tr>
<td>26</td>
<td>Nondeterministic-Subtree-Matching-PDA</td>
<td>68</td>
</tr>
<tr>
<td>27</td>
<td>Subset-Construction-PDA</td>
<td>70</td>
</tr>
<tr>
<td>28</td>
<td>Multiple-Subtree-Matching-Naive</td>
<td>74</td>
</tr>
<tr>
<td>29</td>
<td>Multiple-TreeMatch</td>
<td>75</td>
</tr>
<tr>
<td>30</td>
<td>Multiple-Tree-Search</td>
<td>77</td>
</tr>
<tr>
<td>31</td>
<td>Template-Matching-Ranked-Naive</td>
<td>81</td>
</tr>
<tr>
<td>Number</td>
<td>Algorithm</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>32</td>
<td>Nondeterministic-Tree-Template-Matching-PDA</td>
<td>87</td>
</tr>
<tr>
<td>33</td>
<td>Deterministic-Tree-Template-Matching-PDA</td>
<td>89</td>
</tr>
<tr>
<td>34</td>
<td>Template-Matching-Unranked-Naive</td>
<td>93</td>
</tr>
<tr>
<td>35</td>
<td>Action-Table-Construction</td>
<td>98</td>
</tr>
<tr>
<td>36</td>
<td>Unranked-Tree-Template-Search</td>
<td>101</td>
</tr>
<tr>
<td>37</td>
<td>Subtree-PDA</td>
<td>111</td>
</tr>
<tr>
<td>38</td>
<td>Treetop-PDA</td>
<td>114</td>
</tr>
<tr>
<td>39</td>
<td>Tree-Pattern-PDA</td>
<td>116</td>
</tr>
<tr>
<td>40</td>
<td>Compute-First-Child-Array-Postfix</td>
<td>127</td>
</tr>
<tr>
<td>41</td>
<td>Subtree-Repeats</td>
<td>129</td>
</tr>
<tr>
<td>42</td>
<td>Assign-Level</td>
<td>131</td>
</tr>
<tr>
<td>43</td>
<td>Partition</td>
<td>132</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“To know the road ahead, ask those coming back”

— Chinese proverb

Trees are one of the fundamental data structures in computer science. They form a natural way of expressing a hierarchical structure and are widely used in many applications.

The theory of formal string (or word) languages [4, 65, 97] and the theory of formal tree languages [18, 26, 29, 57, 98] have been extensively studied and developed since the 1950s and 1960s, respectively. Both theories form an important part of the theory of formal languages [96]. The elements of string and tree languages are strings and trees, respectively. Currently, some of the most popular formal models of computation used in the theory of string languages are finite string automata, pushdown string automata, linear bounded automata and Turing machines, whereas in the theory of tree languages the most popular models are various kinds of tree automata.

While trees are usually given as acyclic graph structures, they can also be seen as strings, for example in their prefix (also called preorder) or postfix (also called postorder) notation. These linear notations correspond to (and can be obtained by) a preorder or postorder traversal of the subject tree, respectively. Specifically, it may be said that every sequential algorithm processing a tree structure visits the nodes of the tree in a linear order and thus a corresponding linear notation of the tree is followed.

This thesis presents basic results and principles of arbology. Arbology is a new algorithmic discipline focusing on algorithms on tree structures, whose model of computation is a standard pushdown automaton (PDA) that reads linear notations of trees. We think that arbology is unique in the sense it represents the first systematic approach to solving tree pattern matching problems by means of
PDA, although some particular tree pattern matching algorithms based on PDA, such as the Graham-Glanville technique [59] used for code selection, are known. PDA seem to be an obvious and natural model of computation for algorithms on trees because of the following three facts:

1. Many algorithms for processing trees use recursive procedures, which means that the pushdown store is used for their implementation. For example, finite tree automata are implemented by recursive procedures.

2. Linear notations of trees are context-free languages and PDA are the corresponding models of computation for context-free languages [4, 65, 97]. One of the main arbology results proves that the class of tree languages whose linear notation can be accepted by deterministic PDA is a proper superclass of regular tree languages, which are recognised by finite tree automata [69].

3. The theory of finite automata has been successfully used in stringology [33, 36, 91, 100] — an algorithmic discipline dealing with string processing. Arbology deals with analogous problems in the tree domain that can be solved in a similar fashion.

As already mentioned, the main inspiration for designing arbology algorithms is in the way how finite automata are used to solve many problems from stringology. In stringology, finite automata constitute a handy tool for constructing indexes over strings, performing pattern matching and computing various types of regularities, such as repetitions, covers and seeds [33, 36, 91, 100]. In arbology, PDA are used to resemble the finite-automata-based solutions on strings in order to reflect tree structures. In this way, we have developed methods for indexing trees, computing repeated subtrees, and performing tree pattern matching, which are analogous to the methods of string indexing, string pattern matching and computation of repetitions using finite automata.

There are some differences between finite and pushdown automata theories. One of the major differences is in the existence of an equivalent deterministic automaton to the nondeterministic versions of the two types of automata. For every nondeterministic finite automaton there exists an equivalent deterministic finite automaton which can be constructed using well-known algorithms and methods such as the subset construction (see [81, 107] for algorithms on subset construction). On the contrary, this fact does not generally hold for the case of PDA as for some nondeterministic PDA their equivalent deterministic versions do not exist. Such examples are PDA accepting palindroms of the form $ww^R$, where $w^R$ is the reverse word of $w$. The reason is that a deterministic PDA reading the palindrome from left to right is not able to find the center, i.e. the end of $w$ and the beginning of the reversed $w$. Generally, it is not known how to decide
whether for a given nondeterministic PDA exists a deterministic equivalent. Nevertheless, several classes of PDA for which determinisation (i.e. construction of an equivalent deterministic automaton) is possible. These classes are called input-driven [104], visibly [8] and real-time height-deterministic [93]. Note that many of the PDA which will be presented in the subsequent chapters are input-driven PDA.

1.1 Related work and applications

Existing algorithms processing tree structures are described by a plethora of formalisms, such as tree automata, term-rewriting systems, pushdown automata, or are directly described by programmes written in specific programming languages.

Finite tree automata recognise regular tree languages and may be the most researched kind of tree automata [18, 26, 29, 57]. As it was shown in [69], any problem which can be solved by a finite tree automaton can also be solved by a deterministic PDA. In [70, 94] it is shown that the so-called pushdown tree-walking automata recognise exactly the class of regular tree languages. The underlying principle of a method for transformation of finite tree automata to pushdown tree-walking automata [94] is used in [69] for the transformation of finite tree automata to deterministic pushdown automata. Examples of other existing kinds of tree automata are the pushdown tree automata and generalised tree automata. For more information, see [29]).

Comparing arbology algorithms with the known methods from the theory of tree automata [26, 29], the arbology methods of indexing trees and finding repeating subtrees have no equivalent known solutions by means of tree automata, as opposed to the case of tree pattern matching methods, where equivalent solutions exist described by tree automata.

The study of the tree pattern matching problem started with the paper of Hoffmann and O’Donnell [63], who presented the first non-trivial algorithm (called the top-down algorithm in their paper) to find all the nodes of the subject tree admitting a compact occurrence of the pattern. However, this algorithm has a complexity of $O(nm)$ in the worst case, where $n$ is the number of nodes of the subject tree and $m$ the number of nodes of the pattern. The first algorithm with a better time complexity was presented by Kosaraju [78], who proposed an algorithm running in $O(nm^{3/2}\text{polylog}(m))$ time. This result has been improved successively by Dubiner, Galil and Magen [44], Cole and Hariharan [27], and Cole, Hariharan and Indyk [28], who achieved an almost linear $O(n\log^{3} n)$ time complexity. An algorithm for tree pattern matching with a more general notion of occurrence of the pattern was later introduced by Chauve [21] along with a quadratic worst-case time complexity algorithm for tree pattern matching which,
however, in practice shows a linear time behaviour [20].

Another point worth mentioning is that formalisms for describing semantics are also used for tree processing; there exists a tool named YakYak [38], which is a preprocessor for yacc-compatible generators and serves for generating parsers of the regular tree languages. The output of YakYak is not a syntax defining context-free grammar only, but it is an attributed context-free grammar in which the constraints defining regular tree languages are described not by the syntactic rules, but by the semantic attribute rules (see [5, 41] for the definition and for further information on attributed grammars). As a result, the parser generated by the YakYak + yacc-compatible generator behaves as a deterministic PDA which recognises regular tree languages by an extended attribute semantic evaluation.

An example of a well researched problem, whose solutions have been described by various models of computation, is the code selection problem in compiler backends. The task is to cover the intermediate programme representation which is a tree structure by appropriate target machine code instructions which are represented by tree patterns, and to select the best possible such covering. The best possible covering is usually selected according to the result of the evaluation of a cost function, which describes the cost of the machine code instructions. For the purpose of tree covering, a number of tree pattern matching methods are generally used [20, 21, 27, 28, 44, 63, 78]. A code selection method based on deterministic finite tree automata can be found in [46], where the cost function is computed by an additional semantic evaluation. On the other hand, [59, 82, 99] describe the code selection methods based on deterministic PDA performing the tree pattern matching, where the tree patterns are represented by rules of a context-free grammar, and in this way, generally ambiguous and non-LR(0) context-free grammars are created. Consequently, the LR(0) parsers for those grammars contain conflicts. In [59] these conflicts are resolved by some heuristics; in [82, 99] a special construction of a deterministic parser is used, which corresponds to a determination of the above mentioned LR(0) parser with conflicts. Another code selection method is provided by a family of tools called BURG and IBURG [55, 56], which use tree-rewriting systems as the model of computations. Note, that another code selection method based on the deterministic pushdown automaton would result from the transformation of the deterministic tree automaton in [46] using the method described in [69], where the evaluation of the code function would be implemented by an attribute semantic evaluation. In addition, the transformation gives an unambiguous LR(0) grammar, which means that the resulting code generator could be implemented easily with the use of an existing (yacc-like) parser generator for that grammar.

Models of computations for various linearised forms of unranked trees and their relationship to regular tree languages have been already studied in some papers – the so-called nested words and visibly pushdown languages in [7] and
[8], respectively.

1.2 Dissertation structure

This thesis consists of 7 chapters. Apart from this first, introductory chapter, Chapter 2 contains all mathematical and notational preliminaries necessary for reading the remaining of this thesis. That chapter may be skipped and referred back to as necessary. In Chapter 3, we introduce and discuss the linear notations of tree structures describing both ranked and unranked trees, as well as their properties. We also give several algorithms working with these notations. Chapter 4 is about the problem of tree pattern matching and is split in three parts: The first part discusses the problem of subtree matching, i.e. the tree pattern is a full tree with no wildcard symbols. The second part describes a method for tree pattern matching in ranked ordered trees using pushdown automata, while the third part presents an algorithm for tree pattern matching in unranked ordered trees. Chapter 5 presents methods for tree indexing via pushdown automata which are directly analogous to string indexing methods using finite automata. Chapter 6 presents an elegant algorithm for computing all subtree repeats in ranked tree structures. Finally, Chapter 7 gives an overview of the problems solved in this thesis, and presents some open problems and directions for future research.
Chapter 2

Preliminaries

“If you wish to make an apple pie truly from scratch, you must first invent the universe”


In this chapter, we present basic notations, definitions and properties which will be used throughout the rest of this dissertation thesis. This chapter may be skipped on the first reading and referred back to as necessary.

**Notation 1 (Natural Numbers)** We denote the set of natural numbers by \( \mathbb{N} \).

**Notation 2 (Cardinality)** The cardinality of a set \( X \) is denoted by \( \sigma(X) \).

**Definition 1 (Powerset)** The set of all subsets of a set \( X \) is denoted by \( \mathcal{P}(X) \) or, more simply, \( 2^X \).

### 2.1 Alphabet and strings

**Definition 2 (Alphabet)** An alphabet \( \Sigma \) is a finite non-empty set whose elements are called symbols.

**Notation 3 (Alphabet size)** The size of an alphabet \( \Sigma \) is the number of its elements and is denoted by \( |\Sigma| \).

**Definition 3 (String)** A string on an alphabet \( \Sigma \) is a finite, possibly empty, sequence of elements of \( \Sigma \).
Notation 4 (Empty string) The string of zero elements from any alphabet is called the empty-string, and is denoted by $\varepsilon$.

Notation 5 (Set of all strings) The set of all strings (including the empty string) on an alphabet $\Sigma$ is denoted by $\Sigma^*$.

Notation 6 (Set of all non-empty strings) The set of all non-empty strings on an alphabet $\Sigma$ is denoted by $\Sigma^+$. It holds that $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Notation 7 (Length of string) The length of a string $x$ is defined as the length of the sequence associated with the string $x$, and is denoted by $|x|$.

We denote by $x[i]$, for all $1 \leq i \leq |x|$, the letter at index $i$ of string $x$. Each index $i$, is a position in $x$ when $x \neq \varepsilon$. It follows that the $i$-th letter of $x$ is the letter at position $i$ in $x$, and that $x = x[1..|x|]$.

Definition 4 (Concatenation of strings) The concatenation of two strings $x$ and $y$ is the string of the symbols of $x$ followed by the symbols of $y$. It is denoted by $xy$. We will also use the notation $x \cdot y$ for clarity when necessary.

Definition 5 (Power of string) For every string $x$ and every natural number $n$, we define the $n$-th power of string $x$, denoted by $x^n$, by $x^0 = \varepsilon$ and $x^k = x^{k-1}x$, for all $1 \leq k \leq n$.

Definition 6 (Factor of string) A string $x$ is a factor of a string $y$ if there exist two strings $w$ and $z$, such that $y = wxz$.

Definition 7 (Proper factor of string) A factor $x$ of a string $y$ is proper if $x \neq y$.

Definition 8 (Prefix of string) Let $x, y, w$ and $z$ be strings such that $y = wxz$. If $w = \varepsilon$, then $x$ is a prefix of $y$.

Definition 9 (Suffix of string) Let $x, y, w$ and $z$ be strings such that $y = wxz$. If $z = \varepsilon$, then $x$ is a suffix of $y$.

Notation 8 (Set of suffixes of a string) The set of all suffixes of a string $x$ is denoted as $\text{Suff}(x)$.

Notation 9 (Set of prefixes of a string) The set of all prefixes of a string $x$ is denoted as $\text{Pref}(x)$.
Definition 10 (Occurrence of string) Let \( x \) and \( y \) be two strings such that \( x \) is non-empty. We say that there exists an occurrence of \( x \) in \( y \), or, more simply, that \( x \) occurs in \( y \), if and only if \( x \) is a factor of \( y \).

Every occurrence of \( x \) can be specified by a position in \( y \). Thus, we say that \( x \) occurs at the starting position \( i \) in \( y \) when \( y[i \ldots i + |x| - 1] = x \). It is sometimes more suitable to consider the ending position \( i + |x| - 1 \).

Example 1 Let the string \( x = \text{aba} \) and the string \( y = \text{babaababa} \). The starting and ending positions, where \( x \) occurs in \( y \), are

| \( i \) | 0 1 2 3 4 5 6 7 8 |
|---|---|---|---|---|---|---|---|---|
| \( y[i] \) | b a b a a b a b a |
| starting positions | 1 4 6 |
| ending positions | 3 6 8 |

2.2 Trees, tree patterns, tree templates

Based on concepts from graph theory (see [42]), we use the following notions and definitions.

Definition 11 (Ranked alphabet) A ranked alphabet \( \mathcal{A} \) is a couple \( \mathcal{A} = (\Sigma, \varphi) \) such that \( \Sigma \) is an alphabet (a finite, non-empty set of elements) and \( \varphi: \Sigma \mapsto \mathbb{N} \) called the ranking function.

Notation 10 (Ranked alphabet size) The size of a ranked alphabet \( \mathcal{A} = (\Sigma, \varphi) \) is the size of alphabet \( \Sigma \) and is denoted by \( |\mathcal{A}| \).

Definition 12 (Graph) A graph \( G \) is a pair \( (V, E) \) of disjoint sets such that \( E \subseteq V \times V \). We call the set \( V \) vertices (or nodes) and the set \( E \) edges.

Definition 13 (Directed graph) A directed graph is a graph \( G = (V, E) \) together with two maps \( f: E \mapsto V \) and \( g: E \mapsto V \) assigning to every edge \( e \in E \) an initial vertex \( f(e) \) and a terminal vertex \( g(e) \). The edge \( e \) is said to be directed from \( f(e) \) to \( g(e) \) and is represented as \( (f(e), g(e)) \).

Definition 14 (Path) Given a graph \( G = (V, E) \), a sequence of \( n + 1 \) vertices \( v_0, v_1, \ldots, v_n \in V \), where \( n \geq 1 \) is called a path of length \( n \) from vertex \( v_0 \) to \( v_n \), if and only if the edges \( (v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n) \) exist.

Definition 15 (Cycle) A cycle is a path \( v_0, v_1, \ldots, v_n \), where \( v_0 = v_n \).
Definition 16 (Directed path) Given a directed graph graph \( G = (V, E) \) and the two mappings \( f \) and \( g \), a sequence of vertices \( v_0, v_1, \ldots, v_n \), where \( n \geq 1 \) is a directed path of length \( n \) from vertex \( v_0 \) to \( v_n \), if there exist \( n \) edges \( e_i \) such that \( f(e_i) = v_{i-1} \) and \( g(e_i) = v_i \), for all \( i \) such that \( 1 \leq i \leq n \) and \( e_i \in E \).

Definition 17 (Connected graph) A non-empty graph \( G = (V, E) \) is called connected if any two of its vertices are linked by a path in \( G \).

Definition 18 (Acyclic graph) A graph is acyclic if it has no cycles.

Definition 19 (In-degree of a node) Given a directed graph \( G = (V, E) \), the in-degree of a node \( v \) is the number of distinct pair \( (u, v) \), where \( u \in V \).

Definition 20 (Out-degree of a node) Given a directed graph \( G = (V, E) \), the out-degree of a node \( v \) is the number of distinct pair \( (v, u) \), where \( u \in V \).

Definition 21 (Degree of a node) Given a directed graph \( G = (V, E) \), the degree of a node \( v \) is the number of its in-degree and out-degree.

Definition 22 (Tree) A tree is an acyclic connected graph.

Definition 23 (Rooted tree) Any node of a tree can be selected as the root of the tree. A tree with a root is called rooted tree.

Notation 11 (Leaf node) Any node of a tree with out-degree 0 is called a leaf.

Definition 24 (Ordered tree) An ordered tree is a tree \( t = (V, E) \), where the direct descendants (children) \( v_1, v_2, \ldots, v_m \) of any node \( v \in V \) with arity \( m \), are ordered.

Definition 25 (Labeled tree) A tree \( t = (V, E) \) is labeled if every node \( v \in V \) is labeled by a symbol \( a \in \Sigma \), where \( \Sigma \) is an alphabet. In other words, there exists a function \( \ell : V \rightarrow \Sigma \), such that every node of \( V \) is mapped to some symbol of \( \Sigma \). Function \( \ell \) is called the labeling function.

Definition 26 (Ranked tree) Given a tree \( t = (V, E) \) and a ranked alphabet \( A = (\Sigma, \varphi) \), \( t \) is said to be ranked if every node \( v \in V \) is labeled by some symbol \( a \in \Sigma \) and the out-degree of \( v \) is given by \( \varphi(a) \). In other words, there exists a function \( \ell : V \rightarrow \Sigma \), such that every node of \( v \in V \) is mapped to some symbol of \( \Sigma \), and the out-degree of \( v \) is given by \( \varphi(\ell(v)) \).

Definition 27 (Unranked tree) A tree \( t \) is unranked if its nodes are labeled by some alphabet \( \Sigma \), and the out-degree of its nodes is not given by that alphabet symbol.
Definition 28 (Directed tree) A tree is called directed if it is composed of a directed acyclic graph.

Definition 29 (Rooted directed tree) A rooted and directed tree is a directed acyclic graph \( t = (V, E) \) with a special node \( v \in V \), called the root, such that

1. \( v \) has in-degree 0,
2. all other nodes of \( t \) have in-degree 1,
3. there is just one path from the root \( v \) to every \( u \in V \), such that \( u \neq v \).

Definition 30 (Height of rooted tree) Given a rooted tree \( t \), its height is the length of the longest of the longest path originating from the root node of \( t \) and leading to some leaf node. We denote it by \( h(t) \).

In the rest of this thesis, we will refer to trees as labeled, directed, ordered trees unless stated otherwise. In each chapter (or section) it will be stated whether we refer to ranked or unranked trees.

![Figure 2.1: Tree from Example 2](image)

Example 2 Let \( t = (V, E) \) be the ranked, ordered, labeled and rooted tree illustrated in Figure 2.1. Let \( V = (v_1, v_2, \ldots, v_7) \) be the set of nodes such that \( v_1 \) is the root node, \( A = (\Sigma, \varphi) \) is the ranked alphabet and \( \ell : V \to \Sigma \) the labeling function that maps each node to a specific symbol of \( \Sigma \) in the following way:

\[
\begin{align*}
v_1 & \mapsto a_2 \\
v_3 & \mapsto a_1 \\
v_5 & \mapsto a_1 \\
v_7 & \mapsto a_0 \\
v_2 & \mapsto a_2 \\
v_4 & \mapsto a_0 \\
v_6 & \mapsto a_0
\end{align*}
\]
and $E$ is a set consisting of the following ordered sequences of pairs

\[
((v_1, v_2), (v_1, v_3)),
((v_2, v_4), (v_2, v_5)),
((v_5, v_7)),
((v_3, v_6))
\]

For simplicity, we have encoded the rank of each alphabet symbol in the label as a subscript index.

In the next chapter, we will informally describe rooted, labeled, ordered and directed trees using a parenthesis notation, where each node is described by its label followed by an ordered list of its children nodes. Each node recursively lists its label followed by a list of its children and so on. The parenthesis notation for the tree from Example 2 is $a_2(a_2(a_0, a_1(a_0)), a_1(a_0))$.

### 2.3 Language, finite automata, pushdown automata

We define notions from the theory of string languages similarly as they are defined in [34, 65, 91].

**Definition 31 (Language)** A language over an alphabet $\Sigma$ is the set of strings over $\Sigma$.

**Definition 32 (Language concatenation)** A language $L = L_1L_2$ resulting from the concatenation of two two languages $L_1$ and $L_2$ is defined as $L = \{ xy \mid x \in L_1, y \in L_2 \}$.

**Definition 33 (Deterministic finite automaton)** A deterministic finite automaton (DFA) $M$ on the alphabet $\Sigma$ is a quintuple

\[
M = (Q, \Sigma, \delta, q_I, F)
\]

where

- $Q$ is a finite set of states,
- $\delta$ is a mapping $\delta : Q \times \Sigma \mapsto Q$ called the transition function,
- $q_I \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states.
Definition 34 (Transitions of DFA) Let $M = (Q, \Sigma, \delta, q_I, F)$ be a deterministic finite automaton. The relation $\vdash_M \in (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ is called a transition in $M$. For $\delta(q, a) = p$, then $(q, aw) \vdash_M (p, w)$ for any $w \in \Sigma^*$.

Definition 35 (Nondeterministic finite automaton) A nondeterministic finite automaton $M$ on the alphabet $\Sigma$ is a quintuple $M = (Q, \Sigma, \delta, q_I, F)$ where

- $Q$ is a finite set of states,
- $\delta$ is a mapping $\delta : Q \times \Sigma \mapsto \mathcal{P}(Q)$ called the transition function,
- $q_I \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states.

Remark 1 (Nondeterministic finite automaton) In some literature the set $I \subseteq Q$ is used instead of $q_I$ when defining the nondeterministic finite automaton, i.e. the automaton consists of a set of initial states.

Definition 36 (Extended transition function) Let $M = (Q, \Sigma, \delta, q_I, F)$ be some deterministic (resp. nondeterministic) finite automaton, where $\delta : Q \times \Sigma \mapsto Q$ (resp. $\delta : Q \times \Sigma \mapsto \mathcal{P}(Q)$) is its transition function. The extended transition function is the mapping $\delta^* : Q \times \Sigma^* \mapsto Q$ (resp. $\delta^* : Q \times \Sigma^* \mapsto \mathcal{P}(Q)$) and is defined as:

$\delta^* (q, \varepsilon) = q$

$\delta^* (q, xa) = \delta (\delta^* (q, x), a)$

where $x \in \Sigma^*$ and $a \in \Sigma$.

Definition 37 The language accepted by some finite automaton $M = (Q, \Sigma, \delta, q_I, F)$ is defined as

$L(M) = \{ x \mid x \in \Sigma^*, \delta^* (q_I, x) \in F \}$

Definition 38 (Equivalence) Two finite automata are said to be equivalent if they accept the same language.

Lemma 1 (Subset construction) For every nondeterministic finite automaton $M$, there exists an equivalent deterministic finite automaton $M'$ accepting the same language, i.e. $L(M) = L(M')$
Proof Let \( M = (Q, \Sigma, \delta, I, F) \) be the nondeterministic finite automaton. We define the equivalent deterministic finite automaton as \( M' = (X, \Sigma, \delta', I, F') \) such that

\[
X \in \mathcal{P}(Q) \\
F' = \{ K \mid K \in \mathcal{P}(Q), K \cap F \neq \emptyset \} \\
\delta'(K, a) = \{ p \mid \delta(q, a), \forall q \in K, a \in \Sigma \}
\]

We must now prove that \( L(A) = L(B) \):

First, we prove the trivial case of the empty string:

\[
\varepsilon \in L(M) \iff I \cap F \neq \emptyset \iff I \in F' \iff \varepsilon \in L(M')
\]

Second, we prove the following two cases:

- \( L(M) \subseteq L(M') \)
  A string \( x[1..n] \in L(M) \) if and only if \( \exists q_1, q_2, \ldots, q_{n+1} \in Q \), such that \( q_1 \in I, q_{i+1} \in \delta(q_i, x[i]), \) and \( q_{n+1} \in F \), for \( 1 \leq i \leq n \). Now let \( K_1 = I \), and \( K_{i+1} = \delta'(K_i, x[i]) \), for \( 1 \leq i \leq n \). It holds that \( q_i \in K_i \) for \( 1 \leq i \leq n + 1 \), and thus \( K_{n+1} \in F' \).

- \( L(M') \subseteq L(M) \)
  A string \( x[1..n] \in L(M') \) if and only if \( \exists K_1, K_2, \ldots, K_{n+1} \in \mathcal{P}(Q) \), such that \( K_1 = I, K_{i+1} = \delta(K_i, x[i]), K_{n+1} \in F' \), for \( 1 \leq i \leq n \). Now let \( q_{n+1} \in F \cap K_{n+1} \), and \( q_i \in K_i \), such that \( q_{i+1} \in \delta(q_i, x[i]) \) for \( 1 \leq i \leq n \). It holds that \( q_1 \in K_1 \) and thus \( q_1 \in I \).

**Definition 39 (Nondeterministic pushdown automaton)** A nondeterministic pushdown automaton on the alphabet \( \Sigma \) is a septuple

\[
M = (Q, \Sigma, G, \delta, q_I, Z_I, F)
\]

where

- \( Q \) is a finite set of states,
- \( G \) is the pushdown store alphabet,
- \( \delta \) is a mapping \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \times G \rightarrow Q \times G^* \),
- \( q_I \in Q \) is the initial state,
- \( Z_I \in G \) is the initial pushdown store symbol,
- \( F \subseteq Q \) is the set of final states.

**Definition 40 (Extended nondeterministic pushdown automaton)** An extended nondeterministic pushdown automaton on the alphabet \( \Sigma \) is a nondeterministic PDA

\[
M = (Q, \Sigma, G, \delta, q_I, Z_I, F)
\]

where its transition function \( \delta \) is defined as:

\[
\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times G^* \rightarrow Q \times G^*
\]
For simplicity, in the rest of the text, we will sometimes use the notation \( p \xrightarrow{a}{\alpha} M \rightarrow q \beta \) when referring to the transition \( \delta(p, a, \alpha) = (q, \beta) \).

**Definition 41 (Extended deterministic pushdown automaton)** An extended pushdown automaton 
\[ M = (Q, \Sigma, G, \delta, q_I, Z_I, F) \]
is called extended deterministic pushdown automaton (deterministic PDA) if the following conditions hold:

1. \( |\delta(q, a, \alpha)| \leq 1 \), for all \( q \in Q, a \in \Sigma \cup \{\varepsilon\}, \alpha \in G^* \).
2. If \( \delta(q, a, \alpha) \neq \emptyset \), \( \delta(q, a, \beta) \neq \emptyset \) and \( \alpha \neq \beta \) then \( \alpha \notin \text{Pref}(\beta) \) and \( \beta \notin \text{Pref}(\alpha) \).
3. If \( \delta(q, a, \alpha) \neq \emptyset \), \( \delta(q, \varepsilon, \beta) \neq \emptyset \), then \( \alpha \notin \text{Pref}(\beta) \) and \( \beta \notin \text{Pref}(\alpha) \).

**Definition 42 (Transitions of PDA)** Let \( M = (Q, \Sigma, G, \delta, q_I, Z_I, F) \) be a pushdown automaton. The relation \( \vdash_M \subseteq (Q \times \Sigma^* \times G^*) \times (Q \times \Sigma^* \times G^*) \) is called a transition of \( M \). It holds that \( (q, aw, \alpha \beta) \vdash_M (p, w, \gamma \beta) \) if \( (p, \gamma) \in \delta(q, a, \alpha) \). The \( k \)-th power, transitive closure, and transitive and reflexive closure of the relation \( \vdash_M \) is denoted \( \vdash_M^k, \vdash_M^+, \vdash_M^* \), respectively.

**Definition 43 (Acceptance by pushdown automata)** A language \( L \) accepted by a pushdown automaton 
\[ M = (Q, \Sigma, G, \delta, q_I, Z_I, F) \]
is defined in two distinct ways:

1. Accepting by final state:
\[ L(M) = \{ x \mid (q_I, x, Z_0) \vdash_M^*(q, \varepsilon, \gamma), \ x \in \Sigma^*, \ \gamma \in G^*, \ q \in F \} \]
2. Accepting by empty pushdown store:
\[ L_e(M) = \{ x \mid (q_I, x, Z_I) \vdash_M^*(q, \varepsilon, \varepsilon), \ x \in \Sigma^* q \in Q \} \]

If the pushdown automaton accepts the language by empty pushdown store then the set \( F \) of final states is the empty set.

**Definition 44 (Formal grammar)** A formal grammar is a quadruple 
\[ G = (N, T, P, S) \]
where

- $N$ is a finite set of non-terminal symbols,
- $T$ is a finite set of terminal symbols,
- $P$ is a finite subset of the set $(N \cup T)^* \times (N \cup T)^*$ called the production rules,
- $S \in N$ is the initial nonterminal symbol of $G$.

**Definition 45 (Regular grammar)** A formal grammar $G = (N, T, P, S)$ is called regular, if every production rule is of the form $A \rightarrow aB$ or $A \rightarrow a$, where $A, B \in N$ and $a \in T$. The set of production rules may also contain the rule $S \rightarrow \varepsilon$ only in the case that $S$ does not occur on the right hand side of any other production rule.

**Definition 46 (Context-free grammar)** A formal grammar $G = (N, T, P, S)$ is called context-free, if every production rule is of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.

**Definition 47 (Derivation)** Given a formal grammar $G = (N, T, P, S)$, the relation $\Rightarrow$ is called derivation if $\alpha A \gamma \Rightarrow \alpha \beta \gamma$, where $A \in N$ and $\alpha, \beta, \gamma \in (N \cup T)^*$, then the rule $A \rightarrow \beta$ is in $P$. The symbols $\Rightarrow^+$, and $\Rightarrow^*$ are used for the transitive, and the transitive and reflexive closure of $\Rightarrow$, respectively.

**Definition 48 (Language generated by grammar)** Given a formal grammar $G = (N, T, P, S)$, the language generated by $G$, denoted by $L(G)$, is the set of strings

$$L(G) = \{ w \mid S \Rightarrow^* w, \ w \in T^* \}$$

### 2.4 LR(0) parsing

Given a string $w$, an LR(0) parser for a context-free grammar $G = (N, T, P, S)$ reads the string $w$ from left to right without any backtracking, and is implemented by a deterministic pushdown automaton.

A string $\gamma$ is a viable prefix of $G$, if $\gamma$ is a prefix of $\alpha \beta$, and $S \Rightarrow^*_m \alpha Ax \Rightarrow^*_m \alpha \beta x$ is a rightmost derivation in $G$; the string $\beta$ is called the handle. We use the term complete viable prefix to $\alpha \beta$ in its entirety. The contents of the pushdown store, during parsing, correspond to a viable prefix.

The standard LR(0) parser performs two kinds of transitions:

1. When the contents of the pushdown store correspond to a viable prefix containing an incomplete handle, the parser performs a shift, which reads one symbol $a$ and pushes a symbol corresponding to $a$ onto the pushdown store.
2. When the content of the pushdown store corresponds to a viable prefix by the handle \( \beta \), the parser performs a reduction by a rule \( A \rightarrow \beta \). The reduction pops \(|\beta|\) symbols from the top of the pushdown store and pushes a symbol corresponding to \( A \) onto the pushdown store.

A context-free grammar \( G \) is LR(0) if the two conditions for \( G \):

1. \( S \Rightarrow^* \alpha Aw \Rightarrow \alpha \beta w \),
2. \( S \Rightarrow^* \gamma Bx \Rightarrow \alpha \beta y \),

imply that \( \alpha Ay = \gamma Bx \), that is, \( \alpha = \gamma \), \( A = B \), and \( x = y \).

If the context-free grammar \( G \) is not an LR(0) grammar, then the pushdown automaton constructed as an LR(0) parser contains conflicts, which means the next transition to be performed cannot be determined according to the contents of the pushdown store only.

For context-free grammars without hidden left and right recursions, the number of consecutive reductions between the shifts of two adjacent symbols cannot be greater than a constant, and therefore the LR(0) parser for such a grammar can be optimised by precomputing all its reductions beforehand. Then, the optimised LR(0) parser reads one symbol on each of its transition [12].

For more details on LR parsing, see [3, 4].

2.5 Asymptotic notation

In this section, we introduce the asymptotic notation from [30]. Asymptotic notation applies to functions, which will usually characterise the runtime of algorithms. However, asymptotic notation can apply to functions that characterise other aspects of algorithms such as the amount of space they use.

Let \( f : \mathbb{N} \mapsto \mathbb{N} \) and \( g : \mathbb{N} \mapsto \mathbb{N} \) be two mappings.

**Definition 49 (\( \Theta \)-notation)** We denote by \( \Theta(g(n)) \) the set of functions

\[
\Theta(g(n)) = \{ f(n) | \exists c_1, c_2, n_0 \in \mathbb{N} : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}.
\]

**Definition 50 (\( O \)-notation)** We denote by \( O(g(n)) \) the set of functions

\[
O(g(n)) = \{ f(n) | \exists c, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c g(n), \forall n \geq n_0 \}.
\]

**Definition 51 (\( o \)-notation)** We denote by \( o(g(n)) \) the set of functions

\[
o(g(n)) = \{ f(n) | \exists c, n_0 \in \mathbb{N} : 0 \leq f(n) < c g(n), \forall n \geq n_0 \}.
\]
2.6 Elementary data structures

In this section, we introduce elementary data structures from [30].

Sets and multisets, i.e., sets that allow an element to occur more than once, are fundamental to computer science as they are to mathematics. Whereas mathematical sets can be infinite and unchanging, the sets manipulated by algorithms are usually finite and can grow, shrink, or change over time. We call such sets *dynamic*.

In a typical implementation of a dynamic set, each element of the set is represented by an object, whose attributes is a *tuple* \((a_0, a_1, \ldots, a_n)\), such that \(n \geq 0\), that can be examined and manipulated if we have a pointer to the object.

Here is a list of typical operations on a dynamic set:

- **empty()**: an operation that creates and returns an empty set,
- **is-empty(S)**: an operation that returns `true` if set \(S\) is empty, and `false` otherwise,
- **search(S, k)**: a query that, given a set \(S\) and a key value \(k\), returns a pointer \(x\) to an element in \(S\), such that \(x.key = k\), or nil if no such element belongs to \(S\),
- **insert(S, x)**: a modifying operation that augments the set \(S\) with the element pointed to by \(x\),
- **delete(S, x)**: a modifying operation that, given a pointer \(x\) to an element in the set \(S\), removes the element from \(S\).

2.6.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the `delete` operation is prespecified.

In a *stack*, the element deleted from the set is the one most recently inserted: the stack implements a *last-in first-out* policy. The operation `insert` on a stack is called `push` (Algorithm `Push(S, x)`), and the operation `delete`, which does not take an element argument, is called `pop` (Algorithm `Pop(S)`). We can implement a stack of at most \(n\) elements with an array \(S[0 \ldots n - 1]\). The array has an attribute \(S.top\) that indexes the most recently inserted element. The stack consists of elements \(S[0 \ldots S.top]\), where \(S[0]\) is the element at the bottom of the stack, and \(S[S.top]\) is the element at the top. Operations `empty` and `is-empty` on a stack are called `empty-stack` and `is-stack-empty` (Algorithm `Is-Stack-Empty(S)`), respectively. If an empty stack is popped, we say that stack *underflows*. If \(S.top\)
exceeds \( n - 1 \) (Algorithm \textsc{Is-Stack-Full}(S)), the stack overflows. Each of the stack operations require constant time.

**Algorithm 1: Is-Stack-Empty**

Input : Stack \( S \)

1. if \( S.top = -1 \) then
2.      return true
3. else
4.      return false

**Algorithm 2: Is-Stack-Full**

Input: Stack \( S \)

1. if \( S.top = n - 1 \) then
2.      return true
3. else
4.      return false

**Algorithm 3: Push**

Input: Stack \( S \) and element \( x \) to be pushed

1. if Is-Stack-Full(\( S \)) then
2.      return overflow
3. else
4.      \( S.top \leftarrow S.top + 1 \)
5.      \( S[S.top] \leftarrow x \)

Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in first-out policy. The queue has a head and a tail. When an element is enqueued, it takes its place at the tail of the queue. The element dequeued is always the one at the head of the queue. The operation insert on a queue is called enqueue (Algorithm \textsc{Enqueue}(Q, x)), and the operation delete, which does not take an element argument, is called dequeue (Algorithm \textsc{Dequeue}(Q)). We can implement a queue of at most \( n - 1 \) elements with an array \( Q[0..n-1] \). The queue has an attribute \( Q.head \) that indexes, or points to, its head. The attribute \( Q.tail \) indexes the next
Algorithm 4: Pop

Input : Stack $S$
Output: The popped element, if stack not empty

1 if Is-Stack-Empty($S$) then
  2 return underflow
else
  4 $S\cdot top \leftarrow S\cdot top - 1$
  5 return $S[S\cdot top + 1]$

The elements in the queue are in positions $Q\cdot head, Q\cdot head + 1, \ldots, Q\cdot tail - 1$, where we wrap-around in the sense that position 0 immediately follows position $n - 1$ in a circular order. When $Q\cdot tail = Q\cdot head$, the queue is empty. Initially, we have $Q\cdot tail = Q\cdot head = 0$. Operations empty and is-empty on a queue are called empty-queue and is-queue-empty (Algorithm Is-Queue-Empty($Q$)), respectively. When the queue is empty, an attempt to dequeue an element causes the queue to underflow. When $Q\cdot head = Q\cdot tail + 1$, the queue is full (Algorithm Is-Queue-Full($Q$)), and an attempt to enqueue an element causes the queue to overflow. The pseudocode assumes that $n = Q\cdot length$. Each of the queue operations requires constant time.

Algorithm 5: Is-Queue-Empty

Input : Queue $Q$

1 if $Q\cdot tail = Q\cdot head$ then
  2 return true
else
  4 return false

Algorithm 6: Is-Queue-Full

Input : Queue $Q$

1 if $Q\cdot head = Q\cdot tail + 1$ then
  2 return true
else
  4 return false
Algorithm 7: Enqueue

\begin{algorithm}
\textbf{Input} : Queue $Q$, element $x$ to be enqueued
\begin{algorithmic}[1]
\STATE if Is-Queue-Full($Q$, $x$) then
  \STATE return overflow
\ELSE
  \STATE $Q[Q.tail] \leftarrow x$
  \IF{$Q.tail = Q.length - 1$}
    \STATE $Q.tail \leftarrow 0$
  \ELSE
    \STATE $Q.tail \leftarrow Q.tail + 1$
  \ENDIF
\ENDIF
\end{algorithmic}
\end{algorithm}

Algorithm 8: Dequeue

\begin{algorithm}
\textbf{Input} : Queue $Q$
\begin{algorithmic}[1]
\STATE if Is-Queue-Empty($Q$) then
  \STATE return underflow
\ELSE
  \STATE $x \leftarrow Q[Q.head]$
  \IF{$Q.head = Q.length - 1$}
    \STATE $Q.head \leftarrow 0$
  \ELSE
    \STATE $Q.head \leftarrow Q.head + 1$
  \ENDIF
  \STATE return $x$
\ENDIF
\end{algorithmic}
\end{algorithm}

2.6.2 Linked lists

A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by the pointer in each object. The $i$th element of a list $L$ is denoted by $L(i)$. Each element of a doubly-linked list is an object, whose attributes are a tuple $(a_0, a_1, \ldots, a_n)$, such that $n \geq 0$, and two other attributes: \texttt{next} and \texttt{prev}. Given an element $x$ in the list, $x.$next points to its successor in the linked list, and $x.$prev points to its predecessor. If $x.$prev = nil, the element $x$ has no predecessor and is therefore the first element, or \texttt{head}, of the list. If $x.$next = nil, the element $x$ has no successor and is therefore the last element, or \texttt{tail}, of the list. If a list is singly-linked (see Fig. 2.2 in this regard), we omit the \texttt{prev} pointer in each element. An attribute $L.$head points to the first
The operation search on a list is called list-search (Algorithm List-Search). It finds element with key $k$, where $k$ is an element, or a subset of elements, of tuple $(a_0, a_1, \ldots, a_n)$, in list $L$ by a simple linear search, returning a pointer to this element. If no object with key $k$ appears in the list, then nil is returned. It requires linear time.

**Algorithm 9: List-Search**

**Input**: List $L$, element $k$

1. $x \leftarrow L.\text{head}$
2. while $x \neq \text{nil} \text{ and } x.\text{key} \neq k$ do
   1. $x \leftarrow x.\text{next}$
3. return $x$

The operation insert on a list is called list-insert (Algorithm List-Insert). It inserts element $x$ to the front of list $L$. It requires constant time.

**Algorithm 10: List-Insert**

**Input**: List $L$, element $x$ to be inserted

1. $x.\text{next} \leftarrow L.\text{head}$
2. if $L.\text{head} \neq \text{nil}$ then
   1. $L.\text{head}.\text{prev} \leftarrow x$
3. $L.\text{head} \leftarrow x$
4. $x.\text{prev} \leftarrow \text{nil}$

The operation delete on a list is called list-delete (Algorithm List-Delete($L, x$)). It must be given a pointer to $x$, and it then deletes $x$ from list $L$ by updating pointers. It requires constant time.

Operations empty and is-empty on a linked list are called empty-list and is-empty-list, respectively.
Algorithm 11: List-Delete

**Input**: List $L$, element $x$ to be deleted

1. if $x$.prev $\neq$ nil then
2.   $x$.prev.next $\leftarrow$ $x$.next
3. else
4.   $L$.head $\leftarrow$ $x$.next
5. if $x$.next $\neq$ nil then
6.   $x$.next.prev $\leftarrow$ $x$, prev

2.6.3 Tries

A trie (also called prefix tree) is an ordered tree data structure that is used to store an associative array, where the keys are usually strings.

Let $X = \{x_1, x_2, \ldots, x_m\}$ be a set of pairwise distinct strings. Then a trie (also called prefix tree) is a search tree containing exactly $m + 1$ leaves, one for each $x_i$, $1 \leq i \leq m$ and one for the empty string $\epsilon$.

The edges of a trie are labeled with the symbols that occur in the strings of $x$ and a special end-of-string sentinel, conventionally denoted by $. The path formed by the edges starting from the root node of the trie and terminating at the leaf for a string $x_i$, $1 \leq i \leq m$, spell out $x_i$ in that order, followed by the sentinel $. A characteristic property of this data structure is that common prefixes appear only once. Thus the strings represented by the nodes in a trie are constrained to be pairwise distinct.

The operation insert on a trie is called trie-insert (see Algorithm TRIE-INSERT). It inserts element pair $((Key, Val))$ to the trie $T$ by constructing (in case it does not exist) a path of states $v_0, v_1, \ldots, v_{|Key|}$ and transitions $e_1, e_2, \ldots, e_{|Key|}$, $1 \leq i \leq |Key|$, where $e_i = (u_{i-1}, u_i)$, $u_0$ is the root node, spelling out $Key$, and saves the value $Val$ in a new node $u$ by creating a sentinel ($) labeled edge $(v_{|Key|}, u)$. It requires linear time to the size of $Key$ in order for the operation to be carried out.

The operation search on a trie is called trie-search (Algorithm TRIE-SEARCH). It returns the value associated with key $Key$ by following the path of transitions $e_1, e_2, \ldots, e_{|Key|}$, that originate at the root node, spell out $Key$ and lead to node $v$. Then the operation returns the value of the node $u$ which is connected to node $v$ via a sentinel ($) edge. In case the path of edges or the sentinel edge do not exist, the operation returns nil as an indication of failure.

Another operation is delete, which on a trie is called trie-delete and is presented in Algorithm TRIE-DELETE$(T, Key)$. The method deletes the value associated with key $Key$ in trie $T$, and from the path of nodes $v_0, v_1, \ldots, v_{|Key|}$ spelling out
Algorithm 12: Trie-Insert

Input : Trie $T$, key $Key$ and value $Val$ to be inserted

1. if Is-Trie-Empty($T$) then $T$.root ← Create-New-Node()
2. $v ← T$.root
3. for $i ← 1$ to $|Key|$ do
   4. if not exists $v^{Key[i]}{\rightarrow}u$ then Create-Edge($v^{Key[i]}{\rightarrow}u$)
   5. $v ← u$
   6. $u ←$ Create-New-Node()
   7. $u$.value ← $Val$
   8. Create-Edge($v^{\varepsilon}{\rightarrow}u$)

Algorithm 13: Trie-Search

Input : Trie $T$, key $Key$

1. if Is-Trie-Empty($T$) then return nil
2. $v ← T$.root
3. for $i ← 1$ to $|Key|$ do
   4. if not exists $v^{Key[i]}{\rightarrow}u$ then return nil
   5. $v ← u$
6. if not exists $v^{\varepsilon}{\rightarrow}u$ then return nil
7. return $u$.Value

Key, where $v_0$ is the root node, deletes nodes $v_i, v_{i+1}, \ldots, v_{|Key|}$, $0 \leq i \leq |Key|$, which spell out a suffix of $Key$ and have out-degree 0.

The last operation is is-empty and on a trie is called is-empty-trie. It is presented in Algorithm Is-Trie-Empty.

A special case of a trie that deserves mentioning is the suffix trie. Given a string $x$, a suffix trie is a trie constructed over the set $Suff(x)$ of suffixes of $x$. Another variation of the classical trie is the Patricia trie (or compacted trie) [92]. A Patricia trie is constructed from a trie by eliminating all internal nodes of degree 2 (those with a parent and just a single child), thus forming edges that spell out a substring rather than a single letter.
Algorithm 14: Trie-Delete

Input: Trie $T$, key $Key$

1. if Is-Trie-Empty($T$) then return false
2. $v \leftarrow T.root$
3. $S \leftarrow$ New-Stack
4. Push($S, v$)
5. for $i \leftarrow 1$ to $|Key|$ do
   6. if not exists $v \xrightarrow{K_{key[i]}} u$ then return false
   7. $v \leftarrow u$
   8. Push($S, v$)
9. if not exists $v \xrightarrow{S} u$ then return false
10. Delete-Node($u$)
11. while not exists Is-Stack-Full($S$) do
12.   $v \leftarrow$ Pop($S$)
13.   if Out-Degree($v) \neq 0$ then return true
14.   Delete-Node($v$)
15. return true

Algorithm 15: Is-Trie-Empty

Input: Trie $T$

1. if $T.root = nil$ then
   2. return true
3. else
4. return false
Chapter 3

Linear notations of tree structures

“A mathematician is a device for turning coffee into theorems”
— Paul Erdős (Hungarian mathematician, 1913-1996)

Every sequential algorithm traverses a processed tree structure in a sequential order of nodes, which forms a corresponding linear notation of the tree structure. We consider depth first oriented traversal of the processed tree in which every node is recorded just once during some visit.

In the rest of the text we will cover two types of tree structures — ranked trees and unranked trees. In the corresponding sections we will define the linear notations that can be used to describe the respective tree structure. We will also prove some basic properties of these notations and give algorithms for manipulating with them, which will be used in subsequent chapters.

3.1 Ranked trees

Ranked trees are trees where each node is drawn from a ranked alphabet and as such, the rank of each node, i.e. the number of children, is known beforehand. In this section we formally define two linear notations for describing ranked trees, present their properties and give some necessary algorithms which will be used in chapters 4, 5 and 6.
3.1.1 Definitions and properties

There are two standard one-visit depth-first oriented linear notations of ranked trees: The prefix (also called preorder) notation, and postfix (also called postorder) notation.

![Figure 3.1: Tree a_2(a_2(a_0, a_1(a_0)), a_1(a_0)) in (a) and its subtrees (b)](image)

**Definition 52 (Prefix notation)** The prefix notation $\text{pref}(t)$ of a tree $t$ is recursively defined as follows:

1. $\text{pref}(t) = a$ if $|t| = 1$ and $a$ is the label of its only node.
2. $\text{pref}(t) = a \text{pref}(t_1) \text{pref}(t_2) \ldots \text{pref}(t_n)$, where $a$ is the root and $t_1, t_2, \ldots, t_n$ are direct descendants (subtrees) of $a$.

**Definition 53 (Postfix notation)** The postfix notation $\text{post}(t)$ of a tree $t$ is recursively defined as follows:

1. $\text{post}(t) = a$ if $|t| = 1$ and $a$ is the label of its only node.
2. $\text{post}(t) = \text{post}(t_1) \text{post}(t_2) \ldots \text{post}(t_n) a$, where $a$ is the root and $t_1, t_2, \ldots, t_n$ are direct descendants (subtrees) of $a$.

**Example 3 (Ranked tree Notations)** Let $t$ be the tree with parenthesis notation $a_2(a_2(a_0, a_1(a_0)), a_1(a_0))$ over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$, where $\Sigma = \{a_0, a_1, a_2\}$ and $\varphi(a_0) = 0$, $\varphi(a_1) = 1$, $\varphi(a_2) = 2$. The tree is illustrated in Figure 3.1. The prefix notation of $t$, according to Definition 52 is $\text{pref}(t) = a_2a_2a_0a_1a_0a_1a_0$. The postfix notation, according to Definition 53, is $\text{post}(t) = a_0a_0a_1a_2a_0a_1a_2$.  

43
Lemma 2  The prefix notations of all subtrees of some tree $t$ over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$ are factors of the prefix notation $\text{pref}(t)$ of $t$.

Proof  By induction on the height of the subtree.

1. In case a tree $t'$ consists of only one node $a$, that is $|t'| = 1$, $h(t') = 0$ and $\varphi(a) = 0$, then $\text{pref}(t') = a$ and the claim holds.

2. Assume the claim holds for trees $t_1, t_2, \ldots, t_p$, where $p \geq 1$ and $h(t_i) \leq m$, $1 \leq i \leq p$, $m \geq 0$. It must be proved that the claim holds also for tree $t' = a(t_1, t_2, \ldots, t_p)$, where $\varphi(a) = p$ and $h(t') = m + 1$. Since $\text{pref}(t') = a \text{pref}(t_1) \text{pref}(t_2) \ldots \text{pref}(t_p)$, the claim holds for tree $t'$.

Thus the theorem holds. \hfill $\square$

However, not every factor of the prefix notation of a tree represents a subtree. This is obvious due to the fact that there can be $O(n^2)$ distinct factors of a given prefix notation of some tree with $n$ nodes, but the tree consists of only $n$ subtrees – each node of the tree is the root of one subtree. Only the factors which themselves are trees in prefix notation represent subtrees. This property is formalised by the following definition and theorem.

Definition 54  Let $x[1..m], m \geq 1$, be a string over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$. Then, the arity checksum $ac(x) = \varphi(x[1]) + \varphi(x[2]) + \ldots + \varphi(x[m]) - m + 1 = \sum_{i=1}^{m} \varphi(x[i]) - m + 1$.

Theorem 1  Let $x[1..|t|]$ be the prefix notation of a tree $t$ over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$, and $y$ a factor of $x$. Then, $y$ is the prefix notation of a subtree $t'$ of $t$, if and only if $ac(y) = 0$, and $ac(w) \geq 1$ for each $w$, where $y = wz$, $w, z \in \Sigma^+$.

Proof  For any two subtrees $t_1$ and $t_2$ such that $t_1 \neq t_2$, it holds that $\text{pref}(t_1)$ and $\text{pref}(t_2)$ are either two different strings or one is a substring of the other. The former case occurs if the subtrees $t_1$ and $t_2$ are two different trees with no shared part, and the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in ranked ordered trees. Moreover, for any two adjacent subtrees, it holds that their prefix notations are two adjacent substrings.

- If:  By induction on the height of tree $t'$, where $y = \text{pref}(t')$:

  1. Assume the height of $t'$ is 0, i.e. $h(t') = 0$, which means we consider the case $y = a$, $a \in \Sigma$, where $\varphi(a) = 0$. Then, $ac(y) = 0$. Thus, the claim holds for the case where $h(t') = 0$. 

44
2. Assume that the claim holds for trees \( t_1, t_2, \ldots, t_p, \) where \( p \geq 1, \) 
\( h(t_i) \leq m \) and \( ac(pref(t_i)) = 0, \) for all \( 1 \leq i \leq p. \) We must prove 
that the claim also holds for any tree \( t', \) such that \( h(t') = m + 1: \) 
Assume that the prefix notation of \( t' \) is \( y = prep(t_1) prep(t_2) \ldots prep(t_p), \)
where \( a \in \Sigma \) and \( \varphi(a) = p. \) Then \( ac(y) = p + ac(pref(t_1)) + ac(pref(t_2)) + \) 
\( \ldots + ac(pref(t_p)) - (p + 1) + 1 = 0 \) and \( ac(w) \geq 1 \) for each \( w, \) where 
\( y = wz, w, z \in \Sigma^+. \) Thus, the claim holds for the case \( h(t') = m + 1. \)

- **Only if:** Assume \( ac(y) = 0, \) and \( |y| = k, \) where \( k \geq 1 \) and \( \varphi(y[1]) = p. \) Since 
\( ac(w) \geq 1 \) for each \( w, \) where \( y = wz, w, z \in \Sigma^+, \) none of the substrings \( w \)
can be a subtree in prefix notation. This means that the only possibility for 
\( ac(y) = 0 \) is that \( y \) is of the form \( y = prep(t_1) prep(t_2) \ldots prep(t_p), \)
where \( p \geq 0, \) and \( t_1, t_2, \ldots, t_p \) are adjacent subtrees. In such case, \( ac(y) = \) 
\( p + 0 - (p + 1) + 1 = 0. \) No other possibility of the form of \( y \) for \( ac(y) = 0 \)
is possible. Thus, the claim holds.

Thus the theorem holds.

**Theorem 2** Let \( x[1 \ldots |t|] \) be the postfix notation of a tree \( t \) over a ranked alphabet \( A = (\Sigma, \varphi), \) and \( y \) a factor of \( x. \) Then, \( y \) is the postfix notation of a subtree \( t' \) of 
\( t, \) if and only if \( ac(y) = 0, \) and \( ac(w) \geq 1 \) for each \( w, \) where \( y = zw, w, z \in \Sigma^+. \)

**Proof** For any two subtrees \( t_1 \) and \( t_2 \) such that \( t_1 \neq t_2, \) it holds that \( post(t_1) \) 
and \( post(t_2) \) are either two different strings or one is a substring of the other. 
The former case occurs if the subtrees \( t_1 \) and \( t_2 \) are two different trees with no
shared part and the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in ranked ordered trees. Moreover, for any two adjacent subtrees it holds that their postfix notations are two adjacent substrings.

• If: By induction on the height of subtree $t'$, where $y = \text{post}(t')$:

1. Assume the height of $t'$ is 0, i.e. $h(t') = 0$, which means we consider the case $y = a$, $a \in \Sigma$, where $\varphi(a) = 0$. Then $ac(y) = 0$. Thus, the claim holds for the case where $h(t') = 0$.

2. Assume that the claim holds for trees $t_1, t_2, \ldots, t_p$, where $p \geq 1$, $h(t_i) \leq m$ and $ac(\text{post}(t_i)) = 0$, for all $1 \leq i \leq p$. We must prove that the claim also holds for any tree $t'$, such that $h(t') = m + 1$.

  Assume the postfix notation of $t'$ is $y = \text{post}(t_1)\text{post}(t_2)\ldots\text{post}(t_p)a$, where $a \in \Sigma$ and $\varphi(a) = p$. Then $ac(y) = ac(\text{post}(t_1)) + ac(\text{post}(t_2)) + \ldots + ac(\text{post}(t_p)) + p - (p + 1) + 1 = 0$ and $ac(w) \geq 1$ for each $w$, where $y = zw$, $w, z \in \Sigma^+$. Thus, the claim holds for the case $h(t') = m + 1$.

• Only if: Assume $ac(y) = 0$, and $|y| = k$, where $k \geq 1$ and $\varphi(y[k]) = p$. Since $ac(w) \geq 1$ for each $w$, where $y = zw$, $w, z \in \Sigma^+$, none of the substrings $w$ can be a subtree in postfix notation. This means that the only possibility for $ac(y) = 0$ is that $y$ is of the form $y = \text{post}(t_1)\text{post}(t_2)\ldots\text{post}(t_p)a$, where $p \geq 0$, and $t_1, t_2, \ldots, t_p$ are adjacent subtrees. In such case, $ac(y) = p + 0 - (p + 1) + 1 = 0$. No other possibility of the form of $y$ for $ac(y) = 0$ is possible. Thus, the claim holds.

\[\square\]

Note that the prefix notation and the postfix notation are not the only linear notations of ranked tree structures. There exist more linear tree representations, for instance, the euler notation is used in [95] for nonlinear tree pattern matching.

### 3.1.2 Algorithms for working with ranked linear notations

We will now present some basic algorithms that compute various properties of a tree when given its linear notation as input. These algorithms will be used throughout the rest of this thesis.

The first algorithm we will deal with is Algorithm 16, which computes the size (i.e. number of nodes) of each subtree of a given tree $t$. Specifically, the algorithm takes as input a string $x[1..|t|]$ representing the postfix notation of $t$
Algorithm 16: Subtree-Size-Array-Postfix

Input: The postfix notation \(post(t) = x[1 \ldots n]\)

Output: Array \(V\) where element \(i\) denotes the size of subtree rooted at \(x[i]\)

1. for \(i \leftarrow 1\) to \(n\) do
2. \(V[i] \leftarrow 1\)
3. if \(\varphi(x[i]) \neq 0\) then
4. for \(j \leftarrow 1\) to \(\varphi(x[i])\) do
5. \(V[i] \leftarrow V[i] + V[i - V[i]]\)

and outputs an array \(V\) of \(|t|\) elements, such that the element \(V[i]\), for \(1 \leq i \leq |t|\), is the number of nodes of the subtree of \(t\) whose root is the node \(x[i]\).

Algorithm 16 reads a string \(x\) from left to right, which actually corresponds to a postorder traversal of the tree structure, and at each step (i.e. at each visited node) the size of the subtree rooted at that node is computed by appending the size of the subtrees rooted at each of its children nodes. The algorithm is straightforward and computes the array \(V\) in time linear to the size of the postfix notation of the tree structure without the need of additional storage space.

Given the postfix notation \(x\) of a tree \(t\), another interesting problem is how to compute the parent of each node of \(t\) represented by the symbols of \(x\). Algorithm 17 solves this problem by utilising a stack, and returns an array \(P\) of size \(|t|\) such that the element \(P[i]\), for all \(1 \leq i < n\), is the index of the parent of \(x[i]\). In other words, \(x[P[i]]\) is the parent of \(x[i]\). The element \(P[|t|]\) is undefined, since the root node has no parent.

Algorithm 17: Node-Parents-Array-Postfix

Input: The postfix notation \(post(t) = x[1 \ldots n]\)

Output: Array \(P\) where element \(i\) denotes the parent of \(x[i]\)

1. \(R \leftarrow \text{New-Stack}\)
2. for \(i \leftarrow 1\) to \(n\) do
3. for \(j \leftarrow 1\) to \(\varphi(x[i])\) do
4. \(r \leftarrow \text{POP}(R)\)
5. \(P[r] \leftarrow i\)
6. \(\text{PUSH}(R, i)\)

The corresponding algorithm to Node-Parents-Array-Postfix for the prefix notation is presented in Algorithm 18, which again uses a stack structure in a similar way as Algorithm 17.
Algorithm 18: Node-Parents-Array-Prefix

Input: The prefix notation $\text{pref}(t) = x[1..n]$
Output: Array $P$ where element $i$ denotes the parent of $x[i]$

1. $R \leftarrow \text{NEW-STACK}$
2. $\text{PUSH}(R, 0)$
3. for $i \leftarrow 1$ to $n$ do
   4. $P[i] \leftarrow \text{POP}(R)$
   5. for $j \leftarrow 1$ to $\varphi(x[i])$ do $\text{PUSH}(R, i)$

Algorithm 19 takes as input the postfix notation $x$ of some tree $t$, and computes an array $H$, where each element $H[i]$, for all $1 \leq i \leq |t|$, denotes the height of the subtree whose root is the node $x[i]$. The algorithm uses a stack for the computation and reads $x$ from left to right, i.e. a postorder traversal of the tree. At each visited node $x[i]$, it stores 0 in $H[i]$ in case the node is a leaf, or pops $m$ numbers from the stack, where $m$ is the arity of $x[i]$, and stores the largest from those numbers in $H[i]$. Each time the height of a node is computed, it is pushed on top of the stack.

Algorithm 19: Subtree-Height-Array-Postfix

Input: The postfix notation $\text{post}(t) = x[1..n]$
Output: Array $H$ where element $i$ is the height of subtree rooted at $x[i]$

1. $R \leftarrow \text{NEW-STACK}$
2. for $i \leftarrow 1$ to $n$ do
   3. if $\varphi(x[i]) = 0$ then
      4. $\text{PUSH}(R, 0)$
      5. $H[i] \leftarrow 0$
   else
      6. $r \leftarrow 0$
      7. for $j \leftarrow 1$ to $\varphi(x[i])$ do
         8. $r \leftarrow \max(r, \text{POP}(R))$
      9. $H[i] \leftarrow r + 1$
     10. $\text{PUSH}(R, r + 1)$

The next two algorithms deal with the transformation between the postfix and prefix notation. Given a string $x$ that is the postfix notation of some tree $t$, Algorithm 20 outputs a string $y$ that is the prefix notation of $t$, and vice versa in Algorithm 21.
Algorithm 20: Postfix-To-Prefix-Transform

**Input**: The postfix notation \( post(t) = x[1 \ldots n] \) of \( t \)

**Output**: The prefix notation \( pref(t) = y[1 \ldots n] \) of \( t \)

1. \( P \leftarrow \text{NODE-PARENTS-ARRAY-POSTFIX}(x) \)
2. \( \triangleright \) Initialise queues and a stack
3. for \( i \leftarrow 1 \) to \( n \) do
4. \hspace{1em} \( Q[i] \leftarrow \text{NEW-QUEUE} \)
5. \( R \leftarrow \text{NEW-STACK} \)
6. \( \triangleright \) Construct queues containing the children of each node
7. for \( i \leftarrow 1 \) to \( n \) do
8. \hspace{1em} \( \text{ENQUEUE}(Q[P[i]], i) \)
9. \( \triangleright \) Transform postfix notation to prefix notation
10. \( y \leftarrow x[n] \)
11. for \( i \leftarrow 1 \) to \( \varphi(x[n]) \) do \( \text{PUSH}(R, i) \)
12. while not \( \text{IS-EMPTY-STACK}(R) \) do
13. \hspace{1em} \( k \leftarrow \text{POP}(R) \)
14. \hspace{1em} \( l \leftarrow \text{DEQUEUE}(Q[k]) \)
15. \hspace{1em} \( y \leftarrow y \cdot x[l] \)
16. \hspace{1em} for \( i \leftarrow 1 \) to \( \varphi(x[l]) \) do \( \text{PUSH}(R, l) \)

Algorithm 21: Prefix-To-Postfix-Transform

**Input**: The prefix notation \( pref(t) = x[1 \ldots n] \) of \( t \)

**Output**: The postfix notation \( post(t) = y[1 \ldots n] \) of \( t \)

1. \( P \leftarrow \text{NODE-PARENTS-ARRAY-PREFIX}(x) \)
2. for \( i \leftarrow 1 \) to \( n \) do
3. \hspace{1em} \( V[i] \leftarrow \varphi(x[i]) \)
4. \hspace{1em} \( j \leftarrow i \)
5. while \( V[j] = 0 \) do
6. \hspace{1em} \( y \leftarrow y \cdot x[j] \)
7. \hspace{1em} \( j \leftarrow P[j] \)
8. \hspace{1em} \( V[j] \leftarrow V[j] - 1 \)

### 3.2 Unranked trees

Unranked trees are trees where each node is drawn from an alphabet, and the rank of each node is not known beforehand. Of course, an unranked tree can be
easily transformed to a ranked tree using a method which will be described in section 3.2.2. We will however, present some interesting features and properties of the linear notations of unranked trees and give equivalent algorithms to the ones for ranked trees, without the need of transforming an unranked linear notation to a ranked one.

Similarly as in Section 3.1, we will first introduce the linear notations, present some of the properties of strings representing unranked trees and then give some algorithms which will be used through this thesis.

### 3.2.1 Definitions and properties

Unranked trees, that is trees consisting of nodes whose rank is not known beforehand, can also be represented by a specific form of linear notation. We call this form the bar notation, and as in the case of ranked trees, a bar notation can be constructed from the preorder traversal of the tree, in which case it is called prefix bar notation or from the postorder traversal of the tree, in which case it is called the postfix bar notation.

**Definition 55 (Prefix bar notation)** The prefix bar notation preb(t) of a tree t is defined as follows:

1. \( \text{preb}(t) = a \uparrow \) if \( |t| = 1 \) and a is the label of its only node.

2. \( \text{preb}(t) = a \text{preb}(t_1) \text{preb}(t_2) \ldots \text{preb}(t_n) \uparrow \), where a is the root of tree t and \( t_1, t_2, \ldots, t_n \) are direct descendants (subtrees) of a.

**Definition 56 (Postfix bar notation)** The postfix bar notation pstb(t) of a tree t is defined as follows:

1. \( \text{pstb}(t) = \uparrow a \) if \( |t| = 1 \) and a is the label of its only node.

2. \( \text{pstb}(t) = \uparrow \text{pstb}(t_1) \text{pstb}(t_2) \ldots \text{pstb}(t_n) a \), where a is the root and \( t_1, t_2, \ldots, t_n \) are direct descendants (subtrees) of a.

**Example 4 (Unranked tree notations)** Let t be the tree with parenthesis notation \( a_2(a_2(a_0, a_1(a_0)), a_1(a_0)) \) from Figure 3.1. Assume the tree is not constructed from a ranked alphabet and therefore the rank of each node is unknown. By a simple traversal however, one can construct the postfix and prefix bar notations of t. According to Definition 55 and 56, the prefix bar preb(t) and postfix bar pstb(t) notations of t are \( a_2a_2a_0 \uparrow a_1a_0 \uparrow \uparrow \uparrow a_1a_0 \uparrow \uparrow \uparrow \uparrow \uparrow a_0 \uparrow \uparrow a_0a_1a_2 \uparrow \uparrow a_0a_1a_2 \), respectively.

**Lemma 4** The prefix bar notations of the subtrees of a tree t over an alphabet \( \Sigma \) are factors of the prefix bar notation preb(t) of t.
Proof By induction on the height of the subtree:

- Let \( t' \) be a tree of height 0 (i.e. it consists of only one node \( a \). The only subtree is the tree itself and thus the claim holds for height 0.

- Assume the claim holds for trees \( t_1, t_2, \ldots, t_p \), where \( p \geq 1 \), \( h(t_i) \leq m \), \( 1 \leq i \leq p \), \( m \geq 0 \). It must be proved that the claim also holds for any tree \( t' \) such that \( h(t') = m + 1 \):
  
  Let \( t' \) be a tree of height \( m + 1 \) which consists of a root node \( a \) with \( p \) direct subtrees \( t_1, t_2, \ldots, t_p \). Since \( \text{preb}(t') = \text{preb}(t_1) \text{preb}(t_2) \ldots \text{preb}(t_p) \upsilon \), the claim holds for tree \( t' \). 

\( \square \)

Definition 57 Let \( x[1..m] \), \( m \geq 1 \), be a string over an alphabet \( \Sigma \cup \{ \upsilon \} \). Then, the bar checksum is \( bc(x) = \sum_{i=1}^{m} b(x[i]) \), where

\[
    b(x[i]) = \begin{cases} 
        1 : & x[i] = \upsilon \\
        -1 : & x[i] \in \Sigma 
    \end{cases}
\]

Theorem 3 Let \( x[1..2|t|] \) be the prefix bar notation \( \text{preb}(t) \) of \( t \) over an alphabet \( \Sigma \cup \{ \upsilon \} \), and \( y \) a factor of \( x \). Then \( y \) is the prefix bar notation of a subtree \( t' \) of \( t \), if and only if \( bc(y) = 0 \), and \( bc(w) < 0 \) for each \( w \), where \( y = wz \), \( w, z \in (\Sigma \cup \{ \upsilon \})^+ \).

Proof For any two subtrees \( t_1 \) and \( t_2 \) such that \( t_1 \neq t_2 \), it holds that \( \text{preb}(t_1) \) and \( \text{preb}(t_2) \) are either two different strings or one is a factor of the other. The former case occurs if the subtrees \( t_1 \) and \( t_2 \) are two different trees with no shared part, while the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in unranked ordered trees. Moreover, for any two adjacent subtrees it holds that their bar notations are two adjacent factors.

- If: By induction on the height of subtree \( t' \), where \( y = \text{preb}(t) \):
  
  1. Assume the height of \( t' \) is 0, i.e. \( h(t') = 0 \), which means we consider the case \( y = a \upsilon \), \( a \in \Sigma \). Then \( bc(y) = 0 \). Thus, the claim holds for the case where \( h(t') = 0 \).

  2. Assume that the claim holds for trees \( t_1, t_2, \ldots, t_p \), where \( p \geq 1 \), \( h(t_i) \leq m \) and \( bc(\text{preb}(t_i)) = 0 \), for all \( 1 \leq i \leq p \). We must prove that the claim also holds for any tree \( t' \), such that \( h(t') = m + 1 \):
    Assume \( y = \text{preb}(t_1) \text{preb}(t_2) \ldots \text{preb}(t_p) a \upsilon \), where \( a \in \Sigma \), is the prefix bar notation of \( t' \). Then \( bc(y) = bc(\text{preb}(t_1) + bc(\text{preb}(t_2) + \ldots + bc(\text{preb}(t_p))) - 1 + 1 = 0 \), and \( bc(w) < 0 \) for each \( w \), where \( y = wz \), \( w, z \in (\Sigma \cup \{ \upsilon \})^+ \). Thus, the claim holds for the case \( h(t') = m + 1 \).
• Only if: Assume $bc(y) = 0$, and $|y| = k$, where $k \geq 1$. Since $bc(w) < 0$ for each $w$, where $y = wz$, $w, z \in (\Sigma \cup \{\uparrow\})^+$, none of the substrings $w$ can be a subtree in prefix bar notation. This means that the only possibility for $bc(y) = 0$ is that $y$ is of the form $y = preb(t_1) preb(t_2) \ldots preb(t_p) \uparrow$, where $p \geq 0$, and $t_1, t_2, \ldots, t_p$ are adjacent subtrees. In such case, $bc(y) = 0 - 1 + 1 = 0$. No other possibility of the form of $y$ for $bc(y) = 0$ is possible. Thus, the claim holds.

Analogously to Lemma 4 and Theorem 3 for the prefix bar notation, we prove the following for the postfix bar notation.

**Lemma 5** The postfix bar notations of the subtrees of a tree $t$ over an alphabet $\Sigma$ are factors of the postfix bar notation $pstb(t)$ of $t$.

**Proof** By induction on the height of the subtree:

- Let $t'$ be a tree of height 0 (i.e. it consists of only one node $a$. The only subtree is the tree itself and thus the claim holds for height 0.

- Assume the claim holds for trees $t_1, t_2, \ldots, t_p$, where $p \geq 1$, $h(t_i) \leq m$, $1 \leq i \leq p$, $m \geq 0$. It must be proved that the claim also holds for any tree $t'$ such that $h(t') = m + 1$:

  Let $t'$ be a tree of height $m + 1$ which consists of a root node $a$ with $p$ direct subtrees $t_1, t_2, \ldots, t_p$. Since $pstb(t') = \uparrow pstb(t_1) pstb(t_2) \ldots pstb(t_p) v$, the claim holds for tree $t'$.

Theorem 4 Let $x[1 \ldots |t|]$ be the postfix bar notation $pstb(t)$ of $t$ over an alphabet $\Sigma \cup \{\uparrow\}$, and $y$ a factor of $x$. Then $y$ is the postfix bar notation of a subtree $t'$ of $t$, if and only if $bc(y) = 0$, and $bc(w) < 0$ for each $w$, where $y = zw$, $w, z \in (\Sigma \cup \{\uparrow\})^+$.

**Proof** For any two subtrees $t_1$ and $t_2$ such that $t_1 \neq t_2$, it holds that $pstb(t_1)$ and $pstb(t_2)$ are either two different strings or one is a factor of the other. The former case occurs if the subtrees $t_1$ and $t_2$ are two different trees with no shared part, while the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in unranked ordered trees. Moreover, for any two adjacent subtrees it holds that their bar notations are two adjacent factors.

- If: By induction on the height of subtree $t'$, where $y = pstb(t)$:
1. Assume the height of $t'$ is 0, i.e. $h(t') = 0$, which means we consider the case $y = \uparrow a, a \in \Sigma$. Then $bc(y) = 0$. Thus, the claim holds for the case where $h(t') = 0$.

2. Assume that the claim holds for trees $t_1, t_2, \ldots, t_p$, where $p \geq 1$, $h(t_i) \leq m$ and $bc(pstb(t_i)) = 0$, for all $1 \leq i \leq p$. We must prove that the claim also holds for any tree $t'$, such that $h(t') = m + 1$:

Assume $y = \uparrow pstb(t_1) pstb(t_2) \ldots pstb(t_p) a$, where $a \in \Sigma$, is the postfix bar notation of $t'$. Then $bc(y) = 1 + bc(pstb(t_1)) + bc(pstb(t_2)) + \ldots + bc(pstb(t_p)) - 1 = 0$, and $bc(w) < 0$ for each $w$, where $y = zw, w, z \in (\Sigma \cup \{\uparrow}\}^+)$. Thus, the claim holds for the case $h(t') = m + 1$.

• Only if: Assume $bc(y) = 0$, and $|y| = k$, where $k \geq 1$. Since $bc(w) < 0$ for each $w$, where $y = zw, w, z \in (\Sigma \cup \{\uparrow}\}^+$, none of the substrings $w$ can be a subtree in postfix bar notation. This means that the only possibility for $bc(y) = 0$ is that $y$ is of the form $y = \uparrow pstb(t_1) pstb(t_2) \ldots pstb(t_p) a$, where $p \geq 0$, and $t_1, t_2, \ldots, t_p$ are adjacent subtrees. In such case, $bc(y) = 1 + 0 - 1 = 0$. No other possibility of the form of $y$ for $bc(y) = 0$ is possible. Thus, the claim holds.

\[\square\]

### 3.2.2 Algorithms for working with the unranked linear notations

We are now in a position to present analogous algorithms to the ones given in Section 3.1.2, for computing properties of unranked tree structures when their linear notation is given as the input.

The first algorithm we will present computes the postfix notation of a tree given in its postfix bar notation. In other words, Algorithm 22 transforms the postfix bar notation of an unranked tree $t$ to the equivalent postfix notation as if the ranks of nodes of $t$ were known beforehand.

Algorithm 22 utilises a stack to match each node with its corresponding bar by reading the postfix bar notation of a tree $t$ from left to right. In the case a bar is read, it is placed on top of the stack. In the opposite case, that is when a node is read, symbols are popped from the stack until a bar is popped. The number of pops performed corresponds to the number of children (and thus rank) of the node. After assigning the rank of the read node in a ranked alphabet (consisting of the set $\Sigma$ of symbols and the ranking function $\varphi$), a symbol $\circ$ is placed on top of the stack to indicate that a node (and the corresponding subtree rooted at that node) was read. Each read node is tucked at the end of a string $y$ (initialised as
Algorithm 22: Transform-Postfix-Bar-To-Postfix

| Input: The postfix bar notation \( \text{pstb}(t) = x[1 \ldots n] \) of the subject tree \( t \) |
| Output: The postfix notation \( y = \text{post}(t) \) over the ranked alphabet \( \mathcal{A} = (\Sigma, \varphi) \) |

1. \( y \leftarrow \varepsilon \)
2. \( \Sigma \leftarrow \emptyset \)
3. \( R \leftarrow \text{New-Stack} \)
4. \( \triangleright \) Perform the transformation
5. for \( i \leftarrow 1 \) to \( n \) do
6.   if \( x[i] = \uparrow \) then
7.     Push(\( R, \uparrow \))
8.   else
9.     \( \text{arity} \leftarrow 0 \)
10.    \( k \leftarrow \text{Pop}(R) \)
11.    while \( k \neq \uparrow \) do
12.       \( k \leftarrow \text{Pop}(R) \)
13.       \( \text{arity} \leftarrow \text{arity} + 1 \)
14.       Push(\( R, \circ \))
15.       \( \varphi(x[i]) \mapsto \text{arity} \)
16.       \( \Sigma \leftarrow \Sigma \cup \{x[i]\} \)
17.     \( y \leftarrow y \cdot x[i] \)
18. end

an empty string), which will, at the end of the algorithm, represent the postfix notation of \( t \).

Example 5 (Postfix bar to postfix transform) Let \( t \) be the tree with parenthesis notation \( a_2(a_2(a_0, a_1(a_0)), a_1(a_0)) \) from Figure 3.1. Again, assume the tree is not constructed from a ranked alphabet and therefore the rank of each node is unknown. Its postfix bar notation \( \text{pstb}(t) \) is the string \( x = \uparrow \uparrow \uparrow a_0 \uparrow \uparrow a_0 a_1 a_2 \uparrow \uparrow a_0 a_1 a_2 \) as shown in Example 4. The corresponding postfix notation computed by Algorithm 22 is the string \( y = a_0 a_0 a_1 a_2 a_0 a_1 a_2 \).

The next algorithm we will deal with is the computation of the size (i.e. number of nodes) of each subtree of a tree \( t \) when \( t \) is given in its postfix bar notation. Specifically, Algorithm 23 takes as input a string \( x[1 \ldots 2|t|] \) representing the postfix bar notation of a tree \( t \) and outputs an array \( V \) of \( 2|t| \) elements, such that the element \( V[i] \), for \( 1 \leq i \leq 2|t| \), is the number of nodes of the subtree of \( t \) whose root is the node \( x[i] \) in case \( x[i] \) is a node, or 0 in case \( x[i] \) is a bar.
Algorithm 23: SUBTREE-SIZE-ARRAY-POSTFIX-BAR

Input: The postfix bar notation $pstb(t) = x[1\ldots n]$ of a tree $t$
Output: Array $S$ whose elements are the size of each subtree of $t$

1 ⊲ Transform the unranked postfix bar notation to a ranked postfix notation
2 $y, A = (\Sigma, \varphi) \leftarrow$ TRANSFORM-POSTFIX-BAR-TO-POSTFIX
3 ⊲ Compute the size array of the ranked postfix notation
4 $V[1\ldots n/2] \leftarrow$ SUBTREE-SIZE-ARRAY-POSTFIX($y, A$)
5 ⊲ Compute the size array for the unranked postfix bar notation
6 $j \leftarrow 1$
7 $i \leftarrow 1$ to $n$
do
8 if $x[i] = \uparrow$ then
9 $V[i] \leftarrow 0$
10 else
11 $V[i] \leftarrow V[j]$
12 $j \leftarrow j + 1$
end do

Algorithm 23 utilises Algorithms 22 and 16 to transform the postfix bar notation to a postfix (i.e. ranked) notation, computes the array $V$ of subtree sizes for the ranked postfix notation, and then, because of the fact that the postfix notation is equivalent (in structure) to the postfix bar notation with the bars stripped off, the elements of the array $V$ are copied to the corresponding elements of the array $V$ and zeroes are set as elements of $V$ at the positions where bars occur.

3.3 Grammars and pushdown automata for particular linear notations of trees

In this section we formalise, in terms of grammar, the linear notations presented in Sections 3.1 and 3.2. We also introduce the following notation that will be used for illustrating transitions in PDA transition diagrams.

Notation 12 (Transitions of PDA) For illustrating PDA transition diagrams, we will use an arc with a label $a|\alpha \mapsto \beta$ when describing a transition $\delta(q_1, a, \alpha) \mapsto (q_2, \beta)$, with the arc originating from state $q_1$ and leading to $q_2$. 
3.3.1 Ranked trees

**Definition 58 (Grammar for prefix notation)** Let $\mathcal{A} = (\Sigma, \varphi)$ be a ranked alphabet such that $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \Sigma_n$, where $\Sigma_i = \{a_1^i, a_2^i, \ldots, a_{r_i+1}^i\}$, for all $0 \leq i \leq n$, and $\Sigma_i$ is a set consisting of only symbols of arity $i$. Then the grammar $G = (N, T, P, S)$ containing just one nonterminal symbol $S$ ($N = \{S\}$), the set $T = \Sigma$ of terminal symbols and the rules $P$ defined as

$$
S \rightarrow a_0^1 | a_0^2 | \ldots | a_0^{r_0} \\
S \rightarrow a_1^1 S | a_1^2 S | \ldots | a_1^{r_1} S \\
S \rightarrow a_2^1 SS | a_2^2 SS | \ldots | a_2^{r_2} SS \\
\vdots \\
S \rightarrow a_n^1 S^n | a_n^2 S^n | \ldots | a_n^{r_n+1} S^n
$$

is the grammar generating the prefix notations of ranked trees.

**Definition 59 (Grammar for postfix notation)** Let $\mathcal{A} = (\Sigma, \varphi)$ be a ranked alphabet such that $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \Sigma_n$, where $\Sigma_i = a_1^i a_2^i \ldots a_{r_i+1}^i$, for all $0 \leq i \leq n$, and $\Sigma_i$ is a set consisting of only symbols of arity $i$. Then the grammar $G = (N, T, P, S)$ containing just one nonterminal symbol $S$ ($N = \{S\}$), the set $T = \Sigma$ of terminal symbols and the rules $P$ defined as

$$
S \rightarrow a_0^1 | a_0^2 | \ldots | a_0^{r_0} \\
S \rightarrow S a_1^1 | S a_1^2 | \ldots | S a_1^{r_1} \\
S \rightarrow S S a_2^1 | S S a_2^2 | \ldots | S S a_2^{r_2} \\
\vdots \\
S \rightarrow S^n a_1^1 | S^n a_2^n | \ldots | S^n a_{n+1}^{r_n+1}
$$

is the grammar generating the prefix notations of ranked trees.

Given a grammar $G = (N, \Sigma, P, S)$ that generates prefix notations of tree (see Definition 58), we may can construct a basic deterministic PDA $M = (\{q_0\}, \Sigma, \{S\}, \delta, q_0, \{S\}, \emptyset)$ that accepts the language $L(G)$ by empty pushdown store, by defining its transition function $\delta$ in the following way: for any rule $S \rightarrow a_i^j S^i$, create a transition $\delta(q_0, a_i^j, S) \mapsto (q_0, \varepsilon)$.
Example 6 Let $G = (\{S\}, \{a_0, a_1, a_2\}, P, S)$ be a grammar that generates prefix notations of trees with nodes of rank up-to 2. The rules ($P$) of the grammar are defined as:

- $S \rightarrow a_0$
- $S \rightarrow a_1 S$
- $S \rightarrow a_2 S S$

We may construct a PDA $M = (\{q_0\}, \Sigma, \{S\}, \delta, q_0, \{S\}, \emptyset)$ that accepts $L(G)$ by defining its transition function $\delta$ as

- $\delta(q_0, a_0, S) \rightarrow (q_0, \varepsilon)$
- $\delta(q_0, a_1, S) \rightarrow (q_0, S)$
- $\delta(q_0, a_2, S) \rightarrow (q_0, S S)$

The resulting PDA is depicted in Figure 3.2a.

The postfix notation generated by some grammar $G$ is not a prefix-free language, i.e. prefixes of words in $L(G)$ are also part of $L(G)$, and therefore we cannot construct a deterministic PDA that accepts $L(G)$ by empty pushdown store, since the pushdown store will be emptied at the first occurrences of a prefix that is part of $L(G)$.

We first present an analogous deterministic PDA to the one shown in Example 6, that parses a given postfix notation $x$. Every time a symbol $x[i]$ is read and the corresponding transition taken, where $1 \leq i \leq |x|$, the contents of the
pushdown stores denote the number of trees whose concatenated postfix notations form \( x[1...i] \). This corresponds to the LR syntax analysis, where the pushdown store holds the history of subtrees of the parse tree that were processed before their parent node is reached.

**Example 7** Let \( G = \{S\}, \{a_0, a_1, a_2\}, P, S \) be a grammar that generates prefix notations of trees with nodes of rank up-to 2. The rules \( (P) \) of the grammar are defined as:

\[
S \rightarrow a_0 \\
S \rightarrow S a_1 \\
S \rightarrow S S a_2
\]

We may construct a PDA \( M = (\{q_0\}, \Sigma, \{S\}, \delta, q_0, \emptyset, \emptyset) \) that parses \( L(G) \) by defining its transition function \( \delta \) as

\[
\delta(q_0, a_0, \varepsilon) \mapsto (q_0, S) \\
\delta(q_0, a_1, S) \mapsto (q_0, S) \\
\delta(q_0, a_2, SS) \mapsto (q_0, S)
\]

The resulting PDA is depicted in Figure 3.2b.

It is possible, however, to construct a deterministic PDA accepting \( L(G) \) (the postfix notations generated by \( G \)) by final state.

### 3.3.2 Unranked trees

**Definition 60 (Grammar for prefix bar notation)** Let \( \Sigma = \{a_1, a_2, \ldots a_n\} \) be an alphabet. Then the grammar \( G = (N, T, P, S) \) consisting of the set \( N = \{S, R\} \) of nonterminal symbols, the set \( T = \Sigma \cup \{\uparrow\} \) of terminal symbols and the rules \( P \) defined as

\[
S \rightarrow R \uparrow \\
R \rightarrow a_1 \mid a_2 \mid \ldots \mid a_n \\
R \rightarrow RR \uparrow
\]

is the grammar generating the prefix bar notations of unranked trees.
Given a grammar \( G = (N, \Sigma, P, S) \) that generates prefix bar notations, we can construct a deterministic PDA \( M = (\{q_0\}, \Sigma, \{S, Z_0\}, \delta, q_0, \{Z_0\}, \emptyset) \) that accepts \( L(G) \) by empty pushdown store, by defining its transition function \( \delta \) in the following way: for each terminal symbol \( a \) in \( \Sigma \setminus \{\uparrow\} \) construct two transitions:

\[
\delta(q_0, a, Z_0) \rightarrow (q_0, S)
\]

\[
\delta(q_0, a, S) \rightarrow (q_0, SS)
\]

and one transition \( \delta(q_0, \uparrow, S) \rightarrow (q_0, \varepsilon) \). The transitions simply correspond to pushing one symbol on top of the pushdown store when reading a terminal symbol different than the bar, and popping one symbol from the pushdown store when reading a bar.

**Definition 61 (Grammar for postfix bar notation)** Let \( \Sigma = \{a_1, a_2, \ldots, a_n\} \) be an alphabet. Then the grammar \( G = (N, T, P, S) \) consisting of the set \( N = \{S, R\} \) of nonterminal symbols, the set \( T = \Sigma \cup \{\uparrow\} \) of terminal symbols and the rules \( P \) defined as

\[
S \rightarrow \uparrow R
\]

\[
R \rightarrow a_1 \mid a_2 \mid \ldots \mid a_n
\]

\[
R \rightarrow \uparrow RR
\]

is the grammar generating the postfix bar notations of unranked trees.

Similarly as in the case of the prefix bar notation, we can construct a deterministic PDA \( M = (\{q_0\}, \Sigma, \{S, Z_0\}, \delta, q_0, \{Z_0\}, \emptyset) \) that accepts the language \( L(G) \) of postfix bar notations generated by a grammar \( G = (N, \Sigma, P, S) \) by empty pushdown store, by defining its transition function \( \delta \) in the following way: for each terminal symbol \( a \) in \( \Sigma \setminus \{\uparrow\} \) construct the transition \( \delta(q_0, a, S) \rightarrow (q_0, \varepsilon) \) and for the bar symbol \( \uparrow \) construct two transitions

\[
\delta(q_0, \uparrow, Z_0) \rightarrow (q_0, S)
\]

\[
\delta(q_0, \uparrow, S) \rightarrow (SS)
\]

The transitions are dual to the ones in the prefix bar notations. Reading a terminal symbol other than the bar corresponds to popping one symbol from the pushdown store, and reading a bar corresponds to pushing one symbol on top of the pushdown store.
As in Section 3.3.1, we will present examples for constructing the deterministic PDA accepting the languages generated by the grammars in Definition 60 and 61.

**Example 8** Let $G = (\{S\}, \{a, \uparrow\}, P, S)$ be a grammar that generates prefix bar notations of trees. The rules $P$ of this grammar are defined as:

\[
\begin{align*}
S & \rightarrow R \uparrow \\
R & \rightarrow a \\
R & \rightarrow RR \uparrow
\end{align*}
\]

We may construct a deterministic PDA $M = (\{q_0\}, \Sigma, \{S, Z_0\}, \delta, q_0, Z_0, \emptyset)$ that accepts $L(G)$ by empty pushdown store by defining its transition function $\delta$ as

\[
\begin{align*}
\delta(q_0, a, Z_0) & \rightarrow S \\
\delta(q_0, a, S) & \rightarrow SS \\
\delta(q_0, \uparrow, S) & \rightarrow \varepsilon
\end{align*}
\]

The resulting PDA is depicted in Figure 3.3a.

**Example 9** Let $G = (\{S\}, \{a, \uparrow\}, P, S)$ be a grammar that generates postfix bar notations of trees. The rules $P$ of this grammar are defined as:

\[
\begin{align*}
S & \rightarrow \uparrow R \\
R & \rightarrow a \\
R & \rightarrow \uparrow RR
\end{align*}
\]
We may construct a deterministic PDA \( M = (\{q_0\}, \Sigma, \{S, Z_0\}, \delta, q_0, Z_0, \emptyset) \) that accepts \( L(G) \) by empty pushdown store by defining its transition function \( \delta \) as

\[
\begin{align*}
\delta(q_0, \uparrow, Z_0) & \rightarrow S \\
\delta(q_0, \uparrow, S) & \rightarrow SS \\
\delta(q_0, a, S) & \rightarrow \varepsilon
\end{align*}
\]

The resulting PDA is depicted in Figure 3.3b.
Chapter 4

Tree Pattern Matching

“Doubt everything. Find your own light.”

— Siddhārtha Gautama Buddha

Tree pattern matching, the process of finding all occurrences of a given pattern in a subject tree, is an important operation on which a number of tasks in computer science are based on, i.e. mechanical theorem proving, term-rewriting, instruction selection and non-procedural programming languages [63]. In addition, tree pattern matching has direct applications in computational biology, e.g. glycan classification [79], and in exact and approximate pattern matching and discovery in RNA secondary structure [86].

We distinguish among two types of tree pattern matching: subtree and tree template matching. While subtrees consist of only specific, fixed-labeled nodes, tree templates have some of their leaves denoted as “don’t care”, representing arbitrary subtrees – such nodes match any subtree.

Since 1960, many methods, namely [2, 19, 56, 59, 63, 80, 99], have been described in the literature for solving the tree pattern matching problem. However, most of them lack clear references to a systematic approach of the standard theory of formal languages, grammars and automata. In general, there exist two such approaches using automata. Linearising trees and using string automata represents the first approach [56, 59]. Usage of finite automata is, however, not sufficient, as linear notations of trees are context-free languages. Therefore, the pushdown automaton seems to be an appropriate model of computation. The second approach does not reside on tree linearisation, but represents a generalisation from string automata to tree automata. Cleophas in [26] presents a systematic approach for solving the tree pattern matching problem utilising finite tree automata, which accept regular tree languages, as the computational model.
Recently, it has been shown that the deterministic pushdown automaton is an appropriate model of computation for a proper superset of the regular tree languages \[69\]. Based on this fact, we present two algorithms, based on deterministic pushdown automata, solving the subtree and tree template matching respectively.

We divide this chapter in three sections. In the first section we present a solution for the subtree matching. In the second section we give a pushdown automata based algorithm for solving the tree template matching problem for ranked trees and in the last section we give an algorithm for solving the tree template matching problem for unranked trees.

4.1 Subtree Matching

The material presented in this section was published in part in \[48, 53, 54\].


In this section, we first provide the definitions of the problems that are to be solved. We follow by introducing a naive approach to solving the problems and then we propose optimal algorithms for solving them.

**Problem 1 (Subtree Matching)** Given two trees \(t\) and \(p\), find all occurrences of \(p\) in \(t\).

An example of subtree matching is illustrated in Figure 4.1.

**Problem 2 (Multiple Subtree Matching)** Given a tree \(t\) and a set of \(r\) trees \(P = \{p_1, p_2, \ldots, p_r\}\), find all occurrences of each tree \(p_i\), \(1 \leq i \leq r\), in \(t\).

In the following section we present methods based on pushdown automata solving Problems 1 and 2.
4.1.1 A naive approach

In this section we give a naive algorithm for subtree matching running in time $O(|t| \cdot |p|)$, where $t$ is the subject tree and $p$ the tree pattern.

\[
\text{Algorithm 24: SUBTREE-MATCHING-NAIVE} \\
\text{Input :} \text{The prefix notations } \text{pref}(t) = x[1..n] \text{ and } \text{pref}(p) = y[1..m] \text{ of } t \text{ and } p \\
\text{Output:} \text{The positions in } t \text{ where } p \text{ is matched} \]

\begin{verbatim}
1 for i ← 1 to n − m + 1 do 
    match ← true 
    for j ← 1 to m do 
        if x[i + j − 1] ≠ y[j] then 
            match ← false 
            break 
    if match = true then OUTPUT(i)
\end{verbatim}

The naive approach uses a sliding window mechanism. The size of the window is $|p|$, and every factor of size $|p|$ of $\text{pref}(t)$ is checked for match with $\text{pref}(p)$. In other words, all factors $x[i..i + |p| − 1]$, for all $1 \leq i \leq |t| − |p| + 1$, where
$x = \text{pref}(t)$, are checked whether they match $\text{pref}(p)$. The set of starting positions of matching factors represents the occurrences of $p$ in $t$.

**Lemma 6** Algorithm 24 runs in time $O(nm)$ and requires $O(1)$ space, where $n = |t|$ and $m = |p|$.

**Proof** It is obvious from the fact that the algorithm consists of reading the prefix notation of size $n$ of the subject tree from left to right, and at each step does at most $m$ comparisons, where $m$ is the size of the pattern. The complexity can be clearly shown with an example where the prefix notation of $t$ is $x = a_1^{n-1} a_0$ and the pattern’s prefix notation is $y = a_1^{m-1} a_0$, in which the naive algorithm would carry out exactly $(n - m + 1)m$ comparisons. $\square$

### 4.1.2 Subtree matching by pushdown automata

In this section we present an algorithm for constructing a so-called Subtree Matching PDA (SMPDA) solving Problem 1. We first present and prove an algorithm for constructing a PDA accepting a given tree in its prefix notation. Based on this construction we then design an algorithm that, when given a tree $p$, constructs an SMPDA matching $p$ in some given tree $t$.

**Algorithm 25: Tree-Match-PDA**

| Input                      | Prefix notation $\text{pref}(t) = x[1..|t|]$ of tree $t$ on ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$ |
|----------------------------|--------------------------------------------------------------------------------------------------|
| Output                     | A PDA accepting language $L = \{\text{pref}(t)\}$                                             |

1. $Q \leftarrow \{i \mid 0 \leq i \leq |t|\}$
2. for $i \leftarrow 1$ to $|t|$ do
3.   $\delta(i - 1, x[i], S) \mapsto (i, S^{\varphi(x[i])})$
4. $M \leftarrow (Q, \mathcal{A}, \{S\}, \delta, 0, S, \{|t|\})$

**Example 10** Recall the tree $t$ from Figure 3.1 having prefix notation $\text{pref}(t) = a_2 a_2 a_0 a_1 a_0 a_1 a_0$. The transition diagram of PDA $M = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ accepting $\text{pref}(t)$ is illustrated in Figure 4.2. The states of $M$ is the set $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$, the set of final states is $F = \{7\}$ and the mapping $\delta$ is defined as follows:
Finally, Table 4.1 shows the sequence of transitions (trace) taken by $M$ when reading the prefix notation of $t$.

Figure 4.2: Transition diagram of the (deterministic) PDA accepting tree $t$ in prefix notation $\text{pref}(t) = a_2 a_2 a_0 a_1 a_0 a_1 a_0$

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Pushdown Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$a_2$ $a_2$ $a_0$ $a_1$ $a_0$ $a_1$ $a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>1</td>
<td>$a_2$ $a_0$ $a_1$ $a_0$ $a_1$ $a_0$</td>
<td>$S$ $S$</td>
</tr>
<tr>
<td>2</td>
<td>$a_0$ $a_1$ $a_0$ $a_1$ $a_0$</td>
<td>$S$ $S$ $S$</td>
</tr>
<tr>
<td>3</td>
<td>$a_1$ $a_0$ $a_1$ $a_0$</td>
<td>$S$ $S$</td>
</tr>
<tr>
<td>4</td>
<td>$a_0$ $a_1$ $a_0$</td>
<td>$S$ $S$</td>
</tr>
<tr>
<td>5</td>
<td>$a_1$ $a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>6</td>
<td>$a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>7</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

Table 4.1: Trace of the (deterministic) PDA $M$ from Example 10 for tree $t$ in prefix notation $\text{pref}(t) = a_2 a_2 a_0 a_1 a_0 a_1 a_0$

**Lemma 7** Let $M = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ be a PDA constructed using Algorithm 25, where $\mathcal{A} = (\Sigma, \varphi)$ and thus its transitions have the form $\delta(q, a, S) = (p, S^{\varphi(a)})$. Then, if $(q, x, S) \vdash^+_M (p, \varepsilon, S^j)$, it holds that $j = ac(x)$. 

66
Thus, the lemma holds.

**Proof** By induction on the length of $x$:

1. Assume $|x| = 1$. Then, $(q, x, S) \vdash_M (p, \varepsilon, S^{\varphi(x)})$. Since $ac(x) = \sum_{i=1}^{x} \varphi(x[i]) = |x| + 1$, it is evident that $\varphi(x) = ac(x)$. Thus, the claim holds for the case $|x| = 1$.

2. Assume the claim holds for a string $x[1..k]$, where $k \geq 1$. In other words, it is valid to assume that $(q, x, S) \vdash^k_M (p, \varepsilon, S^j)$, where $j = ac(x)$. We must prove that the claim also holds for $y[1..k+1]$, where $y = xa$ and $a \in \Sigma$. The sequence of transitions taken when processing $y$ is $(q, y, S) \vdash^k_M (r, a, S')$, where $\ell = j + \varphi(a) - 1$. From our assumption that $j = ac(x)$ it holds that $\ell = ac(x) + \varphi(a) - 1 = \sum_{i=1}^{x} \varphi(x[i]) - |x| + 1 + \varphi(a) - 1 = \sum_{i=1}^{k+1} \varphi(y[i]) - |y| + 1 = ac(y)$ and so the claim holds for $|y| = k + 1$.

Thus, the lemma holds. □

The correctness of the (deterministic) PDA constructed by using Algorithm 25, which accepts trees in prefix notation, is described in Lemma 8.

**Lemma 8** Given the prefix notation of a tree $t$ over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$ the PDA $M = (\{0, 1, 2, \ldots, |t|\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$, constructed by Algorithm 25, accepts the language $L = \{\text{pref}(t)\}$.

**Proof** By induction on the height of the tree $t$:

1. If tree $t$ has just one node $a$, where $\varphi(a) = 0$, then the height $h(t) = 0$, $\text{pref}(t) = a$, $\delta(0, a, S) = (1, \varepsilon) \in \delta$, $(0, a, S) \vdash_{M_{\text{pref}(t)}} (1, \varepsilon, \varepsilon)$ and the claim holds for that tree.

2. Assume that claim holds for trees $t_1, t_2, \ldots, t_p$, where $p \geq 1$, $h(t_i) \leq m$, for all $1 \leq i \leq p$, $m \geq 0$.

We have to prove that the claim holds also for each tree $t$ such that $\text{pref}(t) = a \text{ pref}(t_1) \text{ pref}(t_2) \ldots \text{ pref}(t_p)$, where $\varphi(a) = p$, and $h(t) \geq m + 1$: Since $\delta(0, a, S) = (1, S^p) \in \delta$, and $(0, a \text{ pref}(t_1) \text{ pref}(t_2) \ldots \text{ pref}(t_p), S) \vdash_{M_{\text{pref}(t)}} (1, \text{ pref}(t_2) \ldots \text{ pref}(t_p), S^p)$

$\vdash_{M_{\text{pref}(t)}} (1, \text{ pref}(t_2) \ldots \text{ pref}(t_p), S^p)$

$\vdash_{M_{\text{pref}(t)}} (i, \text{ pref}(t_2) \ldots \text{ pref}(t_p), S^{p-1})$

$\vdash_{M_{\text{pref}(t)}} (j, \text{ pref}(t_p), S)$

$\vdash_{M_{\text{pref}(t)}} (k, \varepsilon, \varepsilon)$, the claim holds for that tree.

Thus, the lemma holds. □
We are now in a position to present the construction of the deterministic SM-PDA for trees in prefix notation. The construction consists of two steps. First, a nondeterministic SMPDA is constructed by Algorithm 26. This nondeterministic SMPDA is an extension of the PDA accepting trees in prefix notation, which is constructed by Algorithm 25. Second, the constructed nondeterministic SMPDA is transformed to the equivalent deterministic SMPDA. In spite of the fact that the determinisation of a nondeterministic PDA is not possible in general, the constructed nondeterministic SMPDA is an input–driven PDA and therefore can be determinised [104].

**Algorithm 26: Nondeterministic-Subtree-Matching-PDA**

- **Input**: $M = (Q, A = (\Sigma, \varphi), \{S\}, \delta, 0, S, F)$ constructed using Algorithm 25 given a tree $t$
- **Output**: SMPDA accepting language $L = \{ y\text{pref}(t), \text{pref}(t) \}$, for all $y \in \Sigma^+$ such that $ac(y) > 0$

1. $Q \leftarrow \{ i \mid 0 \leq i \leq |t| \}$
2. **foreach** $a \in \Sigma$ **do**
3. \[ \delta(0, a, S) \mapsto (1, SS) \]
4. $M \leftarrow (Q, A, \{S\}, \delta, 0, S, \{|t|\})$

We demonstrate Algorithm 26 with the following example.

**Example 11** The subtree matching PDA, constructed by Algorithm 26 for tree $t$ with prefix notation $\text{pref}(t) = a2 a2 a0 a1 a0 a1 a0$, is the nondeterministic SPDA $M = (Q, A, \{S\}, \delta, 0, S, F)$, where $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$, $F = \{7\}$ and the mapping $\delta$ is defined as follows:

\[
\begin{align*}
\delta(0, a_2, S) &\mapsto (1, SS) & \delta(5, a_1, S) &\mapsto (6, S) \\
\delta(1, a_2, S) &\mapsto (2, SS) & \delta(6, a_0, S) &\mapsto (7, \varepsilon) \\
\delta(2, a_0, S) &\mapsto (3, \varepsilon) & \delta(0, a_2, S) &\mapsto (0, SS) \\
\delta(3, a_1, S) &\mapsto (4, S) & \delta(0, a_1, S) &\mapsto (0, S) \\
\delta(4, a_0, S) &\mapsto (5, \varepsilon) & \delta(0, a_0, S) &\mapsto (0, \varepsilon)
\end{align*}
\]

The transition diagram of the nondeterministic PDA $M$ is illustrated in Figure 4.3.
Theorem 5 Given the prefix notation $x = \text{pref}(t)$ of some tree $t$ constructed from a ranked alphabet $A = (\Sigma, \varphi)$, Algorithm 26 constructs a nondeterministic SMPDA accepting the language $L = \{ yx \mid y \in \Sigma^+, ac(y) > 0 \} \cup \{ x \}$ and thus matching all occurrences of $x$ in the prefix notation of some input tree $T$.

Proof Trivially, the nondeterminism is caused because of the looped transitions in the initial state. The automaton can read arbitrary many symbols from the alphabet and stay at the initial state. The pushdown store operations ensure that the arity check of the read string is always positive or equal to zero, which means that the read string is a valid prefix of the prefix notation of some tree. At any point, the nondeterminism allows the automaton to proceed to state 1 and read $\text{pref}(t)$ as proved in Lemma 8. □

So far we have described a method for constructing a nondeterministic SMPDA for some given tree pattern. To construct an equivalent deterministic SMPDA we use the transformation described by Algorithm 27, which transforms a nondeterministic input-driven PDA to an equivalent deterministic PDA. It is loosely based on the well known transformation of nondeterministic finite automata to equivalent deterministic ones which constructs the states of the deterministic automaton as subsets of the powerset of states of the nondeterministic automaton and selects only a set of accessible states (subsets) [65]. Similarly, in our case, states of the resulting deterministic SPDA correspond to subsets of the powerset of states of the original nondeterministic SPDA constructed using Algorithm 26.

We demonstrate Algorithm 27 with the following example.

Example 12 Recall the nondeterministic SMPDA $M$ constructed in Example 11 which is illustrated in Figure 4.3. Running Algorithm 27 with $M$ as input, we construct a new, deterministic SMPDA $M' = (Q, A, \{S\}, \delta, \{0\}, S, F)$ which is equivalent to $M$ (i.e. accept the same language), where

$$Q = \{\{0\}, \{0, 1\}, \{0, 1, 2\}, \{0, 3\}, \{0, 4\}, \{0, 5\}, \{0, 6\}, \{0, 7\}\}$$

$$F = \{\{0, 7\}\}$$
Algorithm 27: SUBSET-CONSTRUCTION-PDA

Input: Nondeterministic input-driven PDA
\[ M = (Q, A = (\Sigma, \varphi), \{S\}, \delta, 0, S, F) \]

Output: Deterministic PDA \( M' = (Q', A, \{S\}, \delta', q_I, S, F') \)

1. \( Q' \leftarrow \{\{0\}\} \)
2. \( q_I \leftarrow \{0\} \)
3. \( q_I . \text{marked} \leftarrow \text{false} \)
4. foreach \( q \in Q' \text{ such that } q_. \text{marked} = \text{false} \) do
   5. foreach \( a \in \Sigma \) do
      6. \( q' = \{q'' | \delta(p, a, \alpha) = (q'', b), \forall p \in q\} \)
      7. \( \delta'(q, a, S) \leftarrow (q', S^{p(a)}) \)
      8. if \( q' \notin Q' \) then
         9. \( q'. \text{marked} \leftarrow \text{false} \)
         10. \( Q' \leftarrow Q' \cup \{q'\} \)
      11. \( q_. \text{marked} \leftarrow \text{true} \)
   12. \( F' \leftarrow \{q' | q' \in Q', q' \cap F \neq \emptyset\} \)

and the mapping \( \delta \) is defined as follows:

\[
\begin{align*}
\delta(\{0\}, a_0, S) & \mapsto (\{0\}, \varepsilon) & \delta(\{0\}, a_0, S) & \mapsto (\{0\}, \varepsilon) \\
\delta(\{0\}, a_1, S) & \mapsto (\{0\}, S) & \delta(\{0\}, a_1, S) & \mapsto (\{0\}, S') \\
\delta(\{0\}, a_2, S) & \mapsto (\{0\}, SS) & \delta(\{0\}, a_2, S) & \mapsto (\{0\}, SS') \\
\delta(\{0, 1\}, a_0, S) & \mapsto (\{0\}, \varepsilon) & \delta(\{0, 1\}, a_0, S) & \mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 1\}, a_1, S) & \mapsto (\{0\}, S) & \delta(\{0, 1\}, a_1, S) & \mapsto (\{0\}, S') \\
\delta(\{0, 1\}, a_2, S) & \mapsto (\{0\}, 1, 2), SS) & \delta(\{0, 1\}, a_2, S) & \mapsto (\{0\}, 1, 2), SS') \\
\delta(\{0, 1, 2\}, a_0, S) & \mapsto (\{0, 3\}, \varepsilon) & \delta(\{0, 1, 2\}, a_0, S) & \mapsto (\{0, 3\}, \varepsilon) \\
\delta(\{0, 1, 2\}, a_1, S) & \mapsto (\{0\}, S) & \delta(\{0, 1, 2\}, a_1, S) & \mapsto (\{0\}, S) \\
\delta(\{0, 1, 2\}, a_2, S) & \mapsto (\{0\}, 1, 2), SS) & \delta(\{0, 1, 2\}, a_2, S) & \mapsto (\{0\}, 1, 2), SS') \\
\delta(\{0, 3\}, a_0, S) & \mapsto (\{0\}, \varepsilon) & \delta(\{0, 3\}, a_0, S) & \mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 3\}, a_1, S) & \mapsto (\{0\}, 4, S) & \delta(\{0, 3\}, a_1, S) & \mapsto (\{0\}, 4, S) \\
\delta(\{0, 3\}, a_2, S) & \mapsto (\{0\}, 1, 2), SS) & \delta(\{0, 3\}, a_2, S) & \mapsto (\{0\}, 1, 2), SS')
\end{align*}
\]

The transition diagram of \( M' \) is illustrated in Figure 4.4.

Note that the transition diagram of the deterministic SMPDA over the ranked alphabet \( A = (\Sigma, \varphi) \) is similar to the transition diagram of deterministic string matching finite automaton over alphabet \( \Sigma \) presented in Figure 4.5. The difference
We prove the following claim by induction on proof input tree $t$.

Let $M = (Q, A, \{S\}, \delta, q_0, S, F)$ be a nondeterministic, input-driven PDA. Then, the deterministic PDA $M' = (Q', A, \{S\}, \delta', \{q_0\}, S, F')$ constructed using Algorithm 27 with $M$ as input, is equivalent to $M$.

**Proof** We prove the following claim by induction on $i$:

$$(q_1, w, S) \vdash_M^i (q_2, \varepsilon, S') \iff q_2' = \{ p \mid (q, w, S) \vdash_M^i (p, \varepsilon, S') \text{ for some } q \in q_1 \}$$

1. We first prove the claim for $i = 1$

   - **If**: If $(q_1, a, S) \vdash_M^i (q_2, \varepsilon, S')$, then $\exists q \in q_1'$ such that $(q, a, S) \vdash_M (p, \varepsilon, S')$, $p \in q_2'$.
   - **Only If**: If $(q, a, S) \vdash_M (p, \varepsilon, \beta)$, then for each $q_1' \in Q'$, where $q \in q_1'$, it holds that $(q_1', a, S) \vdash_M (q_2', \varepsilon, S')$, where $p \in q_2'$.

2. Assume the claim holds for $i = 1, 2, \ldots, k, k \geq 1$, i.e.

   $$(q_1, w, S) \vdash_M^k (q_2, \varepsilon, S') \iff q_2' = \{ p \mid (q, w, S) \vdash_M^k (p, \varepsilon, S') \text{ for some } q \in q_1' \}$$

   holds. We must prove that the claim also holds for $i = k + 1$.

---

**Figure 4.4**: Transition diagram of the deterministic SMPDA accepting language $L = \{ xa_2a_2a_0a_1a_0a_1a_0 \mid x \in \Sigma^+, ac(x) > 0 \} \cup \{ a_2a_2a_0a_1a_0a_1a_2a_0a_0 \}$

between the two types of automata is the pushdown operations appearing in the SMPDA and which ensure the validity of the input tree; the input tree is valid if and only if the pushdown store of the SMPDA is empty after the last symbol from the prefix notation of the input tree is read.

To complete the example, the sequence of transitions (trace) performed by the deterministic SMPDA $M$, constructed for a tree pattern $p$, when processing an input tree $t$, where $\text{pref}(t) = a_2a_2a_0a_1a_0a_1a_0a_1a_2a_0a_0$, is shown in Table 4.2.
Figure 4.5: Transition diagram of the deterministic string matching finite automaton accepting language \( L = \Sigma^*\{a_2a_2a_0a_0a_0a_1a_1\} \)

- **If**: If \((q_1', w, S) \vdash_{M'}^k (q_2', a, S^\ell) \vdash_{M'} (q_3', \varepsilon, S^j)\), then there exists a state \( q \in q_2' \), where \((q, a, S^\ell) \vdash_M (p, \varepsilon, S^j)\), \( p \in q_3' \).

- **Only If**: If \((q_0, w, S) \vdash_M (q, a, S^\ell) \vdash_M (p, \varepsilon, S^j)\), then for each \( q_1' \in Q' \), where \( q \in q_1' \), it holds that \((q_1', a, S^\ell) \vdash_{M'} (q_2', \varepsilon, S^j)\), where \( p \in q_2' \).

\[ \square \]

**Theorem 7** Given a tree \( t \) with \( n \) nodes and its prefix notation \( \text{pref}(t) \), the deterministic subtree matching PDA \( M_{pd}t(t) \) constructed by Algorithm 26 and 27 is made of exactly \( n + 1 \) states, one pushdown symbol and \(|A|(n + 1)\) transitions.

**Proof** The states and transitions of the deterministic SMPDA directly correspond to the states and transitions of the deterministic string matching finite automaton. The difference is in the pushdown store operations. A detailed proof on the size of the string matching finite automaton, which actually resembles the KMP algorithm, is presented in [36]. \[ \square \]
<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>PDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0}</td>
<td>$S$</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>$a_2$ $a_2$ $a_0$ $a_1$ $a_0$ $a_1$ $a_0$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SS$</td>
</tr>
<tr>
<td>{0, 1, 2}</td>
<td>$a_2$ $a_0$ $a_1$ $a_0$ $a_1$ $a_0$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SSS$</td>
</tr>
<tr>
<td>{0, 1, 2}</td>
<td>$a_0$ $a_1$ $a_0$ $a_1$ $a_0$ $a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SSSS$</td>
</tr>
<tr>
<td>{0, 3}</td>
<td>$a_1$ $a_0$ $a_1$ $a_0$ $a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SSS$</td>
</tr>
<tr>
<td>{0, 4}</td>
<td>$a_0$ $a_1$ $a_0$ $a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SSS$</td>
</tr>
<tr>
<td>{0, 5}</td>
<td>$a_1$ $a_0$ $a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SS$</td>
</tr>
<tr>
<td>{0, 6}</td>
<td>$a_0$ $a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$SS$</td>
</tr>
<tr>
<td>{0, 7}</td>
<td>match $a_1$ $a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>{0}</td>
<td>$a_1$ $a_1$ $a_2$ $a_0$ $a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>{0}</td>
<td>$a_2$ $a_0$ $a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>{0}</td>
<td>$a_0$</td>
<td>$S$</td>
</tr>
<tr>
<td>{0}</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

Table 4.2: Trace of deterministic subtree PDA $M$ from Example 12 for an input tree $t$ with prefix notation $a_2a_2a_2a_0a_1a_0a_0a_1a_1a_2a_0a_0$
4.1.3 Multiple subtree matching by pushdown automata

So far, we have presented a method which, for a given subtree pattern \( p \), locates all occurrences of \( p \) in some subject tree \( t \). In this section we will focus on the problem of finding all occurrences of a finite set of subtree patterns in some subject tree as described in Problem 2.

We first provide a brief description of the naive method for solving Problem 2 (multiple subtree matching), and is presented in Algorithm 28. Given a subject tree \( t \), where \( n = |t| \), and a set of \( k \) subtree patterns \( P = \{p_1, p_2, \ldots, p_k\} \), the algorithm runs in \( O(nm) \), where \( m = \sum_{i=1}^{k} |p_i| \), if Algorithm 24 is used for subtree matching in line 2, or \( O(kn) \) if the PDA-based algorithm presented in Section 4.1 is used.

Algorithm 28: Multiple-Subtree-Matching-Naive

\[
\text{Input} : \text{The prefix notations } x = \text{pref}(t) \text{ and } y_i = \text{pref}(p_i) \text{ of } t \text{ and } p_i, \\
1 \leq i \leq k \\
\text{Output}: \text{The positions in } t \text{ where } p_i \text{ is matched} \\
1 \text{ for } i \leftarrow 1 \text{ to } k \text{ do} \\
2 \quad \text{Subtree-Matching-Algorithm}(x, y_i) \\
\]

The matching process for multiple patterns may be optimised further by preprocessing the set of patterns and constructing an automaton which accepts strings that fulfill the following properties:

1. They are prefixes of a valid prefix notation of some tree.
2. Their suffix is the prefix notation of some tree in the tree pattern set.

In addition to the linear runtime of the matching process, we also prove that the space complexity of the constructed automaton is linear to the sum of sizes of the given tree patterns.

Lemma 9 Algorithm 29 constructs a PDA accepting the language \( L = \{ \text{pref}(p_i) \mid 1 \leq i \leq k \} \) by final state and empty pushdown store.

Proof The constructed PDA is a prefix tree (trie) of the linear notations of the given \( k \) subtree patterns. The proof can be easily deduced from the proof of Lemma 8. \( \square \)
Algorithm 29: Multiple-TreeMatch

Input: A set of \( k \) subtree patterns \( p_1, p_2, \ldots, p_k \) on ranked alphabet \( A = (\Sigma, \varphi) \)

Output: A PDA accepting language \( L = \{ \text{pref}(p_i) \mid 1 \leq i \leq k \} \)

1. \( F \leftarrow \emptyset \)
2. \( q \leftarrow 0 \)
3. for \( i \leftarrow 1 \) to \( k \) do
   4. \( r \leftarrow 0 \)
   5. \( x \leftarrow \text{pref}(p_i) \)
   6. for \( j \leftarrow 1 \) to \( |x| \) do
      7. if \( \delta(r, x[j], S) \) not defined then
         8. \( q \leftarrow q + 1 \)
         9. \( \delta(r, x[j], S) \leftarrow (q, S^\varphi(x[j])) \)
         10. \( r \leftarrow q \)
      else
         11. \( (r, S^\varphi(x[j])) \leftarrow \delta(r, x[j], S) \)
   12. \( F \leftarrow F \cup \{q\} \)
7. \( Q \leftarrow \bigcup_{i=0}^{k} i \)
8. \( M \leftarrow (Q, A, \{S\}, \delta, 0, S, F) \)

Figure 4.6: Set of subtree patterns

Example 13 Let \( p_1, p_2 \) and \( p_3 \) be the subtree patterns illustrated in Figure 4.6 with prefix notation \( \text{post}(p_1) = a_2a_2a_0a_0b_0 \), \( \text{post}(p_2) = a_2b_1a_0a_0 \) and \( \text{post}(p_3) = a_2a_0a_0 \), respectively. The transition diagram of the PDA \( M = (Q, A, \{S\}, \delta, 0, S, F) \) constructed using Algorithm 29 is illustrated in Figure 4.7, where \( Q = \bigcup_{i=0}^{10} i \) and
\( F = \{5,8,10\} \). The transition function is defined as follows:

\[
\begin{align*}
\delta(0, a_2, S) &\mapsto (1, SS) & \delta(3, a_0, S) &\mapsto (4, \epsilon) \\
\delta(1, a_2, S) &\mapsto (2, SS) & \delta(4, b_0, S) &\mapsto (5, \epsilon) \\
\delta(1, b_1, S) &\mapsto (6, S) & \delta(6, a_0, S) &\mapsto (7, \epsilon) \\
\delta(1, a_0, S) &\mapsto (9, \epsilon) & \delta(7, a_0, S) &\mapsto (8, \epsilon) \\
\delta(2, a_0, S) &\mapsto (3, \epsilon) & \delta(9, a_0, S) &\mapsto (10, \epsilon)
\end{align*}
\]

Figure 4.7: Transition diagram of the (deterministic) PDA accepting the prefix notations of trees \( p_1, p_2 \) and \( p_3 \) from Example 13

Having constructed a PDA \( M \) accepting the linear notations of the patterns, we may now proceed to construct a searching SMPDA in the same way as in the case of only one pattern. The new, searching PDA is constructed using Algorithm 30 and accepts the language \( L = L_T.M \) where \( L_M \) is the language accepted by \( M \) and \( L_T = \{ x | x \in \Sigma^*, ac(x) > 0 \} \cup \{ \epsilon \} \).

**Example 14** The subtree matching PDA, constructed by Algorithm 30 for the trees from Example 13, is the nondeterministic SMPDA \( M = (Q,A,\{S\},\delta,0,S,F) \), where \( Q = \bigcup_{i=0}^{10} i \), \( F = \{5,8,10\} \) and the mapping \( \delta \) is defined as follows:
Figure 4.8: Transition diagram of the (deterministic) PDA accepting the prefix notations of trees $p_1$, $p_2$ and $p_3$ from Example 14

$$\delta(0, a_2, S) \rightarrow (0, SS) \quad \delta(1, a_0, S) \rightarrow (9, \varepsilon)$$
$$\delta(0, b_1, S) \rightarrow (0, S) \quad \delta(2, a_0, S) \rightarrow (3, \varepsilon)$$
$$\delta(0, b_0, S) \rightarrow (0, \varepsilon) \quad \delta(3, a_0, S) \rightarrow (4, \varepsilon)$$
$$\delta(0, a_0, S) \rightarrow (0, \varepsilon) \quad \delta(4, b_0, S) \rightarrow (5, \varepsilon)$$
$$\delta(0, a_2, S) \rightarrow (1, SS) \quad \delta(6, a_0, S) \rightarrow (7, \varepsilon)$$
$$\delta(1, a_2, S) \rightarrow (2, SS) \quad \delta(7, a_0, S) \rightarrow (8, \varepsilon)$$
$$\delta(1, b_1, S) \rightarrow (6, S) \quad \delta(9, a_0, S) \rightarrow (10, \varepsilon)$$

The transition diagram of the resulting PDA is illustrated in Figure 4.8.

**Algorithm 30: MULTIPLE-TREE-SEARCH**

**Input**: PDA $M = (Q, A = (\Sigma, \varphi), \{S\}, \delta, 0, S, F)$ constructed using Algorithm 29 which accepts language $L_M$

**Output**: A PDA accepting language $L = L_T \cdot L_M$, where $L_T = \{ x \mid x \in \Sigma^*, ac(x) > 0 \} \cup \{\varepsilon\}$

1. **foreach** $a \in \Sigma$ **do**
2. $\delta(0, a, S) \rightarrow (0, S^{\varphi(a)})$
3. $M \leftarrow (Q, A, \{S\}, \delta, 0, S, F)$

The SMPDA constructed using Algorithm 30 is an input-driven PDA and thus can be transformed to an equivalent deterministic PDA using Algorithm 27.
which resembles the process of subset construction (see Lemma 1). We conclude with an example to present the equivalent deterministic PDA of the SMPDA constructed in Example 14.

**Example 15** By applying Algorithm 27 on the SMPDA $M$ constructed in Example 14 we obtain a deterministic SMPDA $M' = (Q, A, \{S\}, \delta, \{0\}, S, F)$ which is equivalent to $M$, where $Q = \{\{0\}, \{0, 1\}, \{0, 1, 2\}, \{0, 6\}, \{0, 9\}, \{0, 3, 9\}, \{0, 7\}, \{0, 10\}, \{0, 4, 10\}, \{0, 8\}, \{0, 5\}\}$, $F = \{\{0, 4, 10\}, \{0, 5\}, \{0, 8\}, \{0, 10\}\}$ and the mapping $\delta$ is defined as follows:

\[
\begin{align*}
\delta(\{0\}, a_0, S) &\mapsto (0, \varepsilon) & \delta(\{0, 9\}, b_1, S) &\mapsto (\{0\}, S) \\
\delta(\{0\}, b_0, S) &\mapsto (0, \varepsilon) & \delta(\{0, 9\}, a_2, S) &\mapsto (\{0, 1\}, SS) \\
\delta(\{0\}, b_1, S) &\mapsto (0, S) & \delta(\{0, 3, 9\}, a_0, S) &\mapsto (\{0, 4, 10\}, \varepsilon) \\
\delta(\{0\}, a_2, S) &\mapsto (\{0, 1\}, SS) & \delta(\{0, 3, 9\}, b_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 1\}, a_0, S) &\mapsto (\{0, 9\}, \varepsilon) & \delta(\{0, 3, 9\}, b_1, S) &\mapsto (\{0\}, S) \\
\delta(\{0, 1\}, b_0, S) &\mapsto (\{0\}, \varepsilon) & \delta(\{0, 3, 9\}, a_2, S) &\mapsto (\{0, 1\}, SS) \\
\delta(\{0, 1\}, b_1, S) &\mapsto (\{0, 6\}, S) & \delta(\{0, 7\}, a_0, S) &\mapsto (\{0, 8\}, \varepsilon) \\
\delta(\{0, 1\}, a_2, S) &\mapsto (\{0, 1\}, SS) & \delta(\{0, 7\}, b_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 1, 2\}, a_0, S) &\mapsto (\{0, 3, 9\}, \varepsilon) & \delta(\{0, 7\}, b_1, S) &\mapsto (\{0\}, S) \\
\delta(\{0, 1, 2\}, b_0, S) &\mapsto (\{0\}, \varepsilon) & \delta(\{0, 7\}, a_2, S) &\mapsto (\{0, 1\}, SS) \\
\delta(\{0, 1, 2\}, b_1, S) &\mapsto (\{0, 6\}, S) & \delta(\{0, 10\}, a_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 1, 2\}, a_2, S) &\mapsto (\{0, 1\}, SS) & \delta(\{0, 10\}, b_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 4, 10\}, a_0, S) &\mapsto (\{0\}, \varepsilon) & \delta(\{0, 10\}, b_1, S) &\mapsto (\{0\}, S) \\
\delta(\{0, 4, 10\}, b_0, S) &\mapsto (\{0, 5\}, \varepsilon) & \delta(\{0, 10\}, a_2, S) &\mapsto (\{0, 1\}, SS) \\
\delta(\{0, 4, 10\}, b_1, S) &\mapsto (\{0\}, S) & \delta(\{0, 8\}, a_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 4, 10\}, a_2, S) &\mapsto (\{0, 1\}, SS) & \delta(\{0, 8\}, b_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 6\}, a_0, S) &\mapsto (\{0, 7\}, \varepsilon) & \delta(\{0, 8\}, b_1, S) &\mapsto (\{0\}, S) \\
\delta(\{0, 6\}, b_0, S) &\mapsto (\{0\}, \varepsilon) & \delta(\{0, 8\}, a_2, S) &\mapsto (\{0, 1\}, SS) \\
\delta(\{0, 6\}, b_1, S) &\mapsto (\{0\}, S) & \delta(\{0, 5\}, a_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 6\}, a_2, S) &\mapsto (\{0, 1\}, SS) & \delta(\{0, 5\}, b_0, S) &\mapsto (\{0\}, \varepsilon) \\
\delta(\{0, 9\}, a_0, S) &\mapsto (\{0, 10\}, \varepsilon) & \delta(\{0, 5\}, b_1, S) &\mapsto (\{0\}, S) \\
\delta(\{0, 9\}, b_0, S) &\mapsto (\{0\}, \varepsilon) & \delta(\{0, 5\}, a_2, S) &\mapsto (\{0, 1\}, SS)
\end{align*}
\]

The transition diagram of $M'$ is illustrated in Figure 4.9
In this section we will deal with the problem of tree template matching for ranked tree structures. We will first provide the definition of the problem that is to be solved. We follow by introducing a naive solution to the problem, and then we propose optimal algorithms. Note that, in the rest of this section, we will only consider the postfix notation of trees. This is because by iterating the nodes of the subject tree \( t \) in a postorder traversal, there is only one subtree associated with each node \( v \), that is the subtree rooted at node \( v \). Therefore, at
each node, we may report at most 1 subtree matching the given tree template. On the contrary, if the preorder traversal was used, each right-most leaf node of some subtree of $t$ may be the right-most leaf node of more than one subtrees of $t$, and therefore, when reading a leaf node $v$, there may be more than one subtrees that match the given tree template. However, using the described PDA approach, only the occurrence of the smallest subtree would be kept in the pushdown store.

**Definition 62 (Set of all trees)** Given a ranked alphabet $A = (\Sigma, \varphi)$, $T(A)$ denotes the set of linear postfix notations of all trees over $A$, and is defined as follows:

$$T(A) = \{ x \mid x \in \Sigma^+, ac(x) = 0, ac(y) \geq 1, \ x = zy, \ y, z \in \Sigma^+ \}$$

We also introduce a new, nullary symbol $S$, not in $\Sigma$, serving as a placeholder for any tree $t$, where $\text{post}(t) \in T(A)$. In the rest of the text we will refer to this symbol as the *don’t care* symbol. We denote the set $\Sigma \cup \{S\}$ as $\Sigma_S$, and define $A_S = (\Sigma_S, \varphi_S)$, where:

$$\varphi_S(a) = \begin{cases} \varphi(a) & : a \in \Sigma \\ 0 & : a = S \end{cases}$$

**Definition 63 (Tree pattern)** Given a ranked alphabet $A_S = (\Sigma_S, \varphi_S)$ and the set of all trees $T(A_S)$, a tree pattern is any tree in $T(A_S)$.

**Definition 64 (Tree template)** Tree templates are the elements of set $T(A_S) \setminus T(A)$, i.e. trees having at least one don’t care node.

**Definition 65 (Tree template matching)** A tree template $p$ over a ranked alphabet $A_S = (\Sigma_S, \varphi_S)$ with $k$ occurrences of the nullary placeholder symbol $S$ matches a subject tree $t$ in $T(A)$ at node $v$, if there exist trees $t_1, t_2, \ldots, t_k$ in $T(A)$, such that the tree $p'$, obtained by substituting $t_i$ with the $i$-th occurrence of $S$ in $p$, is equal to the subtree of $t$ rooted at $v$. Two trees are equal if, for example, their postfix notations are equal strings.

While not necessary in general, a new identifier can be encoded for each node of the subject tree, based on its attributed (such as label) and rank. These identifiers, along with the arity of the respective nodes, form the ranked alphabet. In this way, the case when the tree consists of nodes having the same label but different arity, can be easily handled.

**Problem 3** Given a subject tree $t$ and a tree template $p$, find the positions of all subtrees of $t$ (including $t$) that match $p$. 
4.2.1 A naive approach

In Section 4.1.1 we have presented a naive algorithm (see Algorithm 24) for subtree matching when the subject tree and tree pattern are given in their prefix notation. We will now present a naive tree template matching approach which, in contrary to Algorithm 24, will be taking as argument a subject tree and a tree template in their postfix, rather than prefix, notation. This is due to the fact that the linear algorithm we will be presenting in the following sections is based on the postfix notation.

**Algorithm 31: Template-Matching-Ranked-Naive**

| Input   | The postfix notation $post(t) = x[1\ldots n]$ of the subject tree and $post(p) = y[1\ldots m]$ of the tree template |
| Output  | The positions in $t$ where $p$ is matched |

1. $V[1\ldots n] \leftarrow \text{Subtree-Size-Array-Postfix}$
2. For $i \leftarrow n$ downto $m$ do
   3. $match \leftarrow \text{true}$
   4. $r \leftarrow i$
   5. For $j \leftarrow m$ downto 1 do
      6. If $y[j] = S$ then
         7. $r \leftarrow r - V[r]$
      8. Else
         9. If $y[j] \neq x[r]$ then
            10. $match \leftarrow \text{false}$
            11. Break
         12. $r \leftarrow r - 1$
   13. If $match = \text{true}$ then Output($i$)

Again, as in Section 4.1.1, a sliding window mechanism is used for the naive approach. The size of the window is $|p|$. The algorithm scans the postfix notation of the subject tree from right to left with this sliding window which represents the template. The nodes of the template and the corresponding nodes in the subject tree are checked for equality. In case a placeholder symbol $S$ is matched in the template, then the subtree located at the corresponding node in the subject tree is skipped by subtracting from the current position the size of that subtree and which is stored in the precomputed array $V$.

Hence, we obtain the following Lemma:

**Lemma 10** Algorithm 31 solves Problem 3 in time $O(nm)$, where $n = |t|$ and $m = |p|$.
Proof Similarly as in the proof of Lemma 6, it is obvious that the algorithm consists of reading the postfix notation \( x \) of \( t \) and at each step does at most \( m \) comparisons, where \( m \) is the size of the pattern. The complexity can be clearly shown with an example where the postfix notation of \( t \) is \( x = a_0a_1^{n-1} \) and the template’s postfix notation is \( y = Sa_1^{m-1} \), in which case the naive algorithm carries out exactly \((n - m + 1)m\) comparisons. \( \square \)

4.2.2 Tree template matching algorithm

After presenting the necessary definitions, we are in a position to present an algorithm for tree template matching based on PDA. The algorithm preprocesses the tree template once by computing the so-called match-sets, which are required for the construction of a PDA matching the given tree template. The constructed PDA then reads the postfix notation of the subject tree and matches each read subtree with the corresponding subtrees of the tree template. Indication that a read subtree matches the tree template is provided by the final state of the PDA.

The rest of this section is divided in three parts: first, we formally introduce the notion of match-sets; then, we show a method for computing match-sets; finally, the algorithm for preprocessing the tree template is presented.

4.2.2.1 Match-sets

Definition 66 (Set of subtrees) Given a tree \( t \) such that \( \text{post}(t) = x[1 \ldots m] \) over a ranked alphabet \( A_S = (\Sigma_S, \varphi_S) \), the set of subtrees of \( t \) is the set \( \text{Sub}(t) \) consisting of the postfix notations of all subtrees of \( t \), and is formally defined as:

\[
\text{Sub}(t) = \{ x \mid \text{post}(t) = yxz, \ y, z \in \Sigma^*, \ x \in \Sigma^+, \ x \neq S \}
\]

such that Theorem 2 holds for each \( x \in \text{Sub}(t) \).

We are now in a position to formally define the notion of match-sets. Each tree \( t \in T(A_S) \) can be mapped to a set consisting of all subtrees of the given tree template \( p \) that match \( t \). We call this particular set a match set.

Definition 67 (Match-set) Given a tree template \( p \) over \( A_S = (\Sigma_S, \varphi_S) \), a match-set is the mapping:

\[
\mu : T(A) \rightarrow R
\]

where \( R \subseteq \mathcal{P}(\text{Sub}(P) \cup \{S\}) \), and is defined as:

1. For each \( a \in \Sigma \), where \( \varphi(v) = 0 \):

\[
\mu(a) = \begin{cases} 
\{a, S\} : & a \in \text{Sub}(P) \\
\{S\} : & a \notin \text{Sub}(P)
\end{cases}
\]
2. For each \( x = \text{post}(t_1)\text{post}(t_2) \ldots \text{post}(t_q) a, a \in \Sigma, \varphi(a) = q, x \in T(A) \),

\[
\mu(x) = \{S\} \cup \{ y \mid y = \text{post}(t'_1) \ldots \text{post}(t'_q)v \land y \in \text{Sub}(P) \land \text{post}(t'_i) \in \mu(\text{post}(t_i)) \}
\]

In general terms, \( T(A) \) is the domain of the match-set function, while \( R \) is the range of the mapping. For simplicity, throughout the text we will refer to match-sets as the range \( R \) of the defined mapping.

**Definition 68 (Tree Types)** Let \( p \) and \( p' \) be subtrees of the tree template \( P \) over an alphabet \( \Sigma_S \), i.e. \( p, p' \in \text{Sub}(P) \). \( p \) is inconsistent with \( p' \) (\( p \sim p' \)) if there is no tree \( t \in T(\Sigma) \) such that \( p, p' \in \mu(t) \), consistent in the opposite case. \( p \) and \( p' \) are independent (\( p \sim p' \)) if there are trees \( t_1, t_2, t_3 \in T(\Sigma) \), such that \( p \in \mu(t_1), p' \notin \mu(t_2), \) \( p' \in \mu(t_2), p' \notin \mu(t_3) \). \( p \) subsumes \( p' \) (\( p > p' \)) if, for all \( t \in T(\Sigma), p \in \mu(t) \Rightarrow p' \in \mu(t) \).

**Example 16** Let \( p_1, p_2, p_3, p_4 \) be the trees illustrated in Figure 4.10, with postfix notations \( \text{post}(p_1) = a_0S\text{a}_2, \text{post}(p_2) = b_0\text{S}a_2, \text{post}(p_3) = Sb_0\text{a}_2, \text{post}(p_4) = S\text{Sa}_2 \). \( p_1 \) and \( p_2 \) are inconsistent (\( p_1 \sim p_2 \)) as nodes \( a \) and \( b \) cannot be matched at the same position. Trees \( p_1 \) and \( p_3 \) are independent (\( p_1 \sim p_3 \)), since there exist trees \( t_1, t_2, t_3 \), where \( \text{post}(t_1) = a_0a_0\text{a}_2, \text{post}(t_2) = b_0b_0\text{a}_2, \text{post}(t_3) = a_0b_0\text{a}_2 \), holding that \( p_1 \in \mu(t_1) \) and \( p_3 \notin \mu(t_1), p_3 \in \mu(t_2) \) and \( p_1 \notin \mu(t_2) \), and \( p_1, p_3 \in \mu(t_3) \). Finally, \( p_1 > p_4, p_2 > p_4, p_3 > p_4 \).

**Lemma 11 (Size of match-sets)** Given a tree template \( P \), the upper theoretical bound of the number of possible match-sets is \( O(2^{|P|}) \), and is reached if and only if there exist \( \Theta(2^{|P|}) \) sets of pairwise independent subtrees in the tree template.

**Proof** Let \( t \) be a 2^h-ary tree of height \( h+1 \), consisting of 2^h pairwise independent balanced binary trees as the direct successors (children) of the root node \( u \) of \( t \). The postfix notation of \( t \) is the string \( \text{post}(t) = \text{post}(t_{1,1}^h)\text{post}(t_{2,1}^h) \ldots \text{post}(t_{2h,1}^h)a \), where \( \varphi(a) = 2^h \) and

\[
\text{post}(t_{j,k}^0) = \begin{cases} S & : 1 \leq j \leq 2^h, 1 \leq k \leq 2^h, j \neq k \\ a_0 & : 1 \leq j \leq 2^h, 1 \leq k \leq 2^h, j = k \end{cases}
\]

\[
\text{post}(t_{j,k}^i) = \text{post}(t_{j,2k-1}^{i-1})\text{post}(t_{j,2k}^{i-1})a', 1 \leq i \leq h, 1 \leq j \leq 2^h, 1 \leq k \leq 2^{h-i}, \varphi(a') = 2
\]

Let \( Q_i \in \mathcal{P}\{1, 2, \ldots, 2^h\} \) be distinct elements from the powerset of the set of numbers 1 to \( 2^h \), where \( 1 \leq i < 2^{2h} \). For each such set \( Q_i \), we construct a
balanced binary tree \( t_{Q_i}^h \) of height \( h \), such that \( \text{post}(t_{Q_i}^h) = \text{post}(t_{Q_i}^{h-1})\text{post}(t_{Q_i}^{h-1})a'' \), where \( \varphi(a'') = 2 \) and

\[
\text{post}(t_{Q_i,k}^0) = \begin{cases} 
  b_0 & : 1 \leq k \leq 2^h, k \notin Q_i \\
  a_0 & : 1 \leq k \leq 2^h, k \in Q_i
\end{cases}
\]

\[
\text{post}(t_{Q_i,k}^j) = \text{post}(t_{Q_i,2k-1}^{j-1})\text{post}(t_{Q_i,2k}^{j-1})u, 1 \leq j \leq h, 1 \leq k \leq 2^{h-j}, \varphi(u) = 2
\]

Tree \( t_{i,1}^h \) matches \( t_{Q_i}^h \) if and only if \( i \in Q_j \), where \( 1 \leq i \leq 2^h \) and \( 1 \leq j < 2^h \). The tree template \( t \) consists of \( 2^h \times (2^{h+1} - 1) + 1 \) nodes, while there are \( 2^{2^h} - 1 \) sets \( Q_i \) representing possible match-sets. Hence, \( O(2^{|I|}) \) different match-sets may exist.

Let \( p_1 \) and \( p_2 \) be subtrees of a tree template \( p \), and \( p_1 \mid p_2 \). For any tree \( t \in T(\Sigma) \), it holds that \( p_1 \in \mu(t) \Rightarrow p_2 \notin \mu(t) \) and \( p_2 \in \mu(t) \Rightarrow p_1 \notin \mu(t) \). Let \( p_3 \) and \( p_4 \) be subtrees of tree template \( p \), and \( p_3 > p_4 \). Then for any tree \( t \in T(\Sigma) \) it holds that \( p_3 \in \mu(t) \Rightarrow p_4 \in \mu(t) \). Let \( I = (p_1, p_2, \ldots, p_\ell) \) be a set of pairwise independent subtrees. Let \( \mathcal{P}(I) \) be the powerset of \( I \). Then there might exist at most \( 2^\ell - 1 \) trees \( t_1, t_2, \ldots, t_{2^\ell-1} \), such that there exists a set \( q_i \in \mathcal{P}(I) \), where \( q_i \subseteq \mu(t_i) \) and it holds that \( g \notin \mathcal{P}(I) \), for all \( g \in \mathcal{P}(I), g \neq q_i, 1 \leq i < 2^\ell \). \( \square \)
The proof of Lemma 11 is illustrated in Figure 4.11.

![Diagram of Lemma 11 proof](image)

**Definition 69 (Combination of tree templates)** The combination of two pair-wise independent tree templates $p$ and $p'$ (denoted by $p \circ p'$) with $\text{post}(p) =$
post(p_1) post(p_2) \ldots post(p_{\phi(v)}) v and post(p') = post(p'_1) post(p'_2) \ldots post(p'_{\phi(v)}) v, respectively, is the tree t where post(t) = post(t_1) post(t_2) \ldots post(t_{\phi(v)}) is defined as

\[
post(t_j) = \begin{cases} 
  post(p_j) : & post(p_j) > post(p'_j) \lor post(p_j) = post(p'_j) \\
  post(p'_j) : & post(p'_j) > post(p_j) \\
  post(p_j) \circ post(p'_j) : & post(p_j) \sim post(p'_j)
\end{cases}
\]

The combination of a set \( \mu = (t_1, t_2, \ldots, t_{\sigma(\mu)}) \) of pairwise independent tree templates is defined as \( C(\mu) = t_1 \circ t_2 \circ \ldots \circ t_{\sigma(\mu)} \).

### 4.2.2.2 Computing match-sets

We briefly present a naive approach for computing the match-sets of a given tree template \( p \). Trees can be of the types from Definition 68. Subtrees of \( p \), which are pairwise inconsistent, cannot be in the same match-set (or form one), as no tree from \( T(\Sigma) \) can be matched by both. Therefore a match-set can be either of the following two forms:

1. A set \( X \) which consists of a tree \( t \in Sub(p) \) and all trees \( t_1, t_2, \ldots, t_k \in Sub(p) \) such that \( t > t_i \) for all \( 1 \leq i \leq k \)

2. A set \( X = I \cup I' \), where \( I = (t_1, t_2, \ldots, t_k) \) is a set of pairwise independent subtrees of \( p \) and \( I' = (t'_1, t'_2, \ldots, t'_\ell) \) is the set of subtrees of \( p \) such that for each \( t'_i, 1 \leq i \leq \ell \), it holds that \( t_j > t'_i \) for all \( 1 \leq j \leq k \). In other words, \( I' = \{ t' \mid y = C(I) \wedge y > t', t' \in Sub(p) \} \)

The sets of pairwise independent subtrees of \( p \) can be computed with an iterative approach of constructing, for each tree \( t \in Sub(p) \), a set \( I_t \) of all subtrees of \( p \) which are independent with \( t \), i.e. \( t' \sim t \), for all \( t' \in I_t \). The sets of pairwise independent subtrees of \( p \) are the subsets of sets \( I_t \cup \{ t \} \), whose elements are pairwise independent.

The total number of match-sets is therefore \( O(\max(|p|, r)) \), where \( r \) is the number of sets consisting of pairwise independent subtrees of \( p \). In case no sets of pairwise independent subtrees of \( p \) exist, the number of match-sets is \( O(|p|) \). In the worst case, as proved in Lemma 11, the number of sets of pairwise independent subtrees is \( O(2^{|p|}) \), and therefore the number of match-sets is \( O(2^{|p|}) \).

### 4.2.2.3 PDA for ranked tree template matching

The method for tree pattern matching works in a similar fashion as the finite automata based algorithms for string pattern matching: Given a tree template
Algorithm 32: Nondeterministic-Tree-Template-Matching-PDA

Input: Tree template post(p) = post(p₁)post(p₂)…post(pₙ(v))v over A
Output: Nondeterministic PDA M = ({q_I, q_F}, A, Γ, δ, {q_I}, ε, {q_F})

1. Let Γ ← { {x} | x ∈ Sub(p) ∪ {S} }
2. For each a ∈ Σ, let qₐ Tₚ(a) ↦ₜ M qₐ T, where T = {S}
3. Let qₐ Xₚ(a) … X₂ X₁ ↦ₜ M qₐ X, where Xᵢ = {post(tᵢ)}, X = {post(t)}, for each post(t) = post(t₁)post(t₂)…post(tₚ(a))a ∈ Sub(p) \ {post(p)}
4. Let qₐ Xₚ(v) … X₂ X₁ ↦ₜ M qₐ X, where Xᵢ = {post(pᵢ)}, X = {post(p)}

p, Algorithm 32 constructs a nondeterministic PDA M that can match all occurrences of p in some subject tree t, by final state. The constructed PDA belongs to the class of height-deterministic PDA [93] as all of its configurations have a pushdown store of the same height, since for each transition performed, the pushdown store operations always place one symbol on the store, and the number of symbols removed is driven entirely by the input symbol (input-driven). Therefore, an equivalent deterministic PDA may be constructed.

Example 17 Let p be a tree template whose postfix notation post(p) is the string x = a₀Sₐ₀Sₐ₀a₂Sₐ₀a₂a₉. Its graphical representation is illustrated in Figure 4.12. Algorithm 32 constructs a nondeterministic PDA

\[ M = (\{q_I, q_F\}, A, Γ, δ, \{q_I\}, ε, \{q_F\}) \]

where the set Γ consists of the following pushdown store symbols

- \( T \mapsto S \)
- \( X₁ \mapsto a₀ \)
- \( X₂ \mapsto b₀ \)
- \( X₃ \mapsto X₁ T a₂ \)
- \( X₄ \mapsto T T a₂ \)
- \( X₅ \mapsto T X₂ a₂ \)
- \( X₆ \mapsto X₃ X₄ a₂ \)
- \( X₇ \mapsto X₆ X₅ a₂ \)

each of which represents (maps to) a concrete, specific subtree of p. PDA M matches all occurrences of p in some subject tree. The transition diagram of M is illustrated in Figure 4.13, while its transition table is shown in Table 4.3.

Algorithm 33 presents a novel method taking as input the nondeterministic PDA obtained from Algorithm 32, computes the match-sets, and constructs an equivalent deterministic PDA \( M_D \), serving as the tree template matcher.
Figure 4.12: Graphical representation of (a) tree template \( p \) from Example 17 and (b) pushdown store symbols in accordance to the corresponding subtrees of \( p \)

<table>
<thead>
<tr>
<th>( q_t )</th>
<th>( a_0 )</th>
<th>( b_0 )</th>
<th>( a_1 )</th>
<th>( a_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_t</td>
<td>\varepsilon \rightarrow T )</td>
<td>( q_t</td>
<td>\varepsilon \rightarrow X_1 )</td>
<td>( q_t</td>
</tr>
<tr>
<td>( q_t</td>
<td>\varepsilon \rightarrow T )</td>
<td>( q_t</td>
<td>TX_1 \rightarrow X_3 )</td>
<td></td>
</tr>
<tr>
<td>( q_t</td>
<td>TT \rightarrow T )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_t</td>
<td>TX \rightarrow X_4 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_t</td>
<td>X_2T \rightarrow X_5 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_t</td>
<td>X_1X_3 \rightarrow X_6 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_t</td>
<td>X_5X_6 \rightarrow X_7 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Transition table of nondeterministic PDA constructed in Example 17

Lemma 12 Given a nondeterministic PDA constructed using Algorithm 32 by preprocessing a given tree template \( P \), Algorithm 33 constructs an equivalent deterministic PDA matching all occurrences of \( P \) in a subject tree \( T \).

Theorem 8 For tree template \( p \), the space needed for preprocessing is \( O(2^{\|p\|}) \).

Proof In general, there can be \( O(2^{\|p\|}) \) pushdown store symbols (see Lemma 11). The PDA can be implemented as a table and thus \( O(2^{\|p\| \times k} \times |A|) \) space is required for preprocessing, where \( k = \max\{\varphi(x) : \forall x \in A\} \). □

Theorem 9 The deterministic template matching PDA constructed using Algorithms 32 and 33 matches all occurrences of a tree template \( p \) in a subject tree \( T \) in time \( O(|T|) \).
Figure 4.13: Transition diagram of nondeterministic tree template matching PDA from Example 17

Algorithm 33: Deterministic-Tree-Template-Matching-PDA

<table>
<thead>
<tr>
<th>Input</th>
<th>Nondeterministic PDA $M = (Q, A, \Gamma, \delta, q_I, \epsilon, F)$ constructed from tree template $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Deterministic PDA $M' = (Q', A, \Gamma', \delta', {q_I}, \epsilon, F')$</td>
</tr>
</tbody>
</table>

1. for each $t \in \text{Sub}(p)$ do  
2.     for each $X \in \mathcal{P}(I_t)$ in ascending order of cardinality do  
3.         $\Gamma' \leftarrow \Gamma' \cup \{X \cup \{t\} \cup S_t \cup \{t' : y = C(X \cup \{t\} \cup S_t) \land t' \in I_t \land y > t'\}\}$  
4.     Let $Q' \leftarrow \{\{q_I\}, \{q_I, q_F\}\}$ and $F' \leftarrow \{\{q_I, q_F\}\}$  
5.     For each $x \in \Sigma$, let $q' \gamma'_1 \gamma'_2 \cdots \gamma'_{\varphi(x)} \xrightarrow{x}{M'} p'X'$ for all $\gamma'_i \in \Gamma'$, where $q', p' \in Q'$, $1 \leq i \leq \varphi(x)$, $p' = \bigcup_{j=1}^{l} \{p_j\}$ and $X' = \bigcup_{j=1}^{l} \{\theta_j\}$, such that there exist $l$ transitions of form $q_j \gamma_1 \gamma_2 \cdots \gamma_{\varphi(x)} \xrightarrow{x}{M} p_j \theta_j$, $\gamma_i \in \Gamma'$, $q_j \in q'$

**Proof** For each input symbol $x$ of the subject tree, $\varphi(x) + 1$ operations are performed: $\varphi(x)$ pop operations from the pushdown store and one push. The sum of arities of all nodes of the input tree $t$ is $n - 1$ (number of edges). Thus, $n - 1$ pop and $n$ push operations are performed, a total of $2n - 1$ operations. □

The transformation of the nondeterministic PDA constructed using Algorithm 32 to an equivalent deterministic PDA starts with the computation of all possible match-sets which are defined on the basis of subsumption and independence sets, and can be computed with the method presented in section 4.2.2.2. The match-sets correspond to the pushdown store symbols of the deterministic tree template matching PDA and are computed in lines 1-3 of Alg. 33. The list of all possible match-sets resulting from the transformation of the nondeterministic
PDA computed in Example 17 is presented in Table 4.4.

\[
\begin{align*}
Y_1 &= \{T\} & Y_6 &= \{X_4, X_5, X_3, T\} \\
Y_2 &= \{X_1, T\} & Y_7 &= \{X_1, X_5, T\} \\
Y_3 &= \{X_2, T\} & Y_8 &= \{X_6, X_4, T\} \\
Y_4 &= \{X_4, T\} & Y_9 &= \{X_7, X_4, T\} \\
Y_5 &= \{X_4, X_3, T\} &
\end{align*}
\]

Table 4.4: List of computed match-sets resulting from the transformation of the nondeterministic PDA from Example 17 to a deterministic PDA

The resulting deterministic tree pattern matching PDA consists of 2 states and 92 transitions (1 for the input symbol \(a_0\), 1 for \(b_0\), 9 for \(a_1\) and 81 for \(a_2\), see Theorem 8). The initial and final state always have the same transitions as the final state is only used as an indication for matching, and thus can share the same transition table which is presented in Table 4.5.
<table>
<thead>
<tr>
<th>$a_0$</th>
<th>$b_0$</th>
<th>$a_1$</th>
<th>$a_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_I</td>
<td>Y_1\rightarrow Y_1$</td>
<td>$q_I</td>
<td>Y_2\rightarrow Y_1$</td>
</tr>
<tr>
<td>$q_I</td>
<td>Y_6\rightarrow Y_1$</td>
<td>$q_I</td>
<td>Y_7\rightarrow Y_1$</td>
</tr>
</tbody>
</table>

Transitions originating from both states $q_I$ and $q_F$

$q_I|\varepsilon \rightarrow Y_2$

Table 4.5: State $q_I$ denotes the initial state of the deterministic PDA while state $q_F$ denotes the final state. The transitions are shared between the two states. The tree template is matched when taking either of the two transitions $q_F|Y_6Y_8 \rightarrow Y_9$ or $q_F|Y_7Y_8 \rightarrow Y_9$.

91
4.3 Tree template matching in unranked trees

The material presented in this section was published in part in [25].


In this section an algorithm for tree template matching in unranked trees will be presented. First, we will define the problem that will be solved. We follow by introducing a naive solution to that problem, and then we propose an optimal, table-driven algorithm. Note that, in the rest of this section, we will only consider the postfix bar notation of trees.

4.3.1 Problem definition

Analogously to Definition 62 we define the set of all unranked trees.

**Definition 70 (Set of all unranked trees)** Given an alphabet \( \Sigma \), \( T_u(\Sigma) \) denotes the set of all unranked trees over \( \Sigma \), and is defined as follows:

\[
T_u(\Sigma) = \{ x \mid x \in (\Sigma \cup \{\uparrow\})^+, bc(x) = 0, bc(y) < 0, x = zy, y, z \in (\Sigma \cup \{\uparrow\})^+ \}
\]

Similarly as in the case of ranked trees, we introduce a new, nullary symbol \( S \), not in \( \Sigma \), serving as a placeholder for any tree \( t \), where \( \text{pstb}(t) \in T_u(\Sigma) \). We denote the set \( \Sigma \cup \{S\} \) as \( \Sigma_S \).

The definitions for tree pattern,

**Definition 71 (Unranked tree pattern)** Given a ranked alphabet \( \Sigma \) and the set of all trees \( T_u(\Sigma) \), a tree pattern is any tree in \( T_u(\Sigma) \).

**Definition 72 (Unranked tree template)** Tree templates are the elements of the set \( T_u(\Sigma_S) \setminus T(S) \), i.e. trees having at least one “don’t care” node.

**Definition 73 (Unranked tree template matching)** A tree template \( p \) over an alphabet \( \Sigma \) with \( k \) occurrences of the unary placeholder symbol \( S \) matches a subject tree \( t \) in \( T_u(\Sigma) \) at node \( v \), if there exist trees \( t_1, t_2, \ldots, t_k \) in \( T_u(\Sigma) \), such that the tree \( p' \), obtained by substituting \( t_i \) with the \( i \)-th occurrence of \( S \) in \( p \), is equal to the subtree of \( t \) rooted at \( v \). Two trees are equal if, for example, their postfix bar notations are equal strings.

**Problem 4** Given a subject tree \( t \) and a tree template \( p \), find the positions of all subtrees of \( t \) (including \( t \)) that match \( p \).
4.3.2 A Naive approach

We present a naive algorithm for tree template matching in unranked trees. The algorithm is similar to the one presented in section 4.2.1 for the case of ranked trees. It takes as arguments a subject tree and a tree template in their postfix bar notations.

Algorithm 34: Template-Matching-Unranked-Naive

Input: The postfix notation \( pstb(t) = x[1..n] \) of the subject tree and \( pstb(p) = y[1..m] \) of the tree template

Output: The positions in \( t \) where \( p \) is matched

1. \( V[1..n] \leftarrow \text{Postfix-Bar-Subtree-Size-Array} \)
2. for \( i \leftarrow n \) downto \( m \) do
3. \( \text{match} \leftarrow \text{true} \)
4. \( r \leftarrow i \)
5. for \( j \leftarrow m \) downto \( 1 \) do
6. \( \text{if} \ y[j] = S \text{ then} \)
7. \( r \leftarrow r - 2V[r] \)
8. \( \text{else} \)
9. \( \text{if} \ y[j] \neq x[r] \text{ then} \)
10. \( \text{match} \leftarrow \text{false} \)
11. \( \text{break} \)
12. \( r \leftarrow r - 1 \)
13. \( \text{if} \ \text{match} = \text{true} \text{ then} \ \text{OUTPUT}(i) \)

In a similar way as in section 4.2.1, a sliding window mechanism is used for the naive approach. The size of the window is \( 2|p| \). The algorithm scans the postfix bar notation of the subject tree from right to left with this sliding window which represents the template. The nodes of the template and the corresponding nodes in the subject tree are checked for equality. In case a placeholder symbol \( S \) is matched in the template, then the subtree located at the corresponding node in the subject tree is skipped by subtracting from the current position the size of that subtree twice (to include the bars) — which is stored in the precomputed array \( V \).

Hence, we obtain the following Lemma:

Lemma 13 Algorithm 34 solves Problem 4 in time \( \mathcal{O}(nm) \), where \( n = |t| \) and \( m = |p| \).
Proof The algorithm reads the postfix bar notation $x$ of the input subject tree $t$ from right to left, and at each of the $n$ steps, where $n = |x|$, does at most $m$ comparisons, where $m$ is the size of the postfix bar notation of tree template $p$. The complexity is clearly shown if we take the example where the postfix bar notation of $t$ is $x = \uparrow^{n/2} ab^{n/2} - 1$, and the tree template’s postfix bar notation is $y = \uparrow^{m/2} Sb^{m/2} - 1$, in which case the naive algorithm carries out exactly $(n - m + 1)m$ comparisons.

4.3.3 Tree template matching algorithm

In this section, we present a table-driven algorithm for tree template matching in unranked ordered trees. The algorithm is divided in two phases: the preprocessing phase and the searching phase. Given a tree template, the preprocessing phase computes the so-called match-sets, which are required for constructing the action table. This table is the core part of the searching phase. Let us note that the preprocessing phase is done only once for a given tree template, while the searching phase can be carried out on a number of subject trees without the need of preprocessing the tree template again. During the searching phase, the postfix bar notation of the subject tree is read symbol by symbol, from left to right (corresponds to bottom-up reading of the subject tree), and matches each read subtree with the corresponding subtrees of the tree template. Each time a read subtree matches the tree template, the algorithm outputs a pair consisting of the starting and ending position of the subtree.

We divide the rest of this section in four subsections: first, we formally introduce the notion of match-sets; then, we show a method for computing these match-sets; finally, we present an algorithm describing the construction of the action table and an algorithm describing the searching phase.

4.3.3.1 Match-sets

Definition 74 (Set of subtrees) Given a tree $t$ in its postfix bar notation $x[1..m] = pstb(t)$ over an alphabet $\Sigma_S$, the set of subtrees of $t$ is the set $Sub(t)$ consisting of the postfix bar notations of all subtrees of $t$, and is formally defined as

$$Sub(t) = \{ x \mid pstb(t) = yxz, \ y, z \in \Sigma_S^+, \ x \in \Sigma_S^+, \ x \neq \uparrow S \}$$

such that Theorem 4 holds for each $x \in Sub(t)$.

The set of subtrees can be trivially computed by a simple traversal of the postfix bar notation of tree $t$ from left to right.
We are now in a position to formally define the notion of match-sets. Each tree $t \in T(\Sigma)$ can be mapped to a set consisting of all subtrees of the given tree template $p$ that match $t$ at its root node. We call this particular set a match-set.

**Definition 75 (Match-set)** Given a tree template $p$ over $\Sigma$, a match-set is the mapping

$$\mu : T(\Sigma) \rightarrow R$$

where $R \subseteq P(Sub(p) \cup \{S\})$, and is defined as

1. For each $x = \uparrow a$, where $a \in \Sigma$,
   $$\mu(x) = \{ \uparrow a, S \} : \uparrow a \in Sub(p)$$
   $$\{ S \} : \uparrow a \notin Sub(p)$$

2. For each $x = \uparrow pstb(t_1) \, pstb(t_2) \, \ldots \, pstb(t_q) \, a$, $x \in T(\Sigma)$, $a \in \Sigma$,
   $$\mu(x) = \{ S \} \cup \{ y \mid y = \uparrow pstb(t'_1) \, \ldots \, pstb(t'_q) \, a \land y \in Sub(p) \land pstb(t'_i) \in \mu(pstb(t_i)) \}$$

Using general terms, $T(\Sigma)$ is the domain of the match-set function, while $R$ is the range of the mapping. For simplicity, throughout the paper we will refer to match-sets as the range $R$ of the defined mapping.

In direct analogy to Example 16, we present the tree types from Definition 68 for unranked trees with an example.

**Example 18** Let $p_1, p_2, p_3, p_4$ be trees, where $pstb(p_1) = \uparrow \uparrow a \uparrow S c$, $pstb(p_2) = \uparrow \uparrow b \uparrow S c$, $pstb(p_3) = \uparrow \uparrow e \uparrow S \uparrow b c$, $pstb(p_4) = \uparrow \uparrow a \uparrow S \uparrow S c$. $p_1$ and $p_2$ are inconsistent ($p_1 \not\sim p_2$) as nodes $a$ and $b$ cannot be matched at the same position. Trees $p_1$ and $p_3$ are independent ($p_1 \sim p_3$), since there exist trees $t_1, t_2, t_3$, where $pstb(t_1) = \uparrow \uparrow a \uparrow a c$, $pstb(t_2) = \uparrow \uparrow b \uparrow b c$, $pstb(t_3) = \uparrow \uparrow e \uparrow b c$, holding that $p_1 \in \mu(t_1)$ and $p_3 \notin \mu(t_1)$, $p_3 \in \mu(t_2)$ and $p_1 \notin \mu(t_2)$, and $p_1, p_3 \in \mu(t_3)$. Finally, $p_1 > p_4, p_2 > p_4, p_3 > p_4$. A graphical illustration of the trees is presented in Figure 4.14.

**Lemma 14 (Size of match-sets)** Given a tree template $p$, the upper theoretical bound of the number of possible match-sets is $O(2^{|p|})$, and is reached if and only if there exist $O(2^{|p|})$ sets of pairwise independent subtrees in the tree template.

**Proof** The proof is the same as the proof of Lemma 11. □
Figure 4.14: Graphical representation of trees $p_1, p_2, p_3, p_4$ with postfix bar notations $\text{pstb}(p_1) = \uparrow\uparrow\uparrow a \uparrow Sc$, $\text{pstb}(p_2) = \uparrow\uparrow\uparrow b \uparrow Sc$, $\text{pstb}(p_3) = \uparrow\uparrow\uparrow S \uparrow bc$ and $\text{pstb}(p_4) = \uparrow\uparrow\uparrow S \uparrow bc$.

**Definition 76 (Combination of tree templates)** The combination of two consistent tree templates $p$ and $p'$ ($p \circ p'$) having postfix bar notations $\text{pstb}(p) = \uparrow \text{pstb}(p_1) \text{pstb}(p_2) \ldots \text{pstb}(p_k) v$ and $\text{pstb}(p') = \uparrow \text{pstb}(p'_1) \text{pstb}(p'_2) \ldots \text{pstb}(p'_l) v$, respectively, is the tree $t$ where $\text{pstb}(t) = \uparrow \text{pstb}(t_1) \text{pstb}(t_2) \ldots \text{pstb}(t_k) v$ is defined as

$$\text{pstb}(t_j) = \begin{cases} \text{pstb}(p_j) & : \text{pstb}(p_j) > \text{pstb}(p'_j) \lor \text{pstb}(p_j) = \text{pstb}(p'_j) \\ \text{pstb}(p'_j) & : \text{pstb}(p'_j) > \text{pstb}(p_j) \\ \text{pstb}(p_j) \circ \text{pstb}(p'_j) & : \text{pstb}(p_j) \sim \text{pstb}(p'_j) \end{cases}$$

The combination of a set $X = (t_1, t_2, \ldots, t_{\sigma(X)})$ of tree templates is defined as

$$C(X) = t_1 \circ t_2 \circ \ldots \circ t_{\sigma(X)}$$

**4.3.3.2 Computing match-sets**

We briefly present a naive approach for computing the match-sets of a given tree template $p$. Trees can be of the types from Definition 68. Subtrees of $p$, which are pairwise inconsistent, cannot be in the same match-set (or form one), as no
tree from $T(\Sigma)$ can be matched by both. Therefore a match-set can be either of the following two forms:

1. A set $X$ which consists of a tree $t \in \text{Sub}(p)$ and all trees $t_1, t_2, \ldots, t_k \in \text{Sub}(p)$ such that $t > t_i$ for all $1 \leq i \leq k$

2. A set $X = I \cup I'$, where $I = (t_1, t_2, \ldots, t_k)$ is a set of pairwise independent subtrees of $p$ and $I' = (t'_1, t'_2, \ldots, t'_\ell)$ is the set of subtrees of $p$ such that for each $t'_i$, $1 \leq i \leq \ell$, it holds that $t_j > t'_i$ for all $1 \leq j \leq k$. In other words, $I' = \{ t' \mid y = C(I) \land y > t', \ t' \in \text{Sub}(p) \}$

The sets of pairwise independent subtrees of $p$ can be computed with an iterative approach of constructing, for each tree $t \in \text{Sub}(p)$, a set $I_t$ of all subtrees of $p$ which are independent with $t$, i.e. $t' \sim t$, for all $t' \in I_t$. The sets of pairwise independent subtrees of $p$ are the subsets of sets $I_t \cup \{ t \}$, whose elements are pairwise independent.

The total number of match-sets is therefore $O(\max(|p|, r))$, where $r$ is the number of sets consisting of pairwise independent subtrees of $p$. In case no sets of pairwise independent subtrees of $p$ exist, the number of match-sets is $O(|p|)$. In the worst case, as proved in Lemma 14, the number of sets of pairwise independent subtrees is $O(2^{|p|})$, and therefore the number of match-sets is $O(2^{|p|})$.

4.3.3.3 Constructing the action table

To briefly explain how the matching is achieved, and therefore justify the purpose of the action table, assume the searching phase algorithm has read the nodes $v_1, v_2, \ldots, v_r$ (in that order) of the subject tree, which represent the root nodes of adjacent subtrees (siblings) $t_1, t_2, \ldots, t_r$, respectively. The computed match-sets matching $t_1, t_2, \ldots, t_r$, say $X_1, X_2, \ldots, X_r$, respectively, are then placed on top of the stack in that order, i.e. $X_r$ is the element on top of the stack. Now assume that the root node $u$, which is the root of some subtree $t$, and whose immediate children subtrees are $t_1, t_2, \ldots, t_r$, is read. The symbols on top of the stack, corresponding to the match-sets $X_1, X_2, \ldots, X_r$ matching the children (subtrees) of $u$ are popped from the stack. Then the match-set $Y$ matching $t$ is selected from the action table as the element indexed by the sequence $X_r, X_{r-1}, \ldots, X_1$ and node $u$. Finally, $Y$ is placed on top of the stack.

Algorithm 35 constructs the action table in the following way: first, the match-sets for the given tree template $p$ are computed in lines 3-6 as described in Section 4.3.3.2.

For each match-set $X$, we denote by $x$ the postfix notation of the tree resulting from the combination of trees in $X$. $x$ has the form $\uparrow \text{pstub}(t_1)\text{pstub}(t_2)\ldots \text{pstub}(t_k)v$. We construct all possible sequences of match-sets, say $S_1, S_2, \ldots, S_\ell$, where each
Algorithm 35: Action-Table-Construction

**Input:** Tree template \( p \)

**Output:** Action table

1. Let \( R \leftarrow \emptyset \)
2. \( \triangleright \) Compute the match-sets
3. **foreach** set \( I \) consisting of pairwise independent subtrees of \( p \) **do**
   4. \( R \leftarrow R \cup \{ I \cup \{ S \cup \{ t' \mid y = C(I) \land y > t', \forall t' \in \text{Sub}(p) \} \} \} \)
5. **foreach** \( t \in \text{Sub}(p) \) **do**
   6. \( R \leftarrow R \cup \{ \{ t, S \} \cup \{ t' \mid t > t', \forall t' \in \text{Sub}(p) \} \} \)
7. **foreach** \( X \in R \) **do**
   8. Let \( x \leftarrow C(X) \) be of form \( \uparrow \uparrow \text{pstb}(t_1) \text{pstb}(t_2)\ldots \text{pstb}(t_k)v \)
   9. Let \( S_1, S_2, \ldots S_\ell \in R^+ \) be all sequences of match-sets of the form \( S_i = X_1 X_2 \ldots X_k \), where \( X_j \in R \), such that \( \text{pstb}(t_j) \subseteq X_j, 1 \leq j \leq k \)
   10. **for** \( i \leftarrow 1 \) **to** \( \ell \) **do** Insert \( (Q[v], (\text{reverse}(S_i), X)) \)
11. **foreach** \( a \in \Sigma \) **do**
12. Construct trie \( T_a \) from the elements \( (u, v) \) of \( Q[a] \) such that the node indexing \( u \) is labeled as \( v \). All other new nodes created by the indexing are labeled as \( W \). In case \( u \) is already indexed, with the corresponding node labeled as \( w \), replace \( w \) with \( v \) only if \( w \subset v \)
13. Table\([a] \leftarrow T_a \)

\( S_i, 1 \leq i \leq \ell \), is of the form \( S_i = X_1 X_2 \ldots X_k \), where \( X_j, 1 \leq j \leq k \), is a match-set such that \( \text{pstb}(t_j) \subseteq X_j \). In an array of queues \( Q \), we insert the pairs \( (\text{reverse}(Y_i), X) \) in \( Q[v] \), where \( \text{reverse}(S_i) \) is the reversed \( S_i \), according to the root element \( v \) of the trees in match-set \( X \) (lines 7-10).

For each \( a \in \Sigma \), we construct a trie by indexing the elements \( (u, v) \in Q[a] \) by \( u \), and place \( v \) as the node content at which \( u \) is indexed (lines 11-13). The new nodes created by the indexing are labeled as \( W \), where \( W \) corresponds to the match-set consisting only of \( S \) (i.e. matches any tree), while the existing nodes remain unchanged. In case \( u \) already appears in the index, its node content \( w \) is replaced with \( v \) only if it holds that \( w \subset v \). This is due to the fact that in addition to the trees in \( w \), \( v \) also consists of trees subsuming the ones in \( w \), while \( u \) can also be matched by these additional trees.

Example 19 Construct the action table for the tree template \( p \) in Fig. 4.15.

Let \( t_1, t_2, t_3, t_4 \) be the trees having postfix bar notations \( \text{pstb}(t_1) = \uparrow \uparrow S \uparrow S \uparrow ba \), \( \text{pstb}(t_2) = \uparrow \uparrow S \uparrow b \uparrow Sa \), \( \text{pstb}(t_3) = \uparrow \uparrow b \uparrow S \uparrow Sa \) and \( \text{pstb}(t_4) = \uparrow b \). Then,
Match set | Children shape | Reversed children
---|---|---
$X_1$ | ** B | B **
$X_2$ | *B* | *B*
$X_3$ | B ** | ** B
$X_{1,2}$ | *BB | BB *
$X_{1,3}$ | B * B | B * B
$X_{2,3}$ | BB * | *BB
$X_{1,2,3}$ | BBB | BBB

Table 4.6: The sequences of match-sets obtained by lines 7-10 of Algorithm 35. Note that for the sake of saving space, the symbol * represents an arbitrary match-set

Symbol | Action
---|---
$a$ | ![Diagram of $a$]
$b$ | ![Diagram of $b$]
$f$ | ![Diagram of $f$]

Table 4.7: The action table constructed by lines 12-14 of Algorithm 35

the set of all subtrees of $p$ is $\text{Sub}(p) = \{ pstb(t_1), \ pstb(t_2), \ pstb(t_3), \ pstb(t_4), \ pstb(p) \}$. Fours sets consisting of pairwise independent subtrees of $p$ can be constructed:

$I_1 = \{ t_1, t_2 \}$, $I_2 = \{ t_2, t_3 \}$, $I_3 = \{ t_1, t_3 \}$, $I_4 = \{ t_1, t_2, t_3 \}$

According to lines 3-6 of Algorithm 35, the following match-sets are created:
Figure 4.15: Tree template $p$ from Example 1 having bar notation $pstb(p) = \uparrow\uparrow S \uparrow S \uparrow b \uparrow S \uparrow b \uparrow S \uparrow b \uparrow S \uparrow S f$

$$
\begin{align*}
X_{1,3} &= \{pstb(t_1), pstb(t_3), S\} \\
X_3 &= \{pstb(t_3), S\} \\
X_1 &= \{pstb(t_1), S\} \\
X_{1,2,3} &= \{pstb(t_1), pstb(t_2), pstb(t_3), S\} \\
M &= \{pstb(p), S\}
\end{align*}
$$

We construct an additional match-set $W = \{S\}$ which matches all trees that cannot be matched by any subtree of the given tree template $p$.

We are now in a position to proceed with the construction of the action table. The sequences of match-sets constructed from lines 7-10 of Algorithm 35 are shown in Table 4.6. The constructed action table is illustrated in Table 4.7.

**Theorem 10** Given a tree template $p$, the space required for preprocessing $p$ is $\mathcal{O}(\max(|p|, r))$, where $r$ is the number of sets consisting of pairwise independent subtrees of $p$. In the worst case the space required can be $\mathcal{O}(2^{|p|})$ while in case no sets of pairwise independent subtrees of $p$ exist it is $\mathcal{O}(|p|)$.

**Proof** In the worst case, as shown in Lemma 14, the number of sets of pairwise independent subtrees can be $\mathcal{O}(2^{|p|})$. The space required for storing the action table is thus $\mathcal{O}(2^{|p|})$. In case the tree template does not consist of pairwise independent subtrees, the number of match-sets is linear to the size of $p$, and thus the action table requires only $\mathcal{O}(|p|)$ space.

### 4.3.4 Searching phase

We are now in a position to present the algorithm for searching a tree template in a subject tree. That is, given a subject tree $t$ and the action table obtained by preprocessing a tree template $p$, find all the occurrences of $p$ in $t$. 

100
Algorithm 36 reads the subject tree from left to right, symbol by symbol. It uses a stack and the action table to drive the actions needed to be taken each time a symbol is read. The input symbols can be split in two categories: bars and nodes.

We distinguish among two cases when reading an input symbol: in case a bar is read, a stack symbol \( \odot \) is placed on top of the stack with an attribute containing the position of the bar in the input text (lines 2-3); in case a node is read, we consecutively remove all stack symbols from the top of the stack, until a \( \odot \) is found. We denote the first \( \odot \) on top of the stack as \( \odot \), representing the bar corresponding to the read node. Let \( X \) denote the sequence of the removed stack symbols. We retrieve the stack symbol \( Y \), which is the label of the node indexing \( X \) in the trie, located in the action table for the particular input symbol \( x_i \), and place \( Y \) on top of the stack (lines 4-9). In case \( Y \) is equal to \( M \)—the match-set containing \( p \)—the subtree of \( t \), having as its root node the currently read symbol, matches the tree template. Thus, we output the starting and ending positions of the factor corresponding to the bar notation of the matching subtree. The starting position is the attribute stored in the removed \( \odot \), and the ending position is the position of the currently read symbol (line 10).

**Theorem 11** Algorithm 36 matches all occurrences of a tree template \( p \) in a subject tree \( t \) in time \( \Theta(|t|) \).
Figure 4.16: Diagram of the subject tree $t$ having bar notation $pstb(t) = \uparrow \uparrow \uparrow b \uparrow \uparrow \uparrow b \uparrow b \uparrow b a \uparrow \uparrow a \uparrow b a \uparrow \uparrow a \uparrow a \uparrow \uparrow b a \uparrow \uparrow b a \uparrow \uparrow b \uparrow b a \uparrow \uparrow b a \uparrow \uparrow b \uparrow b a \uparrow a \uparrow a f a f a f$

**Proof** The postfix bar notation of a tree $t$ has exactly $2 \times |t|$ symbols ($|t|$ bars and $|t|$ nodes). For each bar read, the algorithm places one symbol on the stack, i.e. a total of $|t|$ operations. For each alphabet symbol $v$ read, the algorithm retrieves $r$ symbols from the stack, where $r$ is the number of the subtrees rooted at $v$. Thus, exactly $|t| - 1$ symbols are retrieved from the stack in total. The algorithm also queries the tries from the action tables for each sequence of symbols read from the stack. The total query time is exactly $|t| - 1$. After each query is carried out, a symbol is placed on top of the stack, i.e. another $|t|$ operations. The total number of performed operations is thus $6 \times |t| - 2$. \hfill $\blacksquare$

**Example 20** Given the subject tree $t$ from Fig. 4.16 and the action table (see Table 4.7) obtained by preprocessing the tree template $p$ from Fig. 4.15, Table 4.8 demonstrates the searching phase.
Table 4.8: Dry run of the searching phase

<table>
<thead>
<tr>
<th>Pushdown store</th>
<th>Input</th>
<th>Pushdown store</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>↑↑↑</td>
<td>$BB \circ X_{1,2}$</td>
<td>↑</td>
</tr>
<tr>
<td>$\circ \circ \circ \circ$</td>
<td>$b$</td>
<td>$\circ BB \circ X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$B \circ \circ \circ \circ$</td>
<td>$b$</td>
<td>$WBB \circ X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$\circ \circ \circ B \circ \circ \circ$</td>
<td>$b$</td>
<td>$X_{2,3}X_{1,2}$</td>
<td>$\uparrow\uparrow$</td>
</tr>
<tr>
<td>$B \circ \circ B \circ \circ \circ$</td>
<td>$b$</td>
<td>$\circ X_{2,3}X_{1,2}$</td>
<td>$b$</td>
</tr>
<tr>
<td>$BB \circ \circ B \circ \circ \circ$</td>
<td>$b$</td>
<td>$B \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$\circ BB \circ \circ B \circ \circ \circ$</td>
<td>$b$</td>
<td>$WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow\uparrow$</td>
</tr>
<tr>
<td>$BBB \circ \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$b$</td>
</tr>
<tr>
<td>$X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$B \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$\circ \circ X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ B \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$W \circ X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$WB \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$\circ W \circ X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$b$</td>
<td>$\circ WB \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$b$</td>
</tr>
<tr>
<td>$BW \circ X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$BBW \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$\circ BW \circ X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$WBW \circ X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ \circ X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$W \circ X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$\circ \circ X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ W \circ X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$b$</td>
</tr>
<tr>
<td>$W \circ X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$BW \circ X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$\circ W \circ X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ BW \circ X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$b$</td>
</tr>
<tr>
<td>$WW \circ X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$BBW \circ X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$\circ WW \circ X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$b$</td>
<td>$X_{1,2,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$BWW \circ X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ \circ X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$b$</td>
</tr>
<tr>
<td>$X_{1}X_{2}X_{1,2,3} \circ B \circ \circ \circ$</td>
<td>$f$</td>
<td>$B \circ X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$WB \circ \circ \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$\circ \circ X_{2}X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$\circ WB \circ \circ \circ \circ \circ$</td>
<td>$b$</td>
<td>$WB \circ X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>$BW \circ \circ \circ \circ \circ \circ$</td>
<td>$a$</td>
<td>$\circ WB \circ X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$X_{1,2} \circ \circ \circ \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>$WWB \circ X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$\circ \circ \circ \circ X_{1,2}X_{1,3} \circ \circ \circ \circ \circ$</td>
<td>$b$</td>
<td>$X_{3}X_{1,2}X_{1,3} \circ \circ WB \circ X_{2,3}X_{1,2}$</td>
<td>$f$</td>
</tr>
<tr>
<td>$B \circ X_{1,2} \circ \circ \circ \circ \circ \circ$</td>
<td>$\uparrow$</td>
<td>match</td>
<td>$MWB \circ X_{2,3}X_{1,2}$</td>
</tr>
<tr>
<td>$\circ B \circ X_{1,2} \circ \circ \circ \circ \circ \circ$</td>
<td>$b$</td>
<td>match</td>
<td>$X_{3}X_{2,3}X_{1,2}$</td>
</tr>
</tbody>
</table>

$\varepsilon$
Chapter 5

Tree indexing

“An expert is a man who has made all the mistakes which can be made in a very narrow field”
— Niels Bohr (Danish Physicist, 1885-1962)

The material presented in this section was published in part in [49, 50].


5.1 Introduction

Indexing of data is an important operation due to the fact that currently we are experiencing a cataclysm of electronic data in many formats. Due to the massive amounts of data it is necessary to correctly and efficiently index it, in order to avoid data duplicity and to be able to effectively query and retrieve this data.

Concerning text indexing, many methods have been proposed in the past, from prefix trees, through suffix trees and various types of DAWGs, to suffix arrays and its variations.

In this chapter we will focus on tree indexing. Given a tree structure we will present methods which are based on pushdown automata and are directly
analogous to the finite automata based solutions for text indexing. Specifically, we will show two new data structures, the *subtree PDA*, which is directly analogous to the suffix automaton used for indexing text, and the *tree pattern PDA*. Using the proposed data structures we will present solutions to the following two problems.

**Problem 5** Compute and classify all repeating subtrees within a given subject tree.

**Problem 6** Construct an indexing structure over a subject tree $t$ so that one can effectively query whether a given subtree $p$ exists in tree $t$.

The following text is divided in three sections. First, a brief introduction on text indexing is given. Then a new type of PDA called *Subtree Pushdown Automaton* is examined in the following section and last, a linear algorithm for computing all repeating subtrees is presented in the last section.

### 5.2 Text indexing structures

Indexing text has been of paramount importance in the last few decades. The main purpose of an index is to provide efficient routines for answering queries related to the content of some fixed text. Given some text $x$ drawn from an alphabet $\Sigma$, an index on $x$ can be considered as an abstract data type whose basic set is the set of factors of $x$, and that possesses operations giving access to information relative to those factors [34]. The notion is analogue to the notion of index of a book that refers to the text from selected keywords. In the following text we rather consider what is called a generalised index, in which all the factors of the text are present.

The indexing data structure usually consists of four main operations. They concern some (query) string $q$ that we search for inside $x$: membership, position, number of occurrences, and list of positions. This list is generally extended in real applications, according to the nature of the data represented by $x$, in order to produce documentary search systems. The four mentioned operations, however, constitute the technical basis from which larger query systems can be developed.

In the following text a brief description of the three classical indexing structures – the suffix tree, suffix automaton and suffix array – is given, paying particular attention to the suffix automaton, which will be used further in the text. Note that all mentioned structures actually index the suffixes of string $x$, and provide a straightforward way to determine whether or not $q$ is a suffix of $x$. This, however, can be determined, probably with greater efficiency, directly from $x$, by comparing $x[[x|−|q|+1..|x|]]$ with $q$. An important property which renders these data structure useful is that every factor of $x$ is a prefix of some suffix of
and therefore the data structures can determine whether or not $q$ is a factor of $x$. Furthermore, with some polishing, the mentioned structures can also be used to locate the first occurrences of $q$ in $x$, and in fact to locate all occurrences of $q$ in $x$. Thus, they can also be useful for solving other problems, such as the computation of repetitions in strings [11, 62, 101].

For more information on these indexing structures or indexing in general, see [13, 34, 35, 100].

5.2.1 Suffix trees

A suffix tree is a trie-like data structure (see section 2.6.3) that presents all suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations. In particular, a suffix tree for string $x$ (sometimes called a subword tree for $x$ [10]) is just a Patricia tree (see Section 2.6.3) on the set $\text{Suff}(x)$ of suffixes of $x$ (including $\epsilon$, the empty string/suffix). A suffix tree for a string $x$ consists of exactly $n + 1$ terminal nodes (leaves) and at most $n$ internal nodes called branch nodes. Thus, there are at most $2n + 1$ nodes and at most $2n$ edges in a suffix tree. The storage required for each edge label can be reduced to a constant by replacing the substring on each edge by two integers specifying a starting position of the label in the string $x$, and its length. It follows that the suffix tree is a desirable data structure in the sense that it requires only $\Theta(n)$ storage.

The concept of suffix trees was first introduced as a position tree by Weiner in 1973 [106]. The construction was greatly simplified by McCreight in 1976 [87] and subsequently by Ukkonen in 1995 [58, 103] for ordered alphabets. These algorithms are all linear-time for constant-size alphabet, and have a worst-case running time of $O(n \log n)$ in general. In 1997, Farach gave the first suffix tree construction algorithm that is optimal for any type of alphabet [45] In particular, this is the first linear-time algorithm for strings drawn from indexed alphabets. Other variants of suffix tree construction include a modified version of Weiner’s algorithm [22] which permits searches for the reversed substrings of the processed string, and [9]

Figure 5.1 illustrates the suffix tree constructed for the string $x = \text{banana}$. Each edge consists of a label which is a substring $x$ and the concatenation of labels of the edges forming a path from the root node to a leaf forms a distinct suffix of $x$. Leaves contain a number which denotes the starting position of the suffix they index in string $x$ (in this case $x$ starts from 0 and not from 1). Note that the arcs drawn with dashed lines, called suffix links, are a key feature for older linear-time construction algorithms such as the on-line construction algorithm presented by Ukkonen, and are not necessarily part of the suffix tree. Most newer algorithms, which are based on Farach’s algorithm, dispense with suffix links. In a complete
Figure 5.1: The suffix tree for string $x = \text{banana}$

suffix tree, all internal non-root nodes have a suffix link to another internal node. If the path from the root to a node spells the string $ay$, where $a \in \Sigma$ ($\Sigma$ is the alphabet) and $y \in \Sigma^*$, it has a suffix link to the internal node representing $y$.

Having briefly introduced the concept of suffix tree, we are now in a position to present how suffix trees could be used as indexing structures. Assume a suffix tree $T_x$ is constructed from a string $x$. First of all, determining whether or not a given string $q$ is a substring of the indexed string $x$ is as simple as checking whether there exists a path of edges originating from the root node of the suffix tree, such that $q$ is a prefix of the concatenation of their labels. The location of all occurrences of $q$ in $x$ can also be trivially obtained by locating the leaves of the subtree of $T_x$ whose root is the node to which the mentioned path of edges originating from the root node lead to (i.e. the shortest path of edges such that $q$ is a prefix of the concatenation of their labels). The numbers stored in these leaves are the starting positions of all occurrences of $q$ in $x$.

Overall, membership (i.e. whether $q$ is a factor of $x$) can be tested in time $O(|q|)$ and locating all occurrences of $q$ in $x$ can be done in time $O(|q| + \text{occ})$, where $\text{occ}$ is the number of occurrences of $q$ in $x$. These results, together with the linear time and space construction complexities of the suffix tree (when using Farach’s algorithm) constitute the suffix tree as a “desirable” indexing structure. For a review on suffix trees and their construction algorithms see [34, 35, 100].

### 5.2.2 Suffix automaton

The *suffix automaton* of a string $x$, is the minimal automaton that accepts the set of suffixes of $x$, i.e. $\text{Suff}(x)$. It also often called a *Directed Acyclic Word Graph* (DAWG). In the following text, the term suffix automaton and DAWG will be used interchangeably. The DAWG was first introduced in [16, 32]. The invention
of the DAWG stems from the observation that suffix trees have a tendency to contain isomorphic subtrees that are repeated. This leads to redundancy since corresponding nodes in the repeated subtrees give rise to corresponding edges with identical labels, and the only difference between the isomorphic copies are the labels of the leaf nodes. The DAWG was introduced essentially as a mechanism for eliminating this redundancy. Figure 5.2 illustrates the suffix automaton constructed for the string $x = \text{banana}$. Doubly-circled states denote final states. The transitions in dashed are not part of the automaton, and denote the suffix links. While suffix links are not necessarily part of the suffix automaton, they, however, are an essential feature for on-line construction algorithms of suffix automata.

The surprising fact about suffix automata is that its size is linear to the length of the indexed string $x$ even though the number of factors of $x$ can be quadratic. Particularly, the suffix automaton for a string $x$ consists of at most $2|\times| - 1$ states and at most $3|\times| - 4$ arcs. For a detailed proof of the presented numbers see [34, 100].

An interesting variation of the suffix automaton is the Compacted Directed Acyclic Word Graph (CDAWG). A CDAWG is an index structure which preserves some features of both suffix tree and DAWGs, and requires less space than both of them. It is constructed from a DAWG by eliminating all internal nodes of degree 2 (those with a parent and just a single child). As an example, Figure 5.3 illustrates the CDAWG constructed for the string $x = \text{banana}$. The first algorithm to construct the CDAWG was presented by Blumer et al. [16]. Although the algorithm runs in linear time, its main drawback is the construction of the DAWG as an intermediate structure. A solution to this matter was provided by Crochemore and Vérin [37], who designed a linear-time, off-line algorithm, based on McCreight’s suffix tree construction algorithm [87], to construct the CDAWG for a string directly, without the need for constructing an intermediate DAWG. Since their algorithm is off-line, the input string has to be known beforehand, and when a new character is appended to the input string, the CDAWG must be rebuilt from scratch. A linear-time, on-line algorithm, based on Ukkonen’s suffix tree construction algorithm [103], was designed [66]. An optimized implementa-
tion of CDAWG is presented in [64]

\[ \text{Figure 5.3: Compacted Suffix automaton (CDAWG) for string } x = \text{banana} \]

### 5.2.3 Suffix array

In computer science, a suffix array is an array of integers giving the starting positions of suffixes of a string in lexicographical order. For example, the string \( x = \text{banana} \) consists of the following suffixes:

\[
\begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 \\
\text{aab} & \quad \text{abaab} & \quad \text{abaab} & \quad \text{baabaab} & \quad \text{baab} & \quad \text{baabaab} & \quad \text{aab} & \quad \text{baabaab}
\end{align*}
\]

which are listed in lexicographical order. Thus, the suffix array of \( x \) is \( \sigma_x = 63741852 \).

Suffix arrays were first introduced by Manber and Myers [84, 85]. For more information on suffix arrays, time and space complexities and for the current state of art, see [71, 72, 74, 75]

### 5.3 Ranked tree indexing structures

This section describes some new types of pushdown automata called the *subtree PDA* and the *tree pattern PDA*, first introduced in [50, 67, 68], for ranked trees given in their prefix notation. A subtree PDA constructed for a prefix notation of some tree \( t \) accepts all strings corresponding to prefix notations of subtrees of \( t \), where as a tree pattern PDA accepts all strings corresponding to prefix notations of tree patterns which match some prefix notation of subtrees of \( t \).
Both the subtree and tree pattern PDA form a complete index for the tree structure $t$ of size $n$ they are constructed for, and the searching phase for locating all occurrences of a subtree or a tree pattern, respectively, of size $m$ is performed in time linear to $m$ and does not depend in any way on $n$. The total size of the deterministic subtree PDA is linear to $n$. As for the deterministic tree pattern PDA we will prove that there might exist cases for which the total size of the PDA can be exponential to the size of the indexed tree $t$. We will divide this section in three parts. First, we will formally introduce the subtree PDA and methods for its construction and usage. Then, we will introduce a new data structure called the treepop PDA which accepts all tree patterns that match the complete indexed tree $t$ (not its subtrees). The treepop PDA combined with the subtree PDA will then form the tree pattern PDA which will be presented after. The last part will be a method for computing and classifying repetitions in tree structures by using the subtree PDA.

5.3.1 Subtree pushdown automata

We will now formally introduce a new type of data structure, the subtree PDA. Both linear notations, prefix and postfix, can be used when building the subtree PDA. We will assume, for the rest of this section, that the prefix notation is used as the linear notation with which the tree structures are represented. Everything that is presented in this section can be applied, with minor changes, to the postfix notation.

Definition 77 (Subtree pushdown automaton) A subtree pushdown automaton for $x$, where $x$ is the prefix notation of a tree $t$, is a PDA that accepts all prefix notations of all subtrees of $t$.

We will first present a method for constructing a non-deterministic subtree PDA for the prefix notation $x$ of some tree $t$. This PDA can be constructed by extending the PDA $\overline{M}$ obtained by Algorithm 25 (see Section 4.1.2).

Algorithm 37 describes the extension of the PDA $\overline{M}$ resulting from algorithm TREE-MATCH-PDA by appending extra transitions for each symbol $x[i]$, $1 \leq i \leq |x|$, originating at the initial state and directing to the corresponding state $i$. In other words, the new PDA accepts the prefix notations of all subtrees of $t$ by empty pushdown store. The non-determinism resides on the fact that there may be several transitions having the same label that originate from state 0 and lead to different states.

Example 21 The subtree PDA for the tree illustrated in Figure 3.1 constructed over its prefix notation $x = a_2a_2a_0a_1a_0a_1a_0$ with the use of Algorithm 37 is the...
Algorithm 37: SUBTREE-PDA

Input: Prefix notation $x = \text{pref}(t)$ of some ranked tree $t$ over ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$

Output: A non-deterministic subtree PDA $M$ for $x$

$M = (\cup_{i=0}^{\mid x \mid}, \mathcal{A}, \{S\}, \delta', 0, S, \{|x|\}) \leftarrow \text{TreeMatch-PDA}(x, \mathcal{A})$

for $i \leftarrow 2 \text{ to } \mid x \mid$ do

Let $\delta(0, x[i], S) \mapsto (i, S^{x[i]})$

$Q \leftarrow \{ i \mid 0 \leq i \leq |t| \}$

$M \leftarrow (Q, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$

$PDA M = (Q, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$ where $Q = \{0, 1, \ldots, 7\}$ and the mapping $\delta$ is defined as follows:

$\delta(0, a_2, S) \mapsto (1, SS) \quad \delta(5, a_1, S) \mapsto (6, S)$
$\delta(1, a_2, S) \mapsto (2, SS) \quad \delta(6, a_0, S) \mapsto (7, \varepsilon)$
$\delta(2, a_0, S) \mapsto (3, \varepsilon) \quad \delta(0, a_2, S) \mapsto (2, SS)$
$\delta(3, a_1, S) \mapsto (4, S) \quad \delta(0, a_1, S) \mapsto (4, S)$
$\delta(4, a_0, S) \mapsto (5, \varepsilon) \quad \delta(0, a_0, S) \mapsto (3, \varepsilon)$
$\delta(0, a_0, S) \mapsto (5, \varepsilon) \quad \delta(0, a_1, S) \mapsto (6, S)$
$\delta(0, a_0, S) \mapsto (7, \varepsilon)$

The transition diagram of $M$ appears in Figure 5.4.

The non-deterministic subtree PDA belongs to the class of input-driven PDA and thus can be transformed to an equivalent deterministic PDA with the use of Algorithm 27.

Example 22 Let $M$ be the subtree PDA constructed from Example 21. By applying Algorithm 27 we obtain an equivalent deterministic PDA which is depicted in Figure 5.5.

Note that from the construction of the nondeterministic subtree PDA is always acyclic and therefore the contents of the pushdown store of the equivalent deterministic subtree PDA in particular states can be computed beforehand. Thus, the states that are reached by a sequence of symbols that do not form a valid prefix notation of some tree may be removed. Consider the deterministic PDA
Figure 5.4: Transition diagram of non-deterministic subtree PDA $M$ from Example 21 constructed for $x = a_2 a_2 a_0 a_1 a_0 a_1 a_0$

Figure 5.5: Transition diagram of the deterministic PDA $M$ from Example 22 constructed for $x = a_2 a_2 a_0 a_1 a_0 a_1 a_0$

constructed in Example 22 and which is illustrated in Figure 5.5. The string $a_0$ which is formed by the transition from state 0 to state $(3, 5, 7)$ is a valid prefix notation of a tree, and as this transition is the only one leading to that state, we may eliminate the outgoing transition of $(3, 5, 7)$ since the strings formed by following the path of outgoing transitions will never give a valid prefix notation of some tree. The string composed by the symbols of the path of transitions leading from the initial state (0) to state $(5, 7)$ is also a valid prefix notation of a tree and thus the pushdown store is emptied when reaching that state. Therefore, we can safely the transition leaving from that state since no valid prefix notation can be composed by following the symbols in the consecutive transitions. The deterministic subtree PDA obtained by eliminating these transitions is depicted in Figure 5.6.
Below is a sequence of transitions (trace) performed by the deterministic subtree PDA $M$ constructed in Example 22 for the tree $t$ with prefix notation $x = a_2a_2a_0a_1a_0a_1a_0$, when querying for a tree with prefix notation $x = a_1a_0$ (an unary with one leaf). The accepting state is $(5, 7)$, which means that there are two occurrences of the subtree with prefix notation $x$ in tree $t$, and that their rightmost leaves are nodes $a_0$ ($x[5]$) and $a_0$ ($x[7]$).

We conclude this section with a lemma on the size of the deterministic subtree PDA.

**Lemma 15 (Size of deterministic subtree PDA)** The deterministic subtree PDA constructed for the prefix notation $x$ of some ranked tree consists of at most $2|x| - 1$ states and at most $3|x| - 4$ transitions.

**Proof** The string $x$ is the prefix notation of a tree consisting of $|x|$ nodes. The number of distinct subtrees of $t$ is equal or smaller than $|x|$. The deterministic subtree PDA has only one pushdown store symbol $S$, and all its states and transitions correspond to the states and transitions, respectively, of the suffix automaton (see section 5.2.2) constructed for $x$. Therefore, the total size of the deterministic subtree PDA cannot be greater than the total size of the deterministic suffix automaton. This means that given a tree $t$ with $n$ nodes, the deterministic subtree PDA consists of at most $2n - 1$ states and at most $3n - 4$ transitions.
subtree PDA for $\text{pref}(t)$ has just one pushdown store symbol, fewer than $2n$ nodes and at most $3n - 4$ transitions.

5.3.2 Tree pattern pushdown automaton

Another important data structure for tree indexing is the tree pattern PDA. We will, though, first introduce another type of PDA, the treetop PDA which, combined with the subtree PDA form the tree pattern PDA.

**Definition 78 (Treetop pushdown automaton)** Let $t, r$ and $x$ be a tree, its root node and its prefix notation $\text{pref}(t)$, respectively. A treetop PDA for $x$ accepts all tree patterns in prefix notation which have $r$ as their root node and match tree $t$, by empty pushdown store.

We present the construction of the deterministic treetop PDA in Algorithm 38, for the prefix notation $x$ of some tree $t$. Similarly as with the subtree PDA, this PDA can be constructed by extending the PDA $M$ obtained by Algorithm 25 (see section 4.1.2).

**Algorithm 38**: Treetop-PDA

| Input : Prefix notation $x = \text{pref}(t)$ of some ranked tree $t$ over ranked alphabet $A$ |
| Output: A deterministic Treetop PDA $M$ for $x$ |

1 $V \leftarrow \text{PREFIX-SUBTREE-SIZE-ARRAY}$
2 $\triangleright \text{Construct the backbone of the PDA}$
3 $\text{for } i \leftarrow 1 \text{ to } |x| \text{ do}$
4 $\quad \text{Let } \delta(i - 1, x[i], S) \mapsto (i, S^{x[i]})$
5 $\text{for } i \leftarrow 2 \text{ to } |x| \text{ do}$
6 $\quad \text{Let } \delta(i - 1, S, S) \mapsto (i + V[i] - 1, \varepsilon)$
7 $Q \leftarrow \{ i \mid 0 \leq i \leq |x| \}$
8 $M \leftarrow (Q, A, \{ S \}, \delta, 0, S, \emptyset)$

**Example 23** The treetop PDA for the tree illustrated in Figure 3.1 constructed over its prefix notation $x = a_2a_2a_0a_1a_0a_1a_0$ with the use of Algorithm 38 is the PDA $M = (Q, A, \{ S \}, \delta, 0, S, \emptyset)$ where $Q = \{ 0, 1, \ldots, 7 \}$ and the mapping $\delta$ is defined as follows:
The transition diagram of $M$ appears in Figure 5.7.

\begin{align*}
\delta(0, a_2, S) &\rightarrow (1, SS) & \delta(5, a_1, S) &\rightarrow (6, S) \\
\delta(1, a_2, S) &\rightarrow (2, SS) & \delta(6, a_0, S) &\rightarrow (7, \varepsilon) \\
\delta(2, a_0, S) &\rightarrow (3, \varepsilon) & \delta(1, S, S) &\rightarrow (5, \varepsilon) \\
\delta(3, a_1, S) &\rightarrow (4, S) & \delta(2, S, S) &\rightarrow (3, \varepsilon) \\
\delta(4, a_0, S) &\rightarrow (5, \varepsilon) & \delta(3, S, S) &\rightarrow (5, \varepsilon) \\
\delta(4, S, S) &\rightarrow (5, \varepsilon) & \delta(5, S, S) &\rightarrow (7, \varepsilon) \\
\delta(6, S, S) &\rightarrow (7, \varepsilon) \\
\end{align*}

Figure 5.7: Transition diagram of deterministic treetop PDA $M$ from Example 23 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$

**Lemma 16** Given a tree $t$ and its prefix notation $x = \text{pref}(t)$, the PDA $M$ constructed by Algorithm 38 is a treetop PDA for $x$.

**Proof** Let $r$ be the root node of tree $t$. The constructed treetop PDA $M = (Q, A, \{S\}, \delta, 0, S, \emptyset)$ is a simple extension of the PDA constructed by Algorithm 25 which accepts the prefix notation of $t$. For the new, added transitions which read the nullary placeholder symbol $S$, it holds that

$$\delta(q_1, S, S) = (q_2, \varepsilon) \text{if and only if} (q_1, w, S) \vdash^+ M (q_2, \varepsilon, \varepsilon)$$

where $q_1, q_2 \in Q$ and $q_1$ is not the initial state 0. This matches with Theorem 1, so that the new added transitions reading $S$ correspond just to subtrees not containing the root $r$. Thus, the PDA $M$ accepts all tree patterns in prefix notation which contain the root $r$ and match the tree $t$.

We may now present and formally define the tree pattern pushdown automaton.
Definition 79 (Tree pattern pushdown automaton) Let \( t \) and \( x \) be a tree and its prefix notation \( \text{pref}(t) \), respectively. A tree pattern PDA for \( x \) accepts all tree patterns in prefix notation which match some subtree of \( t \) (including \( t \)).

The non-deterministic tree pattern PDA for trees in prefix notation is constructed by “combining” the subtree and the treetop PDA. Specifically, given a the prefix notation \( x \) of some tree \( t \), we construct a deterministic treetop PDA and then for each symbol (node) \( x[i] \) we add a transition labeled with \( x[i] \) that originates from the initial state 0 and leads to state \( i \), for all \( 2 \leq i \leq |x| \). In other words, the tree pattern PDA is a basic PDA constructed from Algorithm 25 with the additional transitions of the subtree and treetop PDA. The algorithm for the construction of a non-deterministic tree pattern PDA is described in Algorithm 39.

**Algorithm 39: Tree-Pattern-PDA**

- **Input**: Prefix notation \( x = \text{pref}(t) \) of some ranked tree \( t \) over ranked alphabet \( \mathcal{A} \)
- **Output**: A non-deterministic tree pattern PDA \( M \) for \( x \)

\[
\begin{align*}
V &\leftarrow \text{PREFIX-SUBTREE-SIZE-ARRAY} \\
M_1 &= (Q, \mathcal{A}, \{S\}, \delta_1, 0, S, \emptyset) \leftarrow \text{SUBTREE-PDA} \\
M_2 &= (Q, \mathcal{A}, \{S\}, \delta_2, 0, S, \emptyset) \leftarrow \text{TREETOP-PDA} \\
\delta &\leftarrow \delta_1 \cup \delta_2 \\
M &\leftarrow (Q, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)
\end{align*}
\]

Example 24 The tree pattern PDA for the tree illustrated in Figure 3.1 constructed over its prefix notation \( x = a_3a_2a_0a_1a_0a_1a_0 \) with the use of Algorithm 39 is the PDA \( M = (Q, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset) \) where \( Q = \{0, 1, \ldots, 7\} \) and the mapping \( \delta \) is defined as follows:
The transition diagram of $M$ appears in Figure 5.8.

Figure 5.8: Transition diagram of non-deterministic tree pattern PDA $M$ from Example 24 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$

**Lemma 17** Given a tree $t$ and its prefix notation $x = \text{pref}(t)$, the PDA $M$ constructed by Algorithm 39 is a tree pattern PDA for $x$.

**Proof** Let $r$ be the root node of tree $t$ and $\mathcal{A} = (\Sigma, \varphi)$ the ranked alphabet. The constructed PDA $M$ is a simple extension of the PDA constructed by Algorithm 38, which accepts all the prefix notations of all tree patterns having root
and match the tree $t$ by empty pushdown store. The PDA $M$ contains new added transitions of the form

$$\delta(0, x[i], S) = (i, S^{x[i]}), 2 \leq i \leq |x|$$

These transitions create a non-determinism which allows the root of the tree pattern that is to be accepted, to be matched by any node of tree $t$. Thus, the PDA $M$ accepts all prefix notations of tree patterns which match any subtree of $t$. □

Similarly as with the subtree PDA, the non-deterministic tree pattern PDA is an acyclic input-driven PDA, and therefore can be transformed to an equivalent deterministic tree pattern PDA by using Algorithm 27.

**Example 25** Let $M$ be the tree pattern PDA constructed from Example 24. By applying Algorithm 27 we obtain an equivalent deterministic PDA which is depicted in Figure 5.9. Note that the states and transitions forming paths that are not and cannot form a valid prefix notation of a tree pattern by following that path, have already been removed from the diagram.

![Transition diagram of the deterministic Subtree PDA](image)

Figure 5.9: Transition diagram of the deterministic Subtree PDA $M$ from Example 25 constructed for $x = a_2a_2a_0a_1a_0a_1a_0$ after eliminating the unnecessary transitions

**Lemma 18 (Size of deterministic tree pattern PDA)** Given the prefix notation $\text{pref}(t)$ of some tree $t$ with $n$ nodes, the deterministic tree pattern PDA may consist of $O(2^n)$ states and transitions in the worst case.
Proof Let $t$ be a $2^h$-ary tree of height $h + 1$, consisting of $2^h$ balanced binary trees as the direct successors (children) of the root node $u$ of $t$. Let the prefix notation of $t$ be $\text{pref}(t) = \text{apref}(t^h_{1,1})\text{pref}(t^h_{2,1})\ldots\text{pref}(t^h_{2^h,1})$, where $\varphi(a) = 2^h$ and

$$\text{pref}(t^0_{j,k}) = \begin{cases} a_0 & 1 \leq j \leq 2^h, 1 \leq k \leq 2^h, j \neq k \\ b_0 & 1 \leq j \leq 2^h, 1 \leq k \leq 2^h, j = k \end{cases}$$

$$\text{pref}(t^i_{j,k}) = a^i\text{pref}(t^{i-1}_{j,2k-1})\text{pref}(t^{i-1}_{j,2k}), 1 \leq i \leq h, 1 \leq j \leq 2^h, 1 \leq k \leq 2^{h-i}, \varphi(a') = 2$$

Let $Q_i \in \mathcal{P}(\{1,2,\ldots,2^h\})$ be distinct elements from the powerset of the set of numbers $1$ to $2^h$, where $1 \leq i < 2^h$. For each such set $Q_i$, we construct a balanced binary tree $t^h_{Q_i}$ of height $h$, such that $\text{pref}(t^h_{Q_i}) = a''\text{pref}(t^{h-1}_{Q_i,1})\text{pref}(t^{h-1}_{Q_i,1})$, where $\varphi(a'') = 2$ and

$$\text{pref}(t^0_{Q_i,k}) = \begin{cases} S & 1 \leq k \leq 2^h, k \notin Q_i \\ a_0 & 1 \leq k \leq 2^h, k \in Q_i \end{cases}$$

$$\text{pref}(t^i_{Q_i,k}) = \text{upref}(t^{i-1}_{Q_i,2k-1})\text{pref}(t^{i-1}_{Q_i,2k}), 1 \leq j \leq h, 1 \leq k \leq 2^{h-i}, \varphi(u) = 2$$

Tree $t^h_{i,1}$ matches $t^h_{Q_i}$ if and only if $i \in Q_j$, where $1 \leq i \leq 2^h$ and $1 \leq j < 2^h$. The tree $t$ consists of $2^h \times (2^{h+1} - 1) + 1$ nodes, while there are more than $2^{2^h - 1}$ sets $Q_i$ representing possible matching tree patterns. Hence, $\mathcal{O}(2^{2h})$ different states representing the subsets of matching subtrees may exist.

A graphical illustration of Lemma 18 is shown in Figure 5.10

### 5.3.3 Subtree PDA and repeats in trees

In section 5.3.1 we have presented the subtree PDA, which is analogous, in its properties, to the string suffix automata and represents a complete index, in terms of subtrees, for a given tree $t$ for which it is constructed. In this section we will describe a new and simple method for finding various kinds of repeats of subtrees in a given tree by exploiting the subtree PDA, and thus solving Problem 5.

Given the corresponding subtree PDA for the prefix notation of some tree $t$, our method computes all repeating subtrees in $t$ and provides a summary via a so-called subtree repeat table (see Definitions). We define two version of the subtree repeat table: the first, basic, version of the table contains basic information on repeats and its size is linear to the number of nodes of the tree. The second one, called the extended subtree repeat table, contains further information such as the prefix notation of each distinct subtree and a list of positions.

119
Figure 5.10: Graphical representation of proof of Lemma 18

**Definition 80 (Subtree position set)**  A subtree position set $sps(t', t)$, where $t'$ is a subtree of $t$ over a ranked alphabet $A = (\Sigma, \varphi)$, is the set

$$sps(t', t) = \{ |x| + 1 \mid \text{pref}(t) = x \text{pref}(t')y, \ x, y \in \Sigma^* \}$$
Definition 81 (List of subtree repeats) Let $t$ be a tree over a ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$. Given a subtree $t'$ of $t$, the list of subtree repeats $\text{lsr}(t', t)$ is a relation in $\text{sps}(t', t) \times \{F, S, Q\}$ defined as follows:

- $(i, F) \in \text{lsr}(t', t)$ if and only if $\text{pref}(t) = x\text{pref}(t')y$, $i = |x| + 1$, $x \neq x_1\text{pref}(t')x_2$,
- $(i, S) \in \text{lsr}(t', t)$ if and only if $\text{pref}(t) = x\text{pref}(t')y$, $i = |x| + 1$, $x = x_1\text{pref}(t')$,
- $(i, G) \in \text{lsr}(t', t)$ if and only if $\text{pref}(t) = x\text{pref}(t')y$, $i = |x| + 1$, $x = x_1\text{pref}(t')x_3$,

where $x, x_1, x_2 \in \Sigma^*$ and $x_3 \in \Sigma^+$.

Informally, the list of subtree repeats for a subtree $t'$ contains the starting position of the occurrence of a subtree and the type of occurrence, which is indicated by the letters F, S, and G. The abbreviations F, S, and G stand for First occurrence of the subtree, a repeated occurrence as a Square (siblings), and a repeated occurrence with a Gap (from the previous occurrence), respectively. In comparison with the types of repeats in strings [89, 91], subtree repeats miss the type O (Overlap) since no two different occurrences of the same subtree can overlap.

Definition 82 (Basic subtree repeat table) Given a tree $t$, the basic subtree repeat table $\text{BSRT}(t)$ is the set of all lists of subtree repeats $\text{lsr}(t', t)$, where $t'$ is a subtree with more than one occurrence in $t$.

The extended subtree repeat table $\text{ESRT}(t)$ is the set of all triplets of the form $(\text{sps}(t', t), \text{pref}(t'), \text{lsr}(t', t))$, where $t'$ is a subtree with more than one occurrence in $t$.

Example 26 Consider the tree $t$ illustrated in Figure 5.11 with prefix notation $x = a_2a_2a_0a_1a_0a_2a_0a_1a_0$. The subtrees of $t$ with more than one occurrences are $t_1, t_2$ and $t_3$, with their prefix notations being the strings $\text{pref}(t_1) = a_2a_0a_1a_0$, $\text{pref}(t_2) = a_1a_0$, and $\text{pref}(t_3) = a_0$.

It holds that $\text{sps}(t_1) = \{2, 6\}$, $\text{sps}(t_2) = \{4, 8\}$, $\text{sps}(t_3) = \{3, 5, 7, 9\}$, and the corresponding basic subtree repeat table $\text{BSRT}(t)$ and extended subtree repeat table $\text{ESRT}(t)$ are illustrated in Table 5.1 and 5.2, respectively.
from the initial state that corresponds to the prefix notation of a tree. For each such path corresponding to the prefix notation \( x \) of some tree, the following step is carried out:

**Step:** Let \( V \) be the list of numbers \((i_1, i_2, \ldots, i_r)\) contained in state \( q \). An entry \( E \) is created in \( BSRT(t) \) and the couple \((i_1, F)\) is inserted in \( E \). Then, for each subsequent number \( i_j \) such that \( 2 \leq j \leq r \), a new couple is inserted in \( E \) according to the following two rules:

1. \((i_j, S)\) if \( i_j - i_{j-1} = |x| \)
2. \((i_j, G)\) if \( i_j - i_{j-1} > |x| \)

**Example 27** The subtree PDA for tree \( t \) from Figure 5.11 is illustrated in 5.12, with the states and transitions not forming and not leading to a prefix notation of some tree removed. The method described for computing the repeats produces exactly Table 5.1.
5.3.4 Conclusion

In this section we have described two new kinds of deterministic pushdown automata for indexing ranked tree structures — the subtree PDA and the tree pattern PDA. These pushdown automata are analogous, in their properties, to suffix automata which are widely used in stringology. The two types of PDA represent a complete index of the subject tree of size \( n \) over which they were constructed, and allow us to find all occurrences of input patterns of size \( m \) in time linear to \( m \) and not depending on \( n \). The subtree PDA allows us to locate the occurrences of subtrees (not templates) and consists of a linear number of states and transitions to the size of the indexed tree, while the tree pattern PDA allows us to locate the occurrences of all subtrees matching tree patterns (including templates), however its size, in the worst case, may be exponential to the size of the subject tree.
Chapter 6

Computing repeats in ranked tree structures

“Happiness is the longing for repetition”

— Milan Kundera, from his novel “The Unbearable Lightness of Being”

The material presented in this section was published in part in [23, 24].


6.1 Introduction

Tree pattern matching has been intensively studied over the past decades because of its various applications, among others, in mechanical theorem proving, term-rewriting, instruction selection, and non-procedural programming languages [44, 63, 78]. In addition, tree pattern matching has direct applications
in computational biology, e.g. glycan classification [79], exact and approximate pattern matching and discovery in RNA secondary structure [86].

Periodicity in strings have been of interest since the beginning of the 20th century and effective methods for finding various kinds of repetitions and repeats in a string form an important part of well-researched stringology theory [36, 91, 100]. Some of these methods are based on principles of constructing and analysing string suffix trees or string suffix automata, which represent complete index of the string for suffixes [11, 17, 31, 83, 89].

In many applications, it is essential to extract the repeated patterns in a tree within a mathematical structure [43, 47, 73]. In particular, the common subtrees problem consists of finding all of the subtrees having the same structure and the same labels on the corresponding nodes of two ordered labelled unranked trees [60]. This problem of equivalence, which is strictly related to the common subexpression problem [43, 47], arises, for instance, in the code optimisation phase of compiler design, or in saving storage for symbolic computations [3, 43, 47].

In this chapter, we consider a slightly different problem, and provide a completely different solution to what has been done so far. We focus on finding all subtree repeats – the subtrees occurring more than once – in a tree structure. This problem is analogous to the well-known problem of finding all the repetitions in a given word [31]. Notice that finding all subtree repeats can be solved by the algorithm presented in [60]. However, the presented solution requires the construction of a suffix tree, which is expensive in practical terms. Moreover, by analogy with standard suffix automata and repeats in strings, all repeats of subtrees in a tree can be directly computed by analysing states of the deterministic subtree pushdown automaton, which represents a full index of a tree for all subtrees [67]. This way of computing all repeats of subtrees in a tree can be found in [90]. However, this way of computing repeats leads to $O(n \log n)$ time complexity.

Apart from its pleasing theoretical features, finding all subtree repeats, can be directly and effectively applied as an alternative solution, on the maximum agreement subtree (MAST) problem for trees representing the evolutionary history of a set of species [61]. That is, given a set of evolutionary leaf-labelled trees on the same set of taxa, the MAST problem consists of finding a subtree homeomorphically included in all input trees, and with the largest number of taxa.

The proposed algorithm is divided into two phases: the preprocessing phase and the phase where all subtree repeats are computed. The preprocessing phase transforms the given tree to a string representing its postfix notation, and then computes arrays that store the height of each node of the tree, the parent of each node, and an indicator showing whether a node is the leftmost (first) child of its parent or not. The second phase, for computing all subtree repeats, is
done in a bottom-up manner, using a partitioning technique. The importance of
the proposed algorithm is underlined by the fact that it can be applied in both
unlabelled and labelled ordered ranked trees. Its linear runtime, as well as the
use of linear auxiliary space, are important parts of its quality.

In this chapter we focus on solving the following problem

**Problem 7** Compute all repeating subtrees within a given subject tree.

### 6.2 Algorithm

In this section, we present the algorithm for solving Problem 7. The algorithms
consist of two phases: the preprocessing phase and the phase where all subtree
repeats are computed. We divide the rest of this section in three subsections:
first, we present the preprocessing phase of the algorithm the results of which we
then use to show the algorithm for solving Problem 7.

Before proceeding to the preprocessing phase, however, we present an impor-
tant Lemma which will be used in the computation of repeats.

**Lemma 19** Let \( \text{post}(t) \) and \( w \) be a tree \( t \) in postfix notation and a factor of
\( \text{post}(t) \), respectively. Then \( w = w_1 \ldots w_{|w|}, w_i \in \mathcal{A} \), is the postfix notation of a
subtree of \( t \) iff: \( \text{ac}(w) = 0 \), \( w_1 \) corresponds to a leaf of \( t \) and no subtree rooted at
\( w_\ell \), where \( 1 \leq \ell \leq |w| \), has leftmost node that appears before \( w_1 \) in \( \text{post}(t) \).

**Proof** (\( \Rightarrow \)): Let \( w \) be the postfix notation of a subtree of \( t \). Then the above
conditions are met due to postorder traversal of the tree.

(\( \Leftarrow \)): Let \( w \) be a factor of \( \text{post}(t) \), such that \( \text{ac}(w) = 0 \), \( w_1 \) corresponds to a leaf of
\( t \) and no subtree rooted at \( w_\ell \), where \( 1 \leq \ell \leq |w| \), has leftmost node that appears
before \( w_1 \) in \( \text{post}(t) \). Due to the last restriction factors of \( \text{post}(t) \) starting from
\( w_1 \) might end:

- on a node \( z \) whose subtree has \( w_1 \) as a leftmost leaf. Then that factor is the
  postfix notation of the subtree rooted at \( z \) (we consider postorder traversal
  of the tree).

- on a node \( z \) whose subtree has a leftmost leaf found after \( w_1 \) in \( \text{post}(t) \). Then that factor contains a subtree, say \( s \), having \( w_1 \) as a leftmost leaf and
  a collection of subtrees found later in the postorder traversal of the tree,
  giving an arity checksum of less than 0.

\( \square \)
6.2.1 Preprocessing phase

The preprocessing phase consists of the computation of three auxiliary arrays which will be accessed during the second phase — the computation of the subtree repeats. Let \( x \) be the postfix notation \( post(t) \) of a ranked tree \( t \) of size \( n \). The three arrays that are to be computed are:

1. An integer array \( H \) whose element \( H[i] \), for all \( 1 \leq i \leq n \), is the height of the subtree of \( t \) whose root is the node corresponding to \( x[i] \).

2. An integer array \( P \) whose element \( P[i] \), for all \( 1 \leq i \leq n \), is the index of the parent of the node corresponding to \( x[i] \). In other words the node corresponding to \( x[P[i]] \) is the parent of the node corresponding to \( x[i] \) in \( t \).

3. A binary array \( F \), where \( F[i] \), for all \( 1 \leq i \leq n \), is 1 in case the node corresponding to \( x[i] \) is the first (leftmost) child of its parent node corresponding to \( x[P[i]] \), or 0 in the opposite case.

Arrays \( H \) and \( P \) can be computed in time \( O(n) \) with the Algorithms 17 and 19, respectively. Array \( F \) may also be computed in time \( O(n) \) by using Algorithm 40.

**Algorithm 40: Compute-First-Child-Array-Postfix**

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

1 \( R \leftarrow \text{New-Stack} \)
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            \( r \leftarrow \text{Pop}(R) \)
8            \( F[r] \leftarrow 0 \)
9        \( r \leftarrow \text{Pop}(R) \)
10       \( F[r] \leftarrow 1 \)
11       Push(\( R, i \))
```

6.2.2 Computing subtree repeats in unlabeled trees

We are now in a position to present the computation phase for solving Problem 7. The computation is based on the bottom-up traversal of the input tree \( t \),
which corresponds to the traversal of its postfix notation $x[1 \ldots n] = \text{post}(t)$. The algorithm consists of three functions, SUBTREE-REPEATS, ASSIGN-LEVEL, and PARTITION, which are listed in Algorithms 41, 42, and 43, respectively.

The computation runs in $h(t)$ steps: the subtree repeats of height $i$, where $1 \leq i \leq h(t)$, are computed at step $i$. This is done to ensure that, at step $i$, all subtrees of height $i - 1$ are already processed, their length is known, and thus we avoid processing them again. At step $i = 1$ the starting positions of nodes of arity $0$ (leaves) are already split in sets (repeats) according to their label. Each such set is subsequently partitioned into smaller subsets according to the height of their parents. These subsets are composed of only the starting positions of those nodes that correspond to the leftmost nodes (first child) of their immediate subtree of height 1. Each resulting subset is placed in queue $j$ of an array of queues, where $j$ is the height of the immediate subtree. We will refer to this array as the level array $\text{LA}$. Step $i$ of the computation partitions the subsets of queue $i$ into smaller sets each one containing the starting positions of a particular subtree of height $i$. This process is repeated until tree $t$ is read completely at step $h(t)$.

The main function of this phase is SUBTREE-REPEATS, which creates sets consisting of starting positions of leaves according to their label (lines 12-22). Each such set is represented by queue $k$ of array $A_{\Sigma}$ of queues, where $k$ is the identifier for a symbol and its rank, assigned by the mapping $\mu$. For each starting position $i$ corresponding to a one-node subtree, $T[i]$ represents an identifier for the detected one-node subtree, and $\text{TL}[i]$ represents the detected subtree’s length. $T[i]$ and $\text{TL}[i]$ are set to an identifier $sc$ and 1, respectively (lines 19-22). Each such set $S$, together with two parameters, the size $\ell$ of the subtree it represents and the arity check $ac$, form a triplet. Each such triplet denotes all the occurrences of a specific factor $w = x[i \ldots i + \ell - 1]$, for all $i \in S$. Trivially, if $w$ corresponds to a postfix notation of a tree, $ac$ is 0. Those triplets—in this case $\ell = 1$ and $ac = 0$—are then passed to function ASSIGN-LEVEL. Note that only triplets with $ac = 0$, i.e. triplets describing trees, are passed to ASSIGN-LEVEL.

Given a triplet $(S, \ell, 0)$, ASSIGN-LEVEL creates at most $h(t)$ pairwise disjoint subsets. Each such set $A_n[k]$, $1 \leq k < h(t)$, consists of only those elements $i \in S$ such that the root node of subtree with postfix notation $x[i \ldots i + \ell - 1]$ (line 4) is the first child of its parent node $p = x[P[i + \ell - 1]]$, and the height of the subtree with root node $p$ is $k$. Formally, the work of ASSIGN-LEVEL could be described as

$$A_n[k] = \{ i \mid i \in S \land \text{FC}[i + \ell - 1] = 1 \land \text{H}[P[i + \ell - 1]] = k \}.$$ 

Since $h(t)$ can be at most $n - 1$ (unary tree), we use a bit vector $B_n$ and a queue $Q_4$ to mark the created sets, and to subsequently delete them, without the need of emptying the whole array $A_n$ of queues, which would require $O(n)$ time.
Algorithm 41: Subtree-Repeats

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

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

1. \( sc \leftarrow 0 \)
2. \( A_\Sigma[1..|\Sigma|], A_n[1..n] \leftarrow \text{NEW-QUEUE-ARRAY} \)
3. \( B_\Sigma[1..|\Sigma|] \leftarrow \text{NEW-BIT-ARRAY} \)
4. \( B_n[1..n] \leftarrow \text{NEW-BIT-ARRAY} \)
5. \( C_\Sigma[1..|\Sigma|] \leftarrow \text{NEW-INTEGER-ARRAY} \)
6. \( E_\Sigma[1..|\Sigma|], E_n[1..n] \leftarrow \text{NEW-TRIplet-ARRAY} \)
7. \( Q_5 \leftarrow \text{NEW-QUEUE} \)
8. \( LA[1..h(t)] \leftarrow \text{NEW-QUEUE-ARRAY} \)
9. \( FC[1..n] \leftarrow \text{COMPUTE-FIRST-CHILD-ARRAY} \)
10. \( H[1..n] \leftarrow \text{COMPUTE-NODE-HEIGHT-ARRAY} \)
11. \( P[1..n] \leftarrow \text{COMPUTE-NODE-Parents-ARRAY} \)

for \( i \leftarrow 1 \) to \( n \) do

12. if \( \varphi(x[i]) = 0 \) then

13. \( k \leftarrow \mu(x[i], \varphi(x[i])) \)
14. if \( B_\Sigma[k] = 0 \) then
15. \( B_\Sigma[k] \leftarrow 1 \)
16. \( \text{ENQUEUE}(Q_5, k) \)
17. \( \text{ENQUEUE}(A_\Sigma[k], i) \)
18. if \( C_\Sigma[k] = 0 \) then
19. \( sc \leftarrow sc + 1 \)
20. \( C_\Sigma[k] \leftarrow sc \)
21. \( T[i] \leftarrow C_\Sigma[k]; TL[i] \leftarrow 1 \)
22. else
23. \( T[i] \leftarrow 0; TL[i] \leftarrow 0 \)

while not empty \( Q_5 \) do

25. \( k \leftarrow \text{DEQUEUE}(Q_5), B_\Sigma[k] \leftarrow 0 \)
26. \( \text{OUTPUT}(A_\Sigma[k], 1) \)
27. \( \text{ASSIGN-LEVEL}((A_\Sigma[k], 1, 0)) \)

for \( i \leftarrow 1 \) to \( H[n] \) do

29. while not empty \( LA[i] \) do
30. \( \text{PARTITION}(\text{DEQUEUE}(LA[i]), x) \)

129
A copy of the created triplet \((A_n[k], \ell, 0)\) is then placed in the \(k\)th queue of the level array (line 13), and \(A_n[k]\) is deleted (lines 11-15).

After calling \textsc{Assign-Level}, function \textsc{Subtree-Repeats} performs \(h(t)\) steps. At step \(i\) the triplets stored in the \(i\)th queue of the level array are retrieved, and are subsequently passed to function \textsc{Partition} as arguments (lines 29-31). Function \textsc{Partition} recursively partitions \(S\) into smaller pairwise distinct subsets representing the subtree repeats of height \(i\).

Assume the triplet \((S, \ell, ac)\) represents a factor \(w\) of length \(\ell\) of \(x\). For each \(i \in S\), \textsc{Partition} expands \(w\), where \(w = x[i..i+\ell-1]\), by either appending (concatenating) the string \(v = x[i+\ell..i+\ell + TL[i+\ell]]\) to \(w\), in case the postfix notation \(v\) of a tree starting at \(i\) was already processed by the algorithm (lines 5-11), or the single letter \(x[i+\ell]\), in case it was not previously processed (lines 12-19).

The created triplets are then moved in a single queue \(Q_3\) (lines 20-29). The triplets which themselves describe a tree, i.e. that have \(ac = 0\), are printed in the output (line 33), and are then passed as arguments to the function \textsc{Assign-Level}, which places them in the level array (line 37). The triplets that do not describe a tree, i.e. that have \(ac \neq 0\), are passed as arguments to the function \textsc{Partition} (line 39), which recursively expands them until they represent a tree.

**Theorem 12** \textsc{Algorithm Subtree-Repeats} computes all complete subtree repeats of a given labeled ordered ranked tree \(t\) consisting of \(n\) nodes in time \(\Theta(n)\).

**Proof** The preprocessing phase, i.e. the computation of \textsc{post}(\(t\)) and of arrays \(P\), \(H\), and \(FC\), requires time \(\Theta(n)\). During the expansion of the subtrees, performed in function \textsc{Partition}, the algorithm does not read a symbol more than once, but rather reads the previously expanded subtrees. Merging the subtrees is done in \(n - 1\) operations—number of children of the tree. \(\square\)
Algorithm 42: Assign-Level

**Input**: Triplet \((S, \ell, ac)\)

1. \(Q_4 \leftarrow \text{NEW-QUEUE}\)
2. **while** not empty \(S\) **do**
   3. \(i \leftarrow \text{DEQUEUE}(S)\)
   4. \(\text{root} \leftarrow i + \ell - 1\)
   5. **if** \(FC[\text{root}] = 1\) **then**
      6. \(k \leftarrow H[P[\text{root}]]\)
      7. \(\text{ENQUEUE}(A_n[k], i)\)
      8. **if** \(B_n[k] = 0\) **then**
         9. \(B_n[k] \leftarrow 1\)
         10. \(\text{ENQUEUE}(Q_4, k)\)
3. **while** not empty \(Q_4\) **do**
   4. \(k \leftarrow \text{DEQUEUE}(Q_4)\)
   5. \(\text{ENQUEUE}(LA[k], (A_n[k], \ell, 0))\)
   6. \(B_n[k] \leftarrow 0\)
   7. \(A_n[k] \leftarrow \text{CLEAR-LIST}\)
Algorithm 43: Partition

Input: Triplet $(S, \ell, ac)$, $x[1 \ldots n] = post(t)$

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

We conclude the description of the algorithm with an example to demonstrate the steps carried out by the algorithm.

![Tree t from Example 28](image)

Figure 6.1: Tree t from Example 28

**Example 28** Compute all subtree repeats for the tree t illustrated in Figure 6.1 where \( \text{post}(t) = a_0 a_0 a_0 a_1 a_2 a_0 a_1 a_3 a_0 a_1 a_1 a_0 a_0 a_1 a_2 a_0 a_2 a_0 a_0 a_0 a_2 a_2 a_4 \).

First, the three auxiliary arrays \( P, H \) and \( F \) are computed, which represent the parents array, height array and first child array, respectively. The three computed arrays are presented in Table 6.1. The tree in Figure 6.2 demonstrates the partitioning that is carried out by Algorithm 41 at each level (height). The underlined factors correspond to postfix notations of subtrees of height \( i \) found at level \( i \). Table 6.2 shows the tuples that are inserted in the queues of the level array throughout the computation. Each such triplet corresponds to the postfix notation of some subtree of \( t \) which, for clarity, is given as a subscript index to each triplet in the queues.

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| post(t) | a_0 | a_0 | a_0 | a_1 | a_2 | a_0 | a_1 | a_3 | a_0 | a_1 | a_1 | a_0 | a_0 | a_1 | a_2 | a_0 | a_2 | a_0 | a_0 | a_0 | a_1 | a_2 | a_2 | a_4 |
| P     | 8  | 5  | 4  | 5  | 8  | 7  | 8  | 25 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| H     | 0  | 0  | 0  | 1  | 2  | 0  | 1  | 3  | 0  | 1  | 2  | 3  | 0  | 1  | 2  | 4  | 0  | 1  | 0  | 1  | 0  | 1  | 2  | 3  |
| F     | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  |

Table 6.1: The arrays \( P, H, \) and \( F \) computed during the preprocessing phase
Figure 6.2: An overview of the partitioning carried out by Algorithm 41
Table 6.2: Triplets placed in the level array $L$ during the computation of repeating subtrees for Example 28

<table>
<thead>
<tr>
<th>Index</th>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${(3, 6, 9, 14, 18, 22), 1, 0}_{a_0}$</td>
</tr>
<tr>
<td>2</td>
<td>${(9), 2, 0}<em>{a_0a_1}, ({2, 13, 21}, 1, 0}</em>{a_0}$</td>
</tr>
<tr>
<td>3</td>
<td>${(9), 3, 0}<em>{a_0a_1a_1}, ({1}, 1, 0}</em>{a_0}$</td>
</tr>
<tr>
<td>4</td>
<td>${(9), 4, 0}_{a_0a_1a_1a_1}$</td>
</tr>
<tr>
<td>5</td>
<td>${(1), 8, 0}_{a_0a_0a_0a_1a_2a_0a_1a_3}$</td>
</tr>
</tbody>
</table>

Finally, Table 6.3 lists all factors of $x$ corresponding to subtrees of $t$ along with the identifier they are assigned (array $T$) and the height of the subtree they correspond to. Note, that the table is sorted according to the order the factors are processed by the algorithm.

The output of the algorithm is a list of couples of the form $(S, \ell)$. Each such couple describes a factor $y$ corresponding to a subtree of $t$. The couple consists of $S$ which is the set of starting positions of all occurrences of $y$ in $x$ and $\ell$ is the length of $y$. For our example, the output consists of the following couples:

$(\{1,2,3,6,9,13,14,18,19,21,22\}, 1)$
$(\{18\}, 3)$
$(\{3,6,9,14,22\}, 2)$
$(\{2,13,21\}, 4)$
$(\{9\}, 3)$
$(\{1\}, 8)$
$(\{9\}, 4)$
$(\{9\}, 9)$
$(\{1\}, 25)$

## 6.3 Experimental results

We have conducted an experiment to verify the linear runtime of the proposed algorithm for computing subtree repeats in unlabeled ordered ranked trees in practice. For the experiment we have used randomly generated trees, with their size ranging from 100 to 10,000 nodes, with a step of 100 nodes, and a bounded alphabet with the arity of the symbols ranging between 0 and 5. Figure 6.3 demonstrates the number of operations carried out by our implementation of the algorithm against the number of nodes of the tree instances. The resulting graph
clearly indicates the linear relationship between the runtime and the number of nodes of tree instances.

Table 6.3: Postfix representation of indexed subtrees

<table>
<thead>
<tr>
<th>Id</th>
<th>Height</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$a_0$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$a_0a_0a_2$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>$a_0a_1$</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$a_0a_0a_1a_2$</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>$a_0a_1a_1$</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>$a_0a_0a_0a_1a_2a_0a_1a_3$</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>$a_0a_1a_1a_1$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>$a_0a_1a_1a_1a_0a_0a_1a_2a_2$</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>$a_0a_0a_0a_1a_2a_0a_1a_3a_0a_1a_1a_0a_0a_1a_2a_2a_0a_2a_0a_0a_1a_2a_4$</td>
</tr>
</tbody>
</table>

Figure 6.3: Number of operations performed by Algorithm 41 against the number of nodes
Chapter 7

Conclusion

“Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less”

— Marie Skłodowska-Curie

In this chapter, we turn to the problem statement and research questions formulated in Chapter 1, and consider the answers and contributions provided by this dissertation. We also present a list of suggested future work to extend the research reported on in the dissertation.

7.0.1 Contributions

In this thesis, some basic arbology results, principles and algorithms are presented. The thesis is divided in several chapters each dealing with a specific problem.

In the introduction, some related works are given along with a brief motivation explaining why the arbology research was established. The rest of the thesis presents some novel, systematic approaches to tree pattern matching using pushdown automata, an algorithm for tree pattern matching in unranked trees, tree indexing structures, and methods for computing repeats in ranked trees. Specifically, the following topics and results are presented:

Linear notations Two linear notations for representing ranked, ordered tree structures and two linear notations for representing unranked, ordered are introduced, and some properties of these linear notations are presented and proved. Moreover, some necessary and relevant algorithms manipulating with these linear notations, and which are used in subsequent chapters, are given.
**Subtree matching in ranked trees** In this chapter, we present pushdown automata based solutions for the exact subtree matching problem which are directly analogous to the finite automata based solutions for exact string pattern matching. Specifically, we design a KMP style pushdown automaton built over some linear notation of a subtree pattern \( p \) and which serves as a pattern matcher. The size of such an automaton is \( \mathcal{O}(m) \) and, given a linear notation of a subject tree \( t \), the PDA locates all occurrences of \( p \) in \( t \), in time \( \mathcal{O}(n + occ) \), where \( occ \) is the number of occurrences. Furthermore, we extend this type of PDA to an Aho-Corasick style PDA, which can locate, in the linear notation of a subject tree \( t \), all the occurrences of a set \( X = \{p_1, p_2, \ldots, p_k\} \) of subtree patterns. The size of this automaton is linear to the sum of sizes of all subtree patterns and the searching phase is again carried out in time linear to the size of the subject tree. Note that, once the PDA is constructed, it can be used to search in arbitrary many subject trees (preprocess once, search many).

**Tree template matching in ranked trees** Presented is, in this chapter, a tree template matching method for ranked trees, based on pushdown automata, that locates all occurrences of a given pattern in some subject tree in time linear to the size of the subject tree. Although the preprocessing time can be, in some special cases, exponential to the size of the tree pattern, the preprocessing is done only once, and the PDA that is constructed may be used for locating the referred tree pattern in arbitrary many subject trees (preprocess once, search many).

**Tree template matching in unranked trees** We have presented a tree template matching algorithm for unranked trees that finds all occurrences of a given pattern in some subject tree in time linear to the size of the subject tree. Again, as in the method for the ranked trees, the constructed data structure may be exponential in size to the tree pattern, however once the data structure is constructed, it may be used to search in arbitrary many subject trees. Moreover, the tree does not need to be transformed into a ranked tree in case the arity of nodes is not known beforehand, for example if the children of each node are represented by linked lists. The tree template matching can be carried out by just one postorder traversal of the subject tree.

**Tree indexing** Chapter 5 briefly describes some popular methods of text indexing. Specifically, a brief description of suffix trees, suffix automata (DAWGs) and suffix arrays is provided, and the current research status of these data structures is listed. Analogous data structures for tree indexing which are based on the pushdown automaton, and which resemble the
suffix automata approach for text indexing are then proposed. Particularly, we have introduced the subtree PDA which serves as an index of subtrees for a given subject tree, and the tree pattern PDA which serves as an index of tree patterns for a given subject tree. We have proved that the subtree PDA is analogous to the suffix automaton and its number of states is linear to the size of the tree that is indexed. Concerning the tree pattern PDA, we have proved that the number of states may be exponential to the size of the indexed tree.

**Computing subtree repeats** In the last chapter we have provided a linear time algorithm for computing all subtree repeats in a given subject tree. The algorithm runs in time linear to the size of the subject tree and may be used on both labeled and unlabeled trees. The output of the algorithm can, for example, be used in the identification of the most frequent subtree by a single traversal of the subject tree.

### 7.0.2 Future work

As future work, we see a number of extensions of the research reported in this dissertation thesis. We provide brief definitions of open problem that are to be investigated.

**Problem 8** Construct a pushdown automaton for a given tree pattern \( P \), that will be used for approximate tree pattern matching in a subject tree.

Problem 8 has a certain connection with a well known problem — the tree edit distance problem. The ordered edit distance problem was introduced by Tai [102] as a generalisation of the well-known string edit distance problem [105]. Tai presented an algorithm for the ordered version using \( \mathcal{O}(|T_1||T_2||L_1|^2|L_2|^2) \) time and space, where \( T_1 \) and \( T_2 \) are the two trees, and \( L_1 \) and \( L_2 \) are the sets of leaves of the two trees, respectively. Subsequently, Zhang and Sasha [108] gave a simple algorithm improving the bounds to \( \mathcal{O}(|T_1||T_2|\min(L_1,H_1)\min(L_2,H_2)) \) time and \( c\mathcal{O}(|T_1||T_2|) \) space, where \( H_1 \) and \( H_2 \) denote the heights of trees \( T_1 \) and \( T_2 \), respectively. This algorithm was modified by Klein [76] to get a better worst case time bound of \( \mathcal{O}(\min(|T_1|^2|T_2|\log|T_2|),|T_2|^2|T_1|\log|T_1|) \). Finally, Demaine et al. [39, 40] presented a \( \mathcal{O}(\min(|T_1|^3,|T_2|^3)) \) time algorithm. For a survey on tree edit distance see [14, 15]. Although the problem of the tree edit distance has been addressed several times and many solutions do exist, no solution until now was presented by means of pushdown automata. A pushdown automata solution to the approximate tree pattern matching problem resembling the finite automata based solution for the string edit distance problem presented in [88] would further enhance the theory behind the field of arbology.
Problem 9 Design and propose a compacted version of the tree pattern PDA resembling the compacted version (CDAWG) of the suffix automaton [16, 66].

A data structure that would be more interesting in practice would be a compacted version of the tree pattern PDA.

Another data structure to construct is a compacted version of the tree pattern PDA, as stated in Problem 9, and which would be, in practice, more desirable and interesting than the tree pattern PDA.

Problem 10 Construct a deterministic tree pattern oracle pushdown automaton.

Problem 10 presents another interesting research direction — how to construct a tree pattern oracle PDA which would be analogous, in its properties, to the string factor oracle finite automaton [6]. In other words, the idea is to construct a PDA that will accept a superclass of the language accepted by the tree pattern PDA built over a given tree, while retaining a linear number of states to the size of the subject tree, as opposed to the tree pattern PDA which may consist of an exponential number of states.

Problem 11 Given a linear notation $x$ (postfix or prefix) of a tree $t$, compute the smallest tree pattern $p$, where $|p| < |t|$, such that the occurrences of its corresponding linear notation $y$ in $x$ cover the highest possible number of symbols of $x$ by non-placeholder wildcard symbols.

One more interesting problem from the side of data compression is stated in Problem 11. The problem is to find the smallest tree pattern $p$ such that the number of nodes of $t$, matched by non-wildcard nodes of $p$ at all subtrees of $t$ that are matched by $p$, is the maximal. We present this problem with Example 29.

Example 29 (Problem 11) Let $x = a_2a_0a_1a_2a_0a_1a_2a_0a_1$ be the prefix notation of a subject tree $t$ illustrated in Figure 7.1. There are several tree patterns that cover all the nodes of $x$. For example the tree pattern with prefix notation $a_2SS$, where $S$ is the wildcard symbol, covers just 3 symbols of $x$ with its non-wildcard nodes. The tree pattern with prefix notation $a_2a_0S$ is a better solution as it covers 6 symbols of $x$ with its non-wildcard nodes. However, the best solution is the tree pattern with prefix notation $a_2a_0a_1S$ as it covers 9 symbols of $x$ with its non-wildcard nodes.

The last problem is about approximate tree pattern matching and indexing.

Problem 12 Given an integer $k$, construct an indexing structure for a given subject tree $t$, such that one can query for a tree pattern $p$ against the structure and receive the positions of subtrees $u_1, u_2, \ldots, u_r$ of $t$ in time $O(m + r)$, such that the edit distance between $u_i$ and $p$, for all $1 \leq i \leq r$, is less or equal to $k$.

For a survey on tree edit distance see [14].
Figure 7.1: Subject tree $t$ from Example 29
Bibliography


Relevant publications of the author


Other publications of the author


