Automata Approach to Inexact Tree Pattern Matching Using 1-degree Edit Distance [⋆]

Eliška Šestáková, Ondřej Guth, and Jan Janoušek

Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Praha 6
Czech Republic
{sestaeli,guthondr,janousej}@fit.cvut.cz

Abstract. We compare labeled ordered trees based on unit cost 1-degree edit distance that uses operations vertex relabeling, leaf insertion, and leaf deletion. Given an input tree T and a tree pattern P, we find all subtrees in T that match P with up to k errors. We show that this problem can be solved by finite automaton when T and P are represented in linear, prefix bar, notation. First, we solve this problem by a pushdown automaton. Then, we show that it can be transformed into a nondeterministic finite automaton due to its restricted use of the pushdown store. We also show a simulation of the nondeterministic finite automaton by dynamic programming.

Keywords: inexact tree pattern matching, approximate tree pattern matching, finite automaton, pushdown automaton, subtree matching, dynamic programming, linear tree notation, prefix bar notation, Selkow distance, 1-degree edit distance, ordered trees

1 Introduction

The problem of inexact (or approximate) tree pattern matching is for a given input tree T and tree pattern P, find all subtrees in T that match P with up to k errors. This type of tree pattern matching can be helpful if one of the trees (or both) can be subjects of deformation or corruption; in such circumstances, the tree pattern matching needs to be more tolerant when comparing two trees.

The problem of measuring similarities between two trees is called the tree edit distance problem (or tree-to-tree correction problem). This problem is a generalization of the well-known string edit distance problem, and it is defined as the minimum cost sequence of vertex edit operations that transform one tree into the other [1].

In this paper, we consider labeled ordered trees in which each vertex is associated with a label, and sibling order matters. For labeled ordered trees, Tai [10] introduced the set of operations that included vertex relabeling, vertex insertion, and vertex deletion. A different cost may accompany the operations. Given two labeled ordered trees with m and n vertices, where $n \geq m$, the tree edit distance between those trees can be solved in cubic $\mathcal{O}(n^3)$ time [3]. According to a recent result [2], it is unlikely that a truly subcubic algorithm for the ordered tree edit distance problem exists.

For unordered trees, Zhang et al. proved that the tree edit distance problem is NP-complete, even for binary trees having an alphabet containing just two labels [12].

[⋆] *The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16 019/0000765 "Research Center for Informatics". This work was also supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/208/OHK3/3T/18.*

Several authors have also proposed restricted forms and variations of the tree edit distance problem. For example, Selkow [8] restricted the vertex insertion and deletion to leaves of a tree only. These operations may be used recursively to allow insertion or deletion of a subtree of arbitrary size. This distance is in the literature often referred to as 1-degree edit distance. Selkow also gave an algorithm running in $\mathcal{O}(nm)$ time and space, where n and m are the numbers of vertices of the two labeled ordered input trees. His algorithm uses a dynamic programming approach in which the input trees are recursively decomposed into smaller subproblems. A similar approach is used in most state-of-the-art algorithms for the tree edit distance problem.

The dynamic programming approach was also successfully used to solve the string edit distance problem and the inexact (approximate) string pattern matching problem [7]. Besides dynamic programming, however, finite automata can also be used to solve the inexact string pattern matching problem [6,7].

Inspired by techniques from string matching, we aim to show that the automata approach can also be used to solve the inexact tree pattern matching problem. We consider labeled ordered (unranked) trees and 1-degree edit distance. For simplicity, we use unit cost, where each operation costs one. However, the extension of our approach to non-unit cost distance is also discussed.

First, we solve the problem by a pushdown automaton. Then, we show that it can be transformed into a finite automaton due to its restricted use of the pushdown store. The deterministic version of the finite automaton finds all occurrences of the tree pattern in time linear to the size of the input tree. We also present an algorithm based on dynamic programming, which is a simulation of the nondeterministic finite automaton. The space complexity of this approach is $\mathcal{O}(mk)$ and the time complexity is $\mathcal{O}(kmn)$, where m is the number of vertices of the tree pattern, n is the number of vertices of the input tree, and $k \leq m$ represents the number of errors allowed in the pattern.

Our approach extends the previous result by Šestáková, Melichar, and Janoušek [9]. In their paper, they used a finite automaton to solve the inexact tree pattern matching problem with a more restricted 1-degree edit distance; the distance uses the same set of operations defined by Selkow, but these operations cannot be used recursively to allow insertion or deletion of a subtree of arbitrary size. Therefore, it may not always be possible to transform one tree into the other.

To be able to process trees using (string) automata, we represent trees using a linear notation called the prefix bar notation [5]. This notation is similar to the bracketed notation in which each subtree is enclosed in brackets. The prefix bar notation uses just the closing bracket (denoted by bar "|" symbol) due to the simple observation that the left bracket is redundant; there is always the root of a subtree just behind the left bracket. We note that this notation corresponds, for example, to the notation used in XML; each end-tag can be mapped to the bar symbol. Similarly straightforward is the transformation of JSON.

This paper is organized as follows. In Section 2, we give notational and mathematical preliminaries together with the formal definition of the problem statement. In Section 3, we present an algorithm for the computation of an auxiliary data structure, the subtree jump table. In Section 4, we present our automata approach. In Section 5, we show a dynamic programming algorithm that simulates the nondeterministic finite automaton. In Section 6, we conclude the paper and discuss the future work.

2 Preliminaries

An *alphabet*, denoted by Σ, is a finite nonempty set whose elements are called *symbols*. A *string* over Σ is a finite sequence of elements of Σ. The empty sequence is called the *empty string* and is denoted by ε. *The set of all strings* over Σ is denoted by Σ^*. The *length* of string $\boldsymbol{x}$ is the length of the sequence associated with $\boldsymbol{x}$ and is denoted by $|\boldsymbol{x}|$. By $\boldsymbol{x}[i]$, where $i \in \{1, \dots, |\boldsymbol{x}|\}$, we denote the symbol at index i of $\boldsymbol{x}$. The substring of $\boldsymbol{x}$ that starts at index i and ends at index j is denoted by $\boldsymbol{x}[i \dots j]$; i.e., $\boldsymbol{x}[i \dots j] = \boldsymbol{x}[i]\boldsymbol{x}[i+1] \dots \boldsymbol{x}[j]$. A *language* over an alphabet Σ is a set of strings over Σ.

2.1 Trees

A *tree* is a graph $T = (V, E)$, where V represent the nonempty set of vertices and E the set of edges, and where one of its vertices is called the *root* of the tree; the remaining vertices are called *descendants* of the root and can be partitioned into $s \geq 0$ disjoint sets $T_1, \dots, T_s$, and each of these sets, in turn, is a tree. The trees $T_1, \dots, T_s$ are called the *subtrees* of the root. There is an edge from the root to the root of each subtree. An *ordered* tree is a tree where the relative order of the subtrees $T_1, \dots, T_s$ is important.

If a tree is equipped with a vertex labeling function $V \to \Sigma$, we call it *a labeled tree* over Σ. The set of all labeled ordered trees over alphabet Σ is denoted by $\textsc{Tr}(\Sigma)$. All trees we consider in the paper are labeled ordered trees. Therefore, we will omit the words "labeled ordered" when referencing labeled ordered trees.

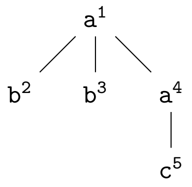
By T_v, where $v \in V$, we denote the *subtree of T rooted at vertex v*; i.e., T_v is a subgraph of tree T induced by vertex subset V' that contains vertex v (the root of tree T_v), and all its descendants. If a vertex does not have any descendants, it is called a *leaf*.

The prefix bar notation [5] of tree T, denoted by $\textsc{pref-bar}(T)$, is defined as follows: If T contains only the root vertex with no subtrees, then $\textsc{pref-bar}(T) = a|$, where a is the label of the root vertex. Otherwise,

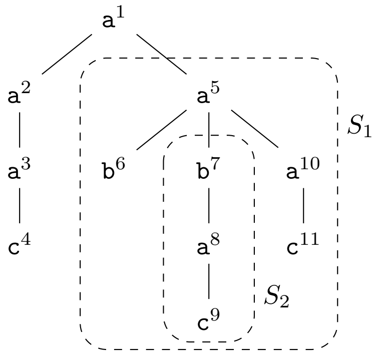
$$\textsc{pref-bar}(T) = a\,\textsc{pref-bar}(T_1)\,\textsc{pref-bar}(T_2) \cdots \textsc{pref-bar}(T_s)\,|,$$

where a is the label of the root of T and $T_1, \dots, T_s$ are its subtrees. For a tree T with n vertices, the prefix bar notation is always of length $2n$. For every label a in the prefix bar notation, there is the corresponding bar symbol "|" indicating the end of the subtree T_a; we call such pair of a label and its corresponding bar symbol a *label-bar pair*.

The *subtree jump table* for prefix bar notation is a linear auxiliary structure introduced by Trávníček [11] that contains the start and the end position of each subtree for trees represented in the prefix bar notation. Formally, given a tree T with n vertices and its prefix bar notation $\textsc{pref-bar}(T)$ with length $2n$, the subtree jump table $\boldsymbol{S}_T$ for T is a mapping from a set of integers $\{1, \dots, 2n\}$ into a set of integers $\{0, \dots, 2n+1\}$. If the substring $\boldsymbol{x}[i \dots j]$, where $1 \leq i < j$ is the prefix bar representation of a subtree of T, then $\boldsymbol{S}_T[i] = j+1$ and $\boldsymbol{S}_T[j] = i-1$. In the prefix bar notation, it holds that every subtree of tree T is a substring of $\textsc{pref-bar}(T)$. It also holds that every such substring ends with the bar symbol. When discussing the time complexity of our algorithms, we will assume that the subtree jump table is implemented as an array, and therefore each position $i \in \{1, \dots, 2n\}$ can be accessed in $\mathcal{O}(1)$ time.



(a) Example of tree pattern P.



(b) Example of input tree T and its subtrees S_1, S_2.

Figure 1: Example ordered labeled trees.

Example 1 (Prefix bar notation and subtree jump table). Let P be a tree illustrated in Figure 1a. Then, PREF-BAR$(P) = \boldsymbol{p} = \texttt{ab|b|ac|||}$. The subtree jump table $\boldsymbol{S}_P$ for P is as follows:

$\boldsymbol{p}$	a	b	|	b	|	a	c	|	|	|
j	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{S}_P[j]$	11	4	1	6	3	10	9	6	5	0

2.2 Pushdown and finite automaton

An *(extended) pushdown automaton* (PDA) is a 7-tuple $\mathcal{M}_{\text{PDA}} = (Q, \Sigma, G, \delta, q_0, \texttt{Z}, F)$ where Q is a finite set of states, Σ is an input alphabet, G is a pushdown store alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times G^* \to \mathcal{P}(Q \times G^*)$ is a transition function (not necessarily total), where $\mathcal{P}(Q \times G^*)$ contains only finite subsets of $Q \times G^*$, $q_0 \in Q$ is the initial state, $\texttt{Z} \in G$ is the initial pushdown symbol, $F \subseteq Q$ is the set of final states. By $L(\mathcal{M}_{\text{PDA}})$ we denote the language accepted by $\mathcal{M}_{\text{PDA}}$ by a final state.

A *nondeterministic finite automaton* (NFA) is a 5-tuple $\mathcal{M}_{\text{NFA}} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is a state transition function (not necessarily total), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states. A finite automaton is *deterministic* (DFA) if $\forall a \in \Sigma$ and $q \in Q : |\delta(q, a)| \leq 1$. By $L(\mathcal{M}_{\text{FA}})$ we denote the language accepted by $\mathcal{M}_{\text{FA}}$.

2.3 Problem statement

In the Introduction, we have defined the problem of inexact tree pattern matching as finding the subtrees in an input tree that match a tree pattern with up to k errors. We now give a more formal definition for which we consider the following primitive operations applied to a tree $T = (V, E)$:

vertex relabel change the label of a vertex v,
leaf insert insert a vertex v as a leaf of an existing vertex u in V, and
leaf delete delete a non-root leaf v from T.

The operations may be used recursively to allow insertion or deletion of a subtree of arbitrary size. This set of operations was originally introduced by Selkow [8], and we will refer to it as to the set of *1-degree edit operations.*

The *unit cost 1-degree edit distance* is a function $d : \text{TR}(\Sigma) \times \text{TR}(\Sigma) \to \mathbb{N}_0$. Given two trees T_1 and T_2, the number $d(T_1, T_2)$ corresponds to the minimal number of 1-degree edit operations that transform T_1 into T_2.

Example 2 (1-degree edit distance). Let P be the tree illustrated in Figure 1a and S_1 be the tree illustrated in Figure 1b. Then, $d(P, S_1) = 2$ since we need to insert a leaf labeled by `a` as a child of the vertex with identifier 3 into P. Then, we add the leaf with label `c` as the child of the node `a` we inserted in the previous step.

Definition 3 (Inexact 1-degree tree pattern matching problem). *Let Σ be an alphabet. Let $T = (V_T, E_T)$ be an input tree with n vertices over Σ. Let $P = (V_P, E_P)$ be a comparatively smaller tree pattern over Σ with m vertices. Let k be a non-negative integer representing the maximum number of errors allowed. Let d be the unit cost 1-degree edit distance function. Given T, P, k, and d, the inexact 1-degree tree pattern matching problem is to return a set*

$$\Big\{v : v \in V_T \wedge d(T_v, P) \leq k\Big\}.$$

In other words, the problem is to return the set of all vertices such that each vertex v represents the root of a subtree of T which distance from the tree pattern P is at most k.

Example 4 (Inexact 1-degree tree pattern matching problem). Let P be the tree pattern illustrated in Figure 1a, T be the input tree shown in Figure 1b, and $k = 2$. The solution of the 1-degree inexact tree pattern matching problem is $\{2, 5\}$; i.e., with respect to the maximal number of allowed errors, P occurs in T in the subtrees rooted at nodes 2 and 5.

In the rest of the text, we will use the following naming conventions: T and P will represent an input tree and a tree pattern, respectively. We use n, m, k to represent the number of vertices in T, the number of vertices in P, and the maximum number of errors allowed. For brevity, we will use $\boldsymbol{t}$ and $\boldsymbol{p}$ as a shorthand for $\text{PREF-BAR}(T)$ and $\text{PREF-BAR}(P)$, respectively.

3 Subtree jump table computation for prefix unranked bar notation

A linear-time algorithm for computation of the subtree jump table was given by Trávníček [11, Section 5.2.2]. However, Trávníček's algorithm works for prefix ranked bar notation, which combines the prefix notation and the bar notation. Therefore, we give an alternative algorithm for computation of the subtree jump table that works directly with prefix (unranked) bar notation of trees (see Algorithm 1).

The central idea of our algorithm is the use of a pushdown store for recording the positions of the labels. When the bar symbol is found in the prefix bar notation, the position of the corresponding label is popped from the pushdown store.

Theorem 5 (Correctness of the subtree jump table computation). *Let $\boldsymbol{p}$ be a string, such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for some tree P. Algorithm 1 correctly computes the subtree jump table for $\boldsymbol{p}$.*

Proof. In the first **for**-loop (line 3), we use the pushdown store to save all indexes (line 7) that correspond to the positions of all vertex labels in the prefix bar notation. When the bar symbol is encountered, the position of the corresponding subtree root label is at the top of the pushdown store; we retrieve it and subtract one (line 5) since

Algorithm 1 Computation of the subtree jump table.

```
Input String $\boldsymbol{p}$, such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for some tree $P$.
Output The subtree jump table $\boldsymbol{S}_P$ for $\boldsymbol{p}$.
 1 $Y$: empty pushdown store
 2 $\boldsymbol{S}_P$: empty array of size $|\boldsymbol{p}|$
 3 for each position $j$ of $\boldsymbol{p}$:
 4     if $\boldsymbol{p}[j] = |$
 5         $\boldsymbol{S}_P[j] \leftarrow \text{POP}(Y) - 1$
 6     else
 7         $\text{PUSH}(Y, j)$
 8         $\boldsymbol{S}_P[j] \leftarrow \text{NULL}$
 9 for each position $j$ of $\boldsymbol{p}$:
10     if $\boldsymbol{S}_P[j] \neq \text{NULL}$
11         $\boldsymbol{S}_P[\boldsymbol{S}_P[j] + 1] \leftarrow j + 1$
12 return $\boldsymbol{S}_P$
```

the subtree jump table contains the index of the previous element, not the index of the subtree root label itself. In the second **for**-loop (line 9), we define the remaining positions in the subtree jump table. The $\boldsymbol{S}_P[\boldsymbol{S}_P[j] + 1]$ expression (line 11) computes the position of the subtree root label corresponding to the bar symbol at position $\boldsymbol{p}[j]$ and saves there the position $j + 1$ that is the index of the element following the current bar symbol at index j.

Theorem 6 (Time complexity of the subtree jump table computation). *Let* $\boldsymbol{p}$ *be a string, such that* $\boldsymbol{p} = \text{PREF-BAR}(P)$ *for some tree* P*. The subtree jump table for* $\boldsymbol{p}$ *can be computed in* $\mathcal{O}(|\boldsymbol{p}|)$ *time using Algorithm 1.*

4 Automata approach

To be able to solve the inexact 1-degree tree pattern matching problem defined in Section 2.3 using (string) automata, we represent trees as strings using the prefix bar notation. Therefore, given a string $\boldsymbol{x} = x_1x_2 \cdots x_r$ of length $r \geq 2$ over alphabet $\Sigma \cup \{|\}$ that represents the prefix bar notation of a tree, the 1-degree (tree) edit operations corresponds to the following string operations:

- the operation relabeling $\mathrm{R}(i, b)$ that for $i \in \{1, \ldots, r-1\}$, $b \in \Sigma$, and $\boldsymbol{x}[i] \in (\Sigma \setminus \{b\})$, change the symbol $\boldsymbol{x}[i]$ into symbol b;
- the operation insertion $\mathrm{I}(i, a)$ that for $i \in \{2, \ldots, r-1\}$ and $a \in \Sigma$ inserts the substring (leaf) "$a|$" at position i; and
- the operation deletion $\mathrm{D}(i)$ that for $i \in \{2, \ldots r-2\}$, $\boldsymbol{x}[i] \in \Sigma$, and $\boldsymbol{x}[i+1] = |$, deletes the substring (leaf) $\boldsymbol{x}[i]\boldsymbol{x}[i+1]$.

Example 7 (Application of 1-degree edit operations to strings). Let P be the tree illustrated in Figure 1a and S_1, S_2 be the trees illustrated in Figure 1b; $\text{PREF-BAR}(P) = \mathsf{ab|b|ac|||}$, $\text{PREF-BAR}(S_1) = \mathsf{ab|bac|||ac|||}$, and $\text{PREF-BAR}(S_2) = \mathsf{bac|||}$. Then, the distance $d(P, S_1) = 2$ and $d(P, S_2) = 3$ since

$$\mathsf{ab|b|ac|||} \xrightarrow{I(5,\mathsf{a})} \mathsf{ab|ba||ac|||} \xrightarrow{I(6,\mathsf{c})} \mathsf{ab|bac|||ac|||} \text{ and}$$

$$\mathsf{ab|b|ac|||} \xrightarrow{R(1,\mathsf{b})} \mathsf{bb|b|ac|||} \xrightarrow{D(2)} \mathsf{bb|ac|||} \xrightarrow{D(2)} \mathsf{bac|||}.$$

Given two strings $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ that both correspond to the prefix bar notation of trees, using the 1-degree (string) edit operations, we can define the unit cost 1-degree (string) edit distance as a function $d_s : (\Sigma \cup \{|\})^* \times (\Sigma \cup \{|\})^* \to \mathbb{N}_0$ such that $d_s(\boldsymbol{t_1}, \boldsymbol{t_2}) = d(T_1, T_2)$, where $\boldsymbol{t_1} = \textsc{pref-bar}(T_1)$ and $\boldsymbol{t_2} = \textsc{pref-bar}(T_2)$. Since the functions d and d_s differ only in argument types, we will use the notation d for both trees and string arguments.

Using the prefix bar representation of trees, we can specify the problem of inexact 1-degree tree pattern matching as finding all positions $i \in \{1, \dots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i]+1 \dots i]) \le k$. Recall that $\boldsymbol{S}_T$ represent the subtree jump table for input tree T and $\boldsymbol{S}_T[i] + 1$ returns the position in $\boldsymbol{t}$ that contains the subtree root label corresponding to the bar symbol at position $\boldsymbol{t}[i]$. In other words, our methods output end positions of the occurrences. The position of the corresponding root label can be computed in $\mathcal{O}(1)$ time for each end position using the subtree jump table for T.

Proposition 8. *Let $\mathcal{M}$ be either a pushdown or finite automaton accepting the language*

$$\{\boldsymbol{sp'} : \boldsymbol{s} \in (\Sigma \cup \{|\})^* \wedge d(\boldsymbol{p}, \boldsymbol{p'}) \le k\}. \quad (1)$$

The automaton $\mathcal{M}$ is called 1-degree automaton and it solves the inexact 1-degree tree pattern matching problem.

The 1-degree automaton $\mathcal{M}$ accepts infinite language. It can read (not necessarily accept) any prefix bar notation of a tree (over alphabet Σ), i.e., it does not fail due to non-existing transition. Algorithm 2 illustrates how $\mathcal{M}$ can be used to solve the inexact 1-degree tree pattern matching problem. In the following sections, we will discuss the construction of the 1-degree automaton in detail.

Algorithm 2 Automata approach to inexact 1-degree tree pattern matching.

Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \textsc{pref-bar}(P)$ for tree pattern P over alphabet Σ, a string $\boldsymbol{t}$ of length $2n$ such that $\boldsymbol{t} = \textsc{pref-bar}(T)$ for input tree T over alphabet Σ, a non-negative integer $k \le m$, a 1-degree automaton $\mathcal{M}$ for P and k.
Output All positions $i \in \{1, \dots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i] + 1 \dots i]) \le k$.

```
1 read $\boldsymbol{t}$ using $\mathcal{M}$ symbol-by-symbol ($\boldsymbol{t}[i]$ is the currently read symbol):
2     if a final state is reached:
3         output $i$
```

4.1 1-degree pushdown automaton

In this section, we show that the 1-degree automaton can be constructed as a pushdown automaton. Our method is similar to the construction of approximate string pattern matching automaton [6]. Algorithm 3 describes the construction of the 1-degree PDA in detail. An example of $\mathcal{M}_{\text{PDA}}$ is illustrated in Figure 2.

Each state of the automaton has a label j^l, where $0 \le j \le 2m$ is a *depth* of the state (position in the pattern) and $l \in \{0, \dots, k\}$ is a *level* of the state (actual number of errors). The pushdown store is used to match label-bar pairs and, therefore, to simulate leaf insertion operation.

Vertex relabeling operation can be applied if there is a different vertex label in $\boldsymbol{p}$ and $\boldsymbol{t}$ at the current position. Each relabel operation increases the distance by 1.

Algorithm 3 Construction of 1-degree pushdown automaton.

```
Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \textsc{pref-bar}(P)$ for a tree pattern $P$ over alphabet $\Sigma$,
a non-negative integer $k \le m$, the subtree jump table $\boldsymbol{S}_P$ for $P$.
Output 1-degree pushdown automaton $\mathcal{M}_{\text{PDA}}$ for $P$ and $k$.
1  define states $Q = \{0^0\} \cup \{j^l : 1 \le j \le 2m \wedge 0 \le l \le k\}$
2  define final states $F = \{2m^l : 0 \le l \le k\}$
3  define pushdown alphabet $G = \{\mathtt{Z}, \mathtt{c}\}$
4  add initial loop for the bar symbol: $\delta(0^0, |, \mathtt{Z}) = \{(0^0, \mathtt{Z})\}$
5  add initial state transitions for labels: $\delta(0^0, a, \mathtt{Z}) = \begin{cases} \{(0^0,\mathtt{Z}),(1^0,\mathtt{Z})\} : a = \boldsymbol{p}[1] \\ \{(0^0,\mathtt{Z}),(1^1,\mathtt{Z})\} : a \in (\Sigma \setminus \{\boldsymbol{p}[1]\}) \end{cases}$
6  for every pattern position $j : 2 \le j \le 2m$:
7      for every allowed number of errors $l : 0 \le l \le k$:
8          $\delta\big((j-1)^l, \boldsymbol{p}[j], \mathtt{Z}\big) = \big\{(j^l, \mathtt{Z})\big\}$   (label or bar match)
9          $\delta\big((j-1)^l, a, \mathtt{Z}\big) = \big\{(j^{l+1}, \mathtt{Z}) : l < k\big\} : a \in \Sigma \setminus \{\boldsymbol{p}[j]\}$   (relabel)
10         $\delta\big((j-1)^l, a, \varepsilon\big) = \delta((j-1)^l, a, \varepsilon) \cup \big\{((j-1)^{l+1}, \mathtt{c}) : l < k\big\} : a \in \Sigma \setminus \{|\}$   (label insert)
11         $\delta\big((j-1)^l, |, \mathtt{c}\big) = \big\{((j-1)^l, \varepsilon) : l > 0\big\}$   (bar insert)
12         $\delta\big((j-1)^l, \varepsilon, \mathtt{Z}\big) = \big\{((\boldsymbol{S}_P[j]-1)^{l+(\boldsymbol{S}_P[j]-j)/2}, \mathtt{Z}) : l + \frac{\boldsymbol{S}_P[j]-j}{2} \le k\big\} : \boldsymbol{S}_P[j] > j$   (delete)
13 return $\mathcal{M}_{\text{PDA}} = (Q, \Sigma \cup \{|\}, G, \delta, 0^0, \mathtt{Z}, F)$
```

These operations are represented by “diagonal” transitions labeled by the symbols of the alphabet Σ for which no “direct” transition to the next state exists.

Leaf deletion operations correspond to a situation in which a vertex label followed by the bar symbol is skipped in $\boldsymbol{p}$ while nothing is read in $\boldsymbol{t}$. The automaton performs such an operation by following one of its ε-transitions. Since 1-degree edit distance allows to delete a subtree of arbitrary size by picking its leaves one by one, the automaton needs to reflect this. That is why the target state of the ε-transitions is provided by the subtree jump table. The number of errors of such an operation is equal to the number of skipped vertex labels. Since a subtree of tree pattern P is a substring of $\boldsymbol{p}$ where label-bar pairs are balanced, the number of errors is equal to the substring length divided by 2.

Leaf insertion operations correspond to a situation in which a vertex label followed by the bar symbol is read in $\boldsymbol{t}$ while there is no advance in $\boldsymbol{p}$. To allow insertion of a subtree of arbitrary size into the tree pattern leaf by leaf, we use a special pushdown symbol $\mathtt{c}$. By pushing it when a label is read and popping it whenever the bar symbol is encountered, we ensure that the substring represents the correct prefix bar notation of a tree. The insertion operation is complete once the pushdown store contains only the initial pushdown store symbol; this is the only case in which the automaton can again start to advance in $\boldsymbol{p}$.

Note 9. Algorithm 3 can be modified to construct PDA that works with non-unit cost 1-degree edit distance. With unit cost operations, each transition in the PDA (corresponding to some edit operation) goes from a state with level l to a state with level $l+1$. With non-unit cost operations, transitions would go to states with a level that is increased accordingly by the cost of the operation.

4.2 1-degree finite automaton

Due to its restricted use of the pushdown store, we can transform the 1-degree PDA into an equivalent finite automaton. The PDA constructed by Algorithm 3 uses only symbol $\mathtt{c}$ for pushdown store operations (the initial pushdown symbol $\mathtt{Z}$ is never pushed). Moreover, pushdown store operations are only used for insertion operations.

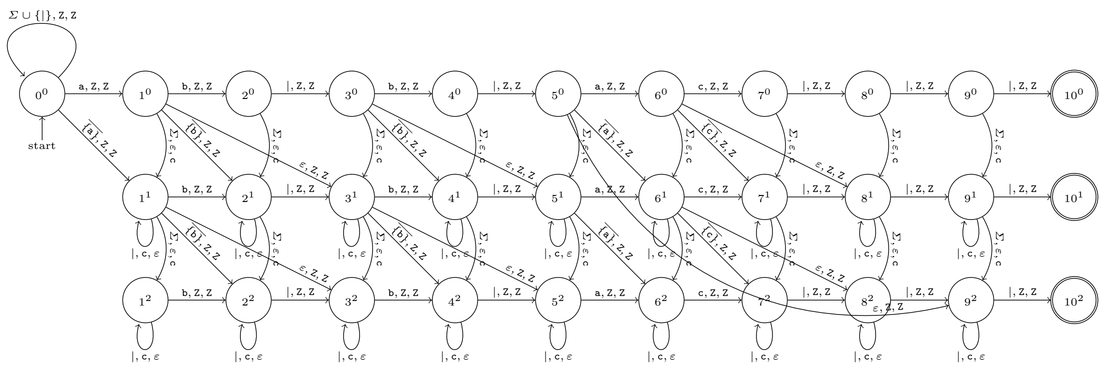


Figure 2: Transition diagram of the 1-degree pushdown automaton for tree pattern P from Figure 1a and $k = 2$. The double-circled nodes correspond to final states. The edge labeled x, y, z from state q_1 to state q_2 corresponds to transition $\delta(q_1, x, y) = \{(q_2, z)\}$. The complement $\overline{a}$ means $\Sigma \setminus \{a\}$.

Since the number of editing operations is limited by $k \le m$, the length of the pushdown store is also bounded by k. In other words, the pushdown store serves as a bounded counter. Therefore, we can represent each possible content of the pushdown store by a state. The construction of the 1-degree nondeterministic finite automaton is described by Algorithm 4. It reuses the $\mathcal{M}_{\text{PDA}}$ structure and construction steps. The only difference is the use of states j^l_c $(c > 0)$ representing a situation where the pushdown store contained c symbols.

An example of $\mathcal{M}_{\text{NFA}}$ is depicted in Figure 3. The set of active states of this automaton while reading the input tree T illustrated in Figure 1b is shown in Table 1.

Algorithm 4 Construction of 1-degree nondeterministic finite automaton.

```
Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \textsc{pref-bar}(P)$ for the tree pattern $P$ over alphabet $\Sigma$, a non-negative integer $k \le m$, the subtree jump table $\boldsymbol{S}_P$ for $\boldsymbol{p}$.
Output 1-degree nondeterministic finite automaton $\mathcal{M}_{\text{NFA}}$ for $P$ and $k$.
1  define states $Q = \{0^0_0\} \cup \{j^l_c : 1 \le j \le |\boldsymbol{p}| \wedge 0 \le l \le k \wedge 0 \le c \le k\}$
2  define final states $F = \{|\boldsymbol{p}|^l_0 : 0 \le l \le k\}$
3  add an initial loop for the bar symbol: $\delta(0^0_0, |) = \{0^0_0\}$
4  add initial state transitions for labels: $\delta(0^0_0, a) = \begin{cases} \{0^0_0, 1^0_0\} : a = \boldsymbol{p}[1] \\ \{0^0_0, 1^1_0\} : a \in \Sigma \setminus \{\boldsymbol{p}[1]\} \end{cases}$
5  for every pattern position $j : 2 \le j \le |\boldsymbol{p}|$:
6      for every allowed number of errors $l : 0 \le l \le k$:
7          $\delta((j-1)^l_0, \boldsymbol{p}[j]) = \{j^l_0\}$                                  (label or bar match)
8          $\delta((j-1)^l_0, a) = \{j^{l+1}_0 : l < k\} : a \in \Sigma \setminus \{\boldsymbol{p}[j]\}$           (relabel)
9          for each counter value $c : 0 \le c < k$:
10             $\delta((j-1)^l_c, a) = \delta((j-1)^l_c, a) \cup \{(j-1)^{l+1}_{c+1} : l < k\} : a \in \Sigma \setminus \{|\}$   (label insert)
11             $\delta((j-1)^l_{c+1}, |) = \{(j-1)^l_c\}$                               (bar insert)
12         $\delta((j-1)^l_0, \varepsilon) = \{(\boldsymbol{S}_P[j]-1)^{l+(\boldsymbol{S}_P[j]-j)/2}_0 : l + \frac{\boldsymbol{S}_P[j]-j}{2} \le k\} : \boldsymbol{S}_P[j] > j$   (delete)
13 return $\mathcal{M}_{\text{NFA}} = (Q, \Sigma \cup \{|\}, \delta, 0^0_0, F)$
```

Because any NFA can be algorithmically transformed into a DFA, a deterministic finite automaton can be used for inexact 1-degree tree pattern matching. In such case, the set of all positions $i \in \{1, \ldots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i] + 1 \ldots i]) \le k$ can be computed in $\mathcal{O}(n)$ time. However, the issue can be the size of the deterministic automaton, which can be exponential in the number of vertices of the tree pattern [4]. Therefore, in the next section, we also present how

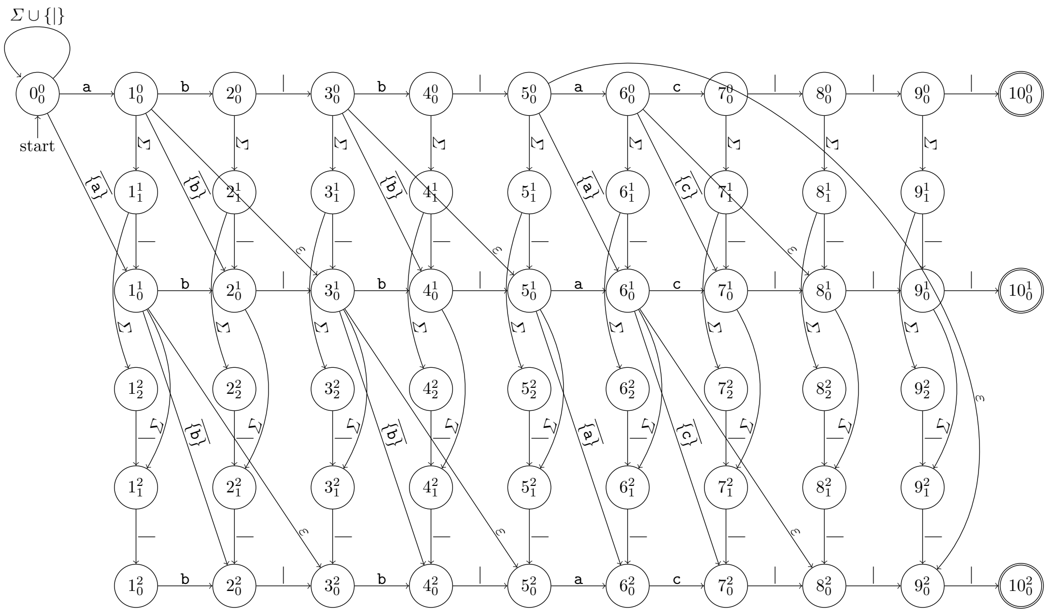


Figure 3: Transition diagram of the 1-degree NFA for tree pattern P illustrated in Figure 1a and $k = 2$. The double-circled nodes correspond to final states. The complement $\overline{a}$ means $\Sigma \setminus \{a\}$.

$\boldsymbol{t}$	a	a	a	c	|	|	|	a	b	|	b	a	c	|	|	|	a	c	|	|	|	|
0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0
	1_0^0	1_0^0	1_0^0	1_0^1	1_0^1	1_0^2	10_0^2	1_0^0	1_0^1	1_0^1	1_0^1	1_0^0	1_0^1	3_0^2	4_0^2	5_0^2	1_0^0	1_0^1	3_0^2	9_0^2	10_0^2	
		1_1^1	1_1^1	1_1^1	3_0^1	3_0^2			1_1^1	3_0^0	2_0^1	2_0^2	3_1^2	5_0^2			6_0^2	3_1^2	3_0^1			
		2_0^1	2_0^1	2_0^1	5_0^2	9_0^2			2_0^0	5_0^1	4_0^2	1_1^2	4_0^2	3_0^1				2_0^1	1_0^1			
		4_0^2	4_0^2	4_0^2	1_1^2				4_0^1	3_0^2	1_1^2	2_1^2	2_0^1	1_0^1				1_1^1	8_0^2			
		6_0^2	6_0^2	1_2^2	2_0^2				3_1^2		4_0^0	4_1^1	1_1^1	4_1^2				7_0^2				
		3_1^2	3_1^2	2_1^2	8_0^2						5_1^2		4_2^2									
			2_1^2	7_0^2	3_0^2						6_0^2											
			1_2^2	3_1^2																		

Table 1: Active states of $\mathcal{M}_{\text{NFA}}$ from Figure 3 for the input tree illustrated in Figure 1b.

dynamic programming can be used to simulate the nondeterministic finite automaton to achieve better space complexity.

5 Dynamic programming

An alternative approach to the use of $\mathcal{M}_{\text{DFA}}$ for inexact 1-degree tree pattern matching is a run simulation of $\mathcal{M}_{\text{NFA}}$ constructed by Algorithm 4. For such a simulation, an approach based on dynamic programming is presented in this section.

Algorithm 2 that uses $\mathcal{M}_{\text{NFA}}$ can be simulated by a three-dimensional array $\boldsymbol{D}$. Each field of $\boldsymbol{D}$ represents possibly active states of $\mathcal{M}_{\text{NFA}}$. More precisely, the first dimension $\boldsymbol{D}_i$ stands for the number of read symbols from $\boldsymbol{t}$; the second dimension $\boldsymbol{D}_{i,j}$ represents the portion of successfully matched pattern (i.e., when state j_c^l is active, $\boldsymbol{D}_{i,j}$ corresponds to j); finally, the third dimension $\boldsymbol{D}_{i,j,c}$ represents the (possibly) unbalanced symbol-bar pair (i.e., when state j_c^l is active, $\boldsymbol{D}_{i,j,c}$ corresponds to c). The value in $\boldsymbol{D}_{i,j,c}$ represents the distance—when state j^l_c is active, the value corresponds to l; the value ∞ represents the situation when no corresponding state of $\mathcal{M}_{\text{NFA}}$ is active. Each field value is computed from other fields value based on the transition function δ of $\mathcal{M}_{\text{NFA}}$.

The part of $\boldsymbol{D}$ recording computation before reading any symbol (i.e., $\boldsymbol{D}_{0,j,c}$) corresponds to the set of active states of $\mathcal{M}_{\text{NFA}}$: the initial state 0^0_0 only. Due to the self-loop in state 0^0_0, the initial state remains active after reading any symbol from the input. This corresponds to value 0 in $\boldsymbol{D}_{i,0,0}$. The initialization of $\boldsymbol{D}$ is formally given in (2).

$$\begin{aligned}
&\forall c, i : 0 \le c \le, 0 \le i \le 2n : \boldsymbol{D}_{i,0,c} = \begin{cases} 0 : c = 0, \\ \infty : c > 0 \end{cases} \\
&\forall c, j : 0 \le c \le k, 1 \le j \le 2m : \boldsymbol{D}_{0,j,c} = \infty
\end{aligned} \tag{2}$$

When matching a symbol without an edit operation, i.e., reading the same symbol from both $\boldsymbol{p}$ and $\boldsymbol{t}$, the transition in $\mathcal{M}_{\text{NFA}}$ goes from state $(j-1)^l_0$ to state j^l_0. Reading from both $\boldsymbol{p}$ and $\boldsymbol{t}$ means increasing both i and j dimensions in $\boldsymbol{D}$. Matching symbols without an edit operation in $\boldsymbol{D}$ is formally given in (3).

$$\boldsymbol{D}_{i,j,0} = \boldsymbol{D}_{i-1,j-1,0} : \boldsymbol{t}[i] = \boldsymbol{p}[j] \wedge 1 \le i \le 2n \wedge 1 \le j \le 2m \tag{3}$$

Representation of vertex relabeling operation in $\mathcal{M}_{\text{NFA}}$ is similar to matching symbols without an edit operation. Relabeling vertices in $\boldsymbol{D}$ is formally given in (4).

$$\boldsymbol{D}_{i,j,0} = \boldsymbol{D}_{i-1,j-1,0} + 1 : \boldsymbol{t}[i], \boldsymbol{p}[j] \in \Sigma \wedge 1 \le i \le 2n \wedge 1 \le j \le 2m \tag{4}$$

In $\mathcal{M}_{\text{NFA}}$, leaf deletion operation is represented by an ε-transition: skipping part of $\boldsymbol{p}$ (the length of the skip is given by the subtree jump table $\boldsymbol{S}_P$) while reading nothing from $\boldsymbol{t}$. These ε-transitions can be (using standard algorithm) replaced by symbol transitions. More precisely, the ε-transition from state $q^{l_1}_0$ to state $r^{l_2}_0$ can be interpreted as transition from state $q^{l_1}_0$ using symbol $\boldsymbol{p}[q+1]$ to state $(r+1)^{l_2}_0$. Also, considering the sequence of operations delete and relabel, the ε-transition can be interpreted as transitions from state $q^{l_1}_0$ using symbols $\Sigma \setminus \{\boldsymbol{p}[q+1]\}$ to state $(r+1)^{l_2+1}_0$. By contrast, the sequence of transitions for operations delete and insert is not considered in the simulation, as it cannot find more matches than single operation relabel. While $\mathcal{M}_{\text{NFA}}$ skips a leaf "forward", during the computation of a value in $\boldsymbol{D}$, we look "backward". Note that in $\mathcal{M}_{\text{NFA}}$, there can be chains of ε-transitions that correspond to deleting multiple leaves (siblings). This is done in $\boldsymbol{D}$ by multiple evaluation of $\boldsymbol{S}_P$. Deleting from the pattern in $\boldsymbol{D}$ is formally given in (5) and (6).

$$\begin{aligned}
\boldsymbol{D}_{i,j,0} = &\boldsymbol{D}_{i-1,\boldsymbol{S}_P[h],0} + \frac{j - \boldsymbol{S}_P[h] + 1}{2} : \boldsymbol{t}[i] = \boldsymbol{p}[j] \wedge \boldsymbol{p}[j-1] = | \wedge \\
&\wedge 1 \le i \le 2n \wedge 2 \le j \le 2m \wedge 1 \le h \le 2m \wedge \boldsymbol{S}_P[h] < j
\end{aligned} \tag{5}$$

$$\begin{aligned}
\boldsymbol{D}_{i,j,0} = &\boldsymbol{D}_{i-1,\boldsymbol{S}_P[h],0} + \frac{j - \boldsymbol{S}_P[h] + 2}{2} : \boldsymbol{t}[i], \boldsymbol{p}[j] \in \Sigma \wedge \boldsymbol{t}[i] \neq \boldsymbol{p}[j] \wedge \\
&\wedge \boldsymbol{p}[j-1] = | \wedge 1 \le i \le 2n \wedge 2 \le j \le 2m \wedge 1 \le h \le 2m \wedge \boldsymbol{S}_P[h] < j
\end{aligned} \tag{6}$$

In $\mathcal{M}_{\text{NFA}}$, leaf insertion operation is represented by a pair of transitions and states: from state j^l_c to state j^{l+1}_{c+1} (read a vertex label from $\boldsymbol{t}$ and record an unbalanced symbol) and from state j_c^l to state j_{c-1}^l (read the bar from $\boldsymbol{t}$ and record a balanced symbol-bar pair). It is not possible to use any transition besides insert until the inserted labels and bars are balanced. To track the balance between inserted labels and bars, the third dimension of $\boldsymbol{D}$ is used. Inserting into the pattern in $\boldsymbol{D}$ is formally given in (7) and (8).

$$\boldsymbol{D}_{i,j,c} = \boldsymbol{D}_{i-1,j,c-1} + 1 : \boldsymbol{t}[i] \in \Sigma \land 1 \leq i \leq 2n \land 1 \leq j \leq 2m \land 1 \leq c \leq k \tag{7}$$

$$\boldsymbol{D}_{i,j,c} = \boldsymbol{D}_{i-1,j,c+1} : \boldsymbol{t}[i] = | \land 1 \leq i \leq 2n \land 1 \leq j \leq 2m \land 0 \leq c < k \tag{8}$$

The previous expressions do not limit the values stored in the cells in $\boldsymbol{D}$. However, only values between 0 and k are useful. This is summarized in the following proposition.

Proposition 10 (Distance value representation in $\boldsymbol{D}$-table). *In $\mathcal{M}_{\text{NFA}}$, there exists no state j_c^l with $l > k$. Therefore, the field values in the $\boldsymbol{D}$-table greater than k can be represented by ∞.*

Among the active states in $\mathcal{M}_{\text{NFA}}$, there can be those of the same depth but different level; for example, states 5_0^0 and 5_0^2. (See Example 11 that shows such a situation.) However, to solve the inexact 1-degree tree pattern matching problem, we do not need multiple integers to represent multiple possibly active states j_0^l and $j_0^{l'}$ in $\boldsymbol{D}_{i,j,0}$. This is summarized in Lemma 12.

Example 11. Let $\mathcal{M}_{\text{NFA}}$ be the NFA depicted in Figure 3. After reading string `ab|b|`, the set of active states of $\mathcal{M}_{\text{NFA}}$ is $\{0_0^0, 1_0^2, 3_0^1, 5_0^0, 5_0^2\}$.

Lemma 12. *Storing only single integer in every field $\boldsymbol{D}_{i,j,c}$ is sufficient for correct solution of the problem from Definition 3.*

Proof. Although $\mathcal{M}_{\text{NFA}}$ can have multiple active states for the same c and j, only states with the smallest l are interesting for solving the problem from Definition 3. If the state $j_c^{l'}$ where $l' > l$ is not considered active, no occurrence of the pattern can be missed, as due to regular structure of $\mathcal{M}_{\text{NFA}}$, there is no additional path from such state $j_c^{l'}$ to a final state compared to state j_c^l. Storing only the minimum integer in $\boldsymbol{D}$ corresponds to considering only the state with the minimum l active.

The simulation of $\mathcal{M}_{\text{NFA}}$ for inexact 1-degree tree pattern matching is summarized in Algorithm 5. See an example of the computation in Table 2.

		1	2	3	4	5	6	7	8	9	10	11	
		a	a	a	c	|	|	|	a	b	|	b	...
	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	0, ∞, ∞	
a	∞, ∞, ∞	0, ∞, ∞	0, 1, ∞	0, 1, 2	1, 1, 2	1, 2, ∞	2, ∞, ∞	∞, ∞, ∞	0, ∞, ∞	1, 1, ∞	1, ∞, ∞	1, 2, ∞	
b	∞, ∞, ∞	∞, ∞, ∞	1, ∞, ∞	1, 2, ∞	1, 2, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	0, ∞, ∞	∞, ∞, ∞	1, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	1, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	0, ∞, ∞	∞, ∞, ∞	
b	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	2, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	1, ∞, ∞	∞, ∞, ∞	0, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	1, ∞, ∞	∞, ∞, ∞	
a	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	
c	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	2, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	

Table 2: Example of dynamic programming computation for $k = 2$, the tree pattern from Figure 1a, and a part of the input tree from Figure 1b. An occurrence is found at position $i = 7$.

Algorithm 5 Simulation of 1-degree nondeterministic finite automaton.

Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for tree pattern P over alphabet Σ, a string $\boldsymbol{t}$ of length $2n$ such that $\boldsymbol{t} = \text{PREF-BAR}(T)$ for tree T over alphabet Σ, a non-negative integer k such that $k \leq m$.

Output All positions $i \in \{1, \dots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i]+1 \dots i]) \leq k$.

```
compute the subtree jump table $\boldsymbol{S}_P$ for $P$ using Algorithm 1
initialize $\boldsymbol{D}$ according to (2)
for each position index $i$ of $\boldsymbol{t}$:
    for each position index $j$ of $\boldsymbol{p}$:
        compute cell $(i, j, c)$ of $\boldsymbol{D}$ as the minimum from (applicable only)
            match according to (3) ($c = 0$ only)
            relabel according to (4) ($c = 0$ only)
            for each counter value $c : 0 \leq c \leq k$:
                insert into the pattern (the bar is pending) according to (7)
                insert into the pattern (match the bar) according to (8)
            delete subtree(s) from the pattern ($c = 0$ only):
                $h = j - 1$
                while $\boldsymbol{S}_P[h] < j$:
                    consider value according to (5) and (6)
                    $h \leftarrow \boldsymbol{S}_P[h]$
    if $\boldsymbol{D}_{i,|\boldsymbol{p}|,0} \leq k$:
        output $i$
```

Theorem 13 (Space complexity). *The problem from Definition 3 can be solved using $\mathcal{O}(km)$ space by Algorithm 5.*

Proof. During the computation of the value of $\boldsymbol{D}_{i,j,c}$, only two columns (i-th and $(i-1)$-th) of $\boldsymbol{D}$ are needed in the memory. Each column contains $2m+1$ rows (each for one position in $\boldsymbol{t}$ plus the 0-th row). Each row stores $k+1$ integers (each for one distinct c value), while their possible and useful values are between 0 and k (plus one additional for all the values greater than k, according to Proposition 10), thus each of these integers may be represented by $\lfloor \log_2(k+1) \rfloor + 1$ bits. Therefore, the entire $\boldsymbol{D}$-table requires $2(2m+1)(k+1)(\lfloor \log_2(k+1) \rfloor + 1)$ bits. Also, $\boldsymbol{S}_P$ and $\boldsymbol{p}$ need to be stored. Array $\boldsymbol{S}_P$ contains $2m$ integers of values between 0 and $2m+1$, thus requires $2m(\lfloor \log_2(2m+1) \rfloor + 1)$ bits. String $\boldsymbol{p}$ contains $2m$ characters that are either the bar or from Σ, thus requires $2m(\lfloor \log_2(|\Sigma|+1) \rfloor + 1)$ bits. In total, it is $2(2km+2m+k+1)(\lfloor \log_2(k+1) \rfloor + 1) + 4m(\lfloor \log_2(2m+1) \rfloor + \lfloor \log_2(|\Sigma|+1) \rfloor + 2)$ bits, i.e., $\mathcal{O}(km \log k + \log |\Sigma|)$. When considering integer and symbol encoding independent of the tree size and the alphabet, we get $\mathcal{O}(km)$.

Theorem 14 (Time complexity). *The problem from Definition 3 can be solved in $\mathcal{O}(kmn)$ time by Algorithm 5.*

Proof. The subtree jump table is computed in $\mathcal{O}(m)$ time. There are $\mathcal{O}(mn)$ match and relabel computations, each needs $\mathcal{O}(1)$ time. There are $\mathcal{O}(kmn)$ insert computations, each in $\mathcal{O}(1)$ time. The number of delete computations depends, besides mn, on number of subtree skips, which is $\mathcal{O}(m)$. Effectively, the number of subtree skips is limited by k, as there is no point to skip subtree(s) with more than k vertices. Therefore, there are $\mathcal{O}(kmn)$ delete computations, each takes $\mathcal{O}(1)$ time.

Recall that the bar position i in $\boldsymbol{t}$ returned by Algorithm 5 corresponds to the vertex v in T where v is the root of the found subtree (therefore, it is a correct solution of the problem from Definition 3). Additionally, it is possible to obtain v from i in $\mathcal{O}(1)$ time while still having the $\mathcal{O}(kmn)$ time complexity of Algorithm 5, e.g., by adding pointers to vertices of T into $\text{PREF-BAR}(T)$. This could be done at the cost of adding $2n$ to space complexity.

Note 15. Algorithm 5 can be extended for non-unit cost operations in a straightforward way. When computing the value of a field of $\boldsymbol{D}$, instead of adding one (for an edit operation), we add the value corresponding to the cost of the used edit operation.

6 Conclusions

Inspired by techniques from string matching, we showed that the automata approach can also be used to solve the inexact tree pattern matching problem. To process trees using (string) automata, we represented trees as strings using the prefix bar notation. We considered labeled ordered (unranked) trees and 1-degree edit distance where tree operations are restricted to vertex relabeling, leaf insertion, and leaf deletion. For simplicity, we used the unit cost for all operations. However, the extension of our approach to non-unit cost distance was also discussed.

Given a tree pattern P with m vertices, an input tree T with n vertices, and $k \leq m$ representing the maximal number of errors, we first proposed a pushdown automaton that can find all subtrees in T that match P with up to k errors. Then, we discussed that the pushdown automaton can be transformed into a finite automaton due to its restricted use of the pushdown store. The deterministic version of the finite automaton finds all occurrences of the tree pattern in time linear to the size of the input tree.

We also presented an algorithm based on dynamic programming, which was a simulation of the nondeterministic finite automaton. The space complexity of this approach is $\mathcal{O}(mk)$ and the time complexity is $\mathcal{O}(kmn)$, where m is the number of vertices of the tree pattern, n is the number of vertices of the input tree, and $k \leq m$ represents the number of errors allowed in the pattern. In the paper, we also presented the algorithm for subtree jump table construction for a tree in prefix bar notation where the arity (rank) of each vertex is not known in advance.

In future work, we aim to study the space complexity of the DFA and the time complexity of its direct construction in detail. We also want to experimentally evaluate our algorithms. Bit parallelism can also be explored as way of simulating the NFA for tree pattern matching.

References

1. P. Bille: *Pattern Matching in Trees and Strings*, PhD thesis, University of Copenhagen, 2007.
2. K. Bringmann, P. Gawrychowski, S. Mozes, and O. Weimann: *Tree edit distance cannot be computed in strongly subcubic time (unless APSP can).* ACM Trans. Algorithms, 16(4) 2020.
3. E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann: *An optimal decomposition algorithm for tree edit distance.* ACM Trans. Algorithms, 6(1) Dec. 2010, pp. 1–19.
4. J. Hopcroft, R. Motwani, and J. Ullman: *Introduction to automata theory, languages, and computation*, Pearson Education, Harlow, Essex, 3 ed., 2014.
5. J. Janoušek: *Arbology: Algorithms on trees and pushdown automata*, Habilitation thesis, Brno University of Technology, 2010.
6. B. Melichar: *Approximate string matching by finite automata*, in Computer Analysis of Images and Patterns, Springer Berlin Heidelberg, 1995, pp. 342–349.
7. G. Navarro: *A guided tour to approximate string matching.* ACM Comput. Surv., 33(1) 2001, pp. 31–88.

8. S. M. Selkow: *The tree-to-tree editing problem.* Inf. Process. Lett., 6(6) 1977, pp. 184–186.
9. E. Šestáková, B. Melichar, and J. Janoušek: *Constrained approximate subtree matching by finite automata*, in Proceedings of the Prague Stringology Conference 2018, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2018, pp. 79–90.
10. K.-C. Tai: *The Tree-to-Tree correction problem.* J. ACM, 26(3) 1979, pp. 422–433.
11. J. Trávníček: *(Nonlinear) Tree Pattern Indexing and Backward Matching*, PhD thesis, Faculty of Information Technology, Czech Technical University in Prague, 2018.
12. K. Zhang, R. Statman, and D. Shasha: *On the editing distance between unordered labeled trees.* Inf. Process. Lett., 42(3) May 1992, pp. 133–139.