Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array

Jan Trávníček, Robin Obůrka, Tomáš Pecka*, and Jan Janoušek**

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Praha 6
Czech Republic
{Jan.Travnicek, oburkrob, Tomas.Pecka, Jan.Janousek}@fit.cvut.cz

Abstract. We define a tree pattern border array as a property of linearised trees analogous to border arrays from the string domain. We use it to define a new forward tree pattern matching algorithm for ordered trees, which finds all occurrences of a single given linearised tree pattern in a linearised input tree. As with the classical Morris-Pratt algorithm, the tree pattern border array is used to determine shift lengths in the searching phase of the tree pattern matching algorithm. We compare the new algorithm with the best performing previously existing algorithms based on backward linearised tree pattern matching algorithms, (non-)linearised tree pattern matching algorithms using finite tree automata or stringpath matchers. We show that the presented algorithm outperforms these for single tree pattern matching.

Keywords: tree processing, tree linearisation, Morris-Pratt algorithm

1 Introduction

Trees are one of the fundamental data structures used in Computer Science and the theory of formal tree languages has been extensively studied and developed since the 1960s [9,11]. Tree pattern matching on node-labeled trees is an important algorithmic problem with applications in many tasks such as compiler code selection, interpretation of nonprocedural languages, implementation of rewriting systems, or markup languages processing. Trees can be represented as a string by various linearisations [16]. Such a linear notation can be obtained by a corresponding tree traversal. Moreover, every sequential algorithm on a tree traverses its nodes in a sequential order, which corresponds to some linear notation. Such a linear representation need not be built explicitly.

Tree patterns are trees whose leaves can be labelled by a special wildcard, the nullary symbol S, which serves as a placeholder for any subtree. Since the linear notation of a subtree of a tree is a substring of the linear notation of that tree, the subtree matching and tree pattern matching problems are in many ways similar to the string pattern matching problem. We note that the tree pattern matching problem is more complex than the string matching one because there can be at most $n(n-1)/2$ distinct substrings of a string of size n, whereas there can be at most $2^{n-1}+n$ distinct tree patterns which match a tree of size n [13].

* This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/208/OHK3/3T/18.

** 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".

Jan Trávníček, Robin Obůrka, Tomáš Pecka, Jan Janoušek: Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array, pp. 61–73.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

Border array and borders in general are one of well-studied string properties used in various efficient single pattern string pattern matching algorithms (Morris-Pratt and Knuth-Morris-Pratt algorithms, etc.) [17,15,4], and multi-pattern ones (Aho-Corasick) [1].

For unrestricted tree pattern sets, among the fastest pattern matching algorithms in practice are algorithms based on deterministic frontier-to-root (bottom-up) tree automata (DFRTAs) [6,8,12] and Hoffmann-O'Donnell-style stringpath matchers [2,12]. The latter uses Aho-Corasick pattern matching algorithm, but processes the tree in its natural representation.

Tree pattern matching algorithms processing a linearised representation of a tree exist as well. They either use pushdown automata [10] or, in the case of single pattern matching, adapt ideas known from backward string pattern matching [20]. However, no existing tree pattern matching algorithm uses the border array constructed for a linearised tree pattern directly on linearised trees.

The best performing algorithms using deterministic tree automata or deterministic pushdown automata generally run in $\Theta(n+occ)$ time, where n is the size of the subject tree [8]. On the other hand, the backward tree pattern matching algorithm require $\Omega(n/m + occ)$ and $O(m \cdot n + occ)$ time, where m is the size of the tree pattern [20].

While modifying forward string pattern matching to forward subtree matching (searching for occurrences of given subtrees) is straightforward, this is not the case for forward tree pattern matching, where complications arise due to the use of nullary symbol S and matched subtrees being possibly recursively nested.

In this paper, a definition of tree pattern border array is presented. The size of the tree pattern border array table is linear with the size of the linearised pattern. The tree pattern border array is later used in the design of a new forward tree pattern matching algorithm. The presented forward tree pattern matching algorithm is a modification of the Morris-Pratt algorithm from the string domain and even though it does not keep the linear complexity of the searching phase with respect to the size of the subject tree n, it often performs better in practice than sublinear backward tree pattern matching algorithm [20]. Even though the Knuth-Morris-Pratt algorithm is a straight-forward extension of the Morris-Pratt algorithm, this is not the case in trees and therefore the presented algorithm is based on the Morris-Pratt algorithm. Our experimental results show that the presented algorithm outperforms even the aforementioned DFRTAs and Aho-Corasick stringpath matchers in single-pattern matching case.

The paper is organised as follows: Section 2 recalls basic definitions and properties of trees and the Morris-Pratt algorithm. Section 3 defines the tree pattern border array and presents the new forward linearised tree pattern matching algorithm based on the Morris-Pratt algorithm. Section 4 compares the results with other state-of-the-art algorithms. Some concluding remarks are presented in Section 5.

2 Basic notions

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

A *string* s is a sequence of n symbols $s_1 s_2 s_3 \cdots s_n$ from a given alphabet, where n is the size of the string. A sequence of zero symbols is called the empty string. The empty string is denoted by symbol ε. A *s[i .. j]* is a substring (factor) $s_i \cdots s_j$ of s, note that ε substring of s is obtained by $s[i..i-1]$. A prefix and a suffix of a string s of length n is a substring $s[1..j]$ and $s[i..n]$, respectively, where $1 \leq j \leq n$ and $1 \leq i \leq n$. A proper prefix and a proper suffix of s is a prefix and a suffix, respectively, which is not equal to s.

Based on concepts and notations from graph theory [3], a *rooted tree* t is a weakly connected acyclic directed graph $t = (V, E)$ with a special node $r \in V$, called the *root*, such that r has in-degree 0, all other nodes of t have in-degree 1, and there is just one path from the root r to every $f \in V$ and $f \neq r$, where a path from a node f_0 to a node f_n is a sequence of nodes $(f_0, f_1, ..., f_n)$ for $n > 0$ and $(f_i, f_{i+1}) \in E$ for each $0 \leq i < n$. Nodes of a rooted tree with out-degree 0 are called *leaves*.

A node g is a *direct descendant* of node f if a pair $(f, g) \in E$ and *descendant* of node f if $(f, f_1, f_2, ..., f_n, g)$ for $n \geq 0$ is a path in t.

A tree is an *ordered, ranked and labelled* rooted tree with nodes labelled by symbols from a ranked alphabet satisfying that the out-degree of a node f labelled by symbol $\ell \in \mathcal{A}$ equals $Arity(\ell)$ and with the direct descendants $g_1, g_2, \ldots, g_n$ of a node f ordered.

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

1. V' is an nonempty subset of V,
2. $E' = (V' \times V') \cap E$, and
3. no node of $V \setminus V'$ is a descendant of a node in V'.

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

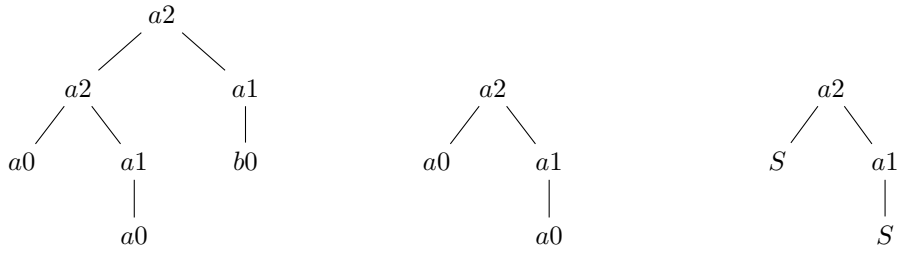
1. $pref(\ell) = \ell 0$ if ℓ is a leaf,
2. $pref(t) = \ell n\ pref(t_1)\ pref(t_2) \cdots pref(t_n)$, where ℓ is the label of the root of tree t, $n = Arity(\ell)$ and $t_1, t_2, \ldots, t_n$ are direct descendants of the root of t.

Let $s = s_1 s_2 \cdots s_n$, $n \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the *arity checksum* $ac(s) = \sum_{i=1}^{n} Arity(s_i) - n + 1$. Let $pref(t)$ and s of size n be a tree t in prefix notation and a substring of $pref(t)$, respectively. Then, s is the prefix notation of a subtree of t, if and only if $ac(s) = 0$, and $ac(s[1..j]) \geq 1$ for all $1 \leq j < n$ [16].

Example 1. Consider a tree t_{1r} over a ranked alphabet $\mathcal{A} = \{a2, a1, a0\}$, $pref(t_{1r}) =$ $a2\ a2\ a0\ a1\ a0\ a1\ a0$. Trees can be represented graphically, as is done for tree t_{1r} in Figure 1a.

To define a *tree pattern*, we use a special wildcard symbol $S \notin \mathcal{A}$, $Arity(S) = 0$, which serves as a placeholder for any subtree. A tree pattern is defined as a labelled ordered tree over ranked alphabet $\mathcal{A} \cup \{S\}$. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. Note that the wildcard symbol can only label leaves of tree pattern.

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



(a) Tree t_{1r} over a ranked alphabet from Example 1.

(b) Subtree (tree) p_{1r} from Example 2.

(c) Tree pattern p_{2r} from Example 2.

Figure 1. Trees and tree patterns over a ranked alphabet from Example 1 and Example 2.

Example 2. Consider a tree t_{1r} from Example 1, which is illustrated in Figure 1a. Consider a subtree p_{1r} over ranked alphabet $\mathcal{A}$, $pref(p_{1r}) = a2\ a0\ a1\ a0$ and a tree pattern p_{2r} over ranked alphabet $\mathcal{A} \cup \{S\}$, $pref(p_{2r}) = a2\ S\ a1\ S$, which are illustrated in Figure 1b and Figure 1c. Tree pattern p_{1r} occurs once in t_{1r} — match is at node 2 of t_{1r}. Tree pattern p_{2r} occurs twice in t_{1r}, it matches at nodes 1 and 2 of t_{1r}.

2.1 Forward string pattern matching algorithm

A classical representant of a forward string pattern matching algorithm is the Morris-Pratt algorithm [17,15]. The algorithm's execution splits into preprocessing and searching phases.

A precomputed table to determine the length of shift and also to know the number of symbols not required to be matched again in the following match attempt is constructed during the preprocessing phase. The table used in the algorithm is the border array [18,5]. The border array for a string of length m can be constructed in $O(m)$ time.

Note the border array is defined as in [5] without explicitly defining borders to ease transition from strings to linearised tree patterns where concepts of prefix and suffix do not apply.

Definition 3 (border array ([5]) $\mathcal{B}(s)$). *Let s be a string of length n. The border array $\mathcal{B}(s)$ is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}(s)[1] = 0$ and otherwise $\mathcal{B}(s)[i] = max(\{0\} \cup \{k : s[1..k] = s[i-k+1..i] \wedge k \geq 1 \wedge i-k+1 > 1\})$.*

From the definition, the substrings $s[1..k]$ and $s[i-k+1..i]$ are referred to as borders.

Example 4. Consider a string $s = ababc$ of length 5. The border array $\mathcal{B}(s) = 0, 0, 1, 2, 0$.

The algorithm locates all occurrences of a pattern in a text in a searching phase. Note that the presented algorithms deviates from the classical presentation of the Morris-Pratt algorithm to simplify the transition from its string version to a tree version.

The searching phase of the Morris-Pratt algorithm has time complexity $O(m+n)$. The algorithm makes at most $2n-1$ comparisons during the searching phase [17,15]. Line 10 of Algorithm 1 represents the shift and line 11 of Algorithm 1 represents a carry of information of how many symbols do not need to be matched again.

Algorithm 1: Morris-Pratt matching function.

Input: The subject string s of size n, the pattern string p of size m, the border array table $\mathcal{B}(p)$
Result: A list of matches.

```
1  begin
2      i := 0
3      j := 1
4      while i ≤ n − m do
5          while j ≤ m and s[i + j] = p[j] do
6              j += 1
7          end
8          if j > m then yield i + 1
9          if j ≠ 1 then
10             i += j − B(p)[j − 1] − 1
11             j := B(p)[j − 1] + 1
12         else
13             i += 1
14         end
15     end
16 end
```

3 Forward Linearised Tree Pattern Matching

The pattern occurrences in linear notation are represented by substrings of trees in the linear notation. They can contain "gaps" given by a special wildcard symbol S, which serves as a placeholder for any subtree.

The string pattern matching algorithm must be modified to handle these gaps. The wildcard symbol S represents any subtree. Moreover, matched subtrees may be possibly nested. The wildcard symbol S therefore needs special care.

In order to handle these gaps a *Subtree jump table* structure is defined. The structure was introduced in [14].

Definition 5 (subtree jump table for prefix notation $sjt(pref(t))$). *Let t and $pref(t) = \ell_1\ell_2\cdots\ell_n$, $n \geq 1$, be a tree and its prefix notation, respectively. A* subtree jump table for prefix notation *$sjt(pref(t))$ is a mapping from a set $\{1..n\}$ into a set $\{2..n+1\}$. If $\ell_i\ell_{i+1}\cdots\ell_{j-1}$ is the prefix notation of a subtree of tree t, then $sjt(pref(t))[i] = j$, $1 \leq i < j \leq n+1$.*

Note that the definition of subtree jump table for prefix notation is applicable to tree patterns without changes as well.

Informally, the subtree jump table contains an entry for each subtree r of tree t. The entry for subtree r is located at the position of its root in the prefix notation $pref(t)$ of the tree t. The entry stores an index into string $pref(t)$ to a symbol that is one after the last of the subtree r, i.e., position of the root of r plus the length of $pref(r)$. This structure has the same size as the prefix notation of the tree t. The construction time is $O(n)$ where n is the length of $pref(t)$ [14].

3.1 Linearised tree border

The Morris-Pratt algorithm uses shift heuristics based on borders. One needs to define a tree pattern border array in order to obtain similar shift heuristics in the Morris-Pratt algorithm modification for trees.

In order to define the tree pattern border array, first, let us define equivalence of linear representations of a tree pattern and its factor.

Definition 6 (matches relation *s matches r*)**.** *Let S be a wildcard symbol representing a complete subtree in prefix ranked notation of trees. Two strings s and r are in relation* matches *if:*

$$
\begin{array}{lll}
s = \ell s' & r = \ell r' & \textit{and } s' \textit{ matches } r' \\
 & & \textit{and } \ell \in \mathcal{A}, \\
s = Ss' & r = Sr' & \textit{and } s' \textit{ matches } r', \\
s = \ell_1 \cdots \ell_m s' & r = Sr' & \textit{and } ac(\ell_1 \cdots \ell_m) = 0 \\
 & & \textit{and } \forall k, 1 \leq k < m, ac(\ell_1 \cdots \ell_k) \geq 1 \\
 & & \textit{and } s' \textit{ matches } r', \\
s = Ss' & r = \ell_1 \cdots \ell_m r' & \textit{and } ac(\ell_1 \cdots \ell_m) = 0 \\
 & & \textit{and } \forall k, 1 \leq k < m, ac(\ell_1 \cdots \ell_k) \geq 1 \\
 & & \textit{and } s' \textit{ matches } r', \\
s = Ss' & r = \ell_1 \cdots \ell_m & \textit{and } \forall k, 1 \leq k \leq m, ac(\ell_1 \cdots \ell_k) \geq 1, \\
s = \varepsilon \textit{ or } r = \varepsilon & &
\end{array}
$$

The two strings s and r are in relation matches if symbols of strings r and s compare on corresponding positions, wildcards in string r correspond to complete subtrees in string s, and wildcards in string s correspond to complete subtrees within string r and possibly incomplete subtree at the end of string r. Note that the corresponding symbols may not be on the same indexes in strings s and r as a subtree corresponding to wildcard S may be longer than a single symbol.

Informally, the two strings s and r representing prefixes of prefix notation of a tree pattern are in relation matches if the corresponding tree pattern subgraphs structurally and symbol-wise align.

Definition 7 (tree pattern border array $\mathcal{B}(pref(p))$). *Let $pref(p)$ be a tree pattern in a prefix notation of length n. The $\mathcal{B}(pref(p))$ is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}(pref(p))[1] = 0$ and otherwise $\mathcal{B}(pref(p))[i] = max(\{0\} \cup \{k : pref(p) \textit{ matches } pref(p)[i-k+1..i] \wedge k \geq 1 \wedge i-k+1 > 1\})$.*

Notice that the definition of tree pattern border array is different from the definition of the border array for strings in the use of the matches relation and in use of the complete linear representation of tree pattern as its left argument.

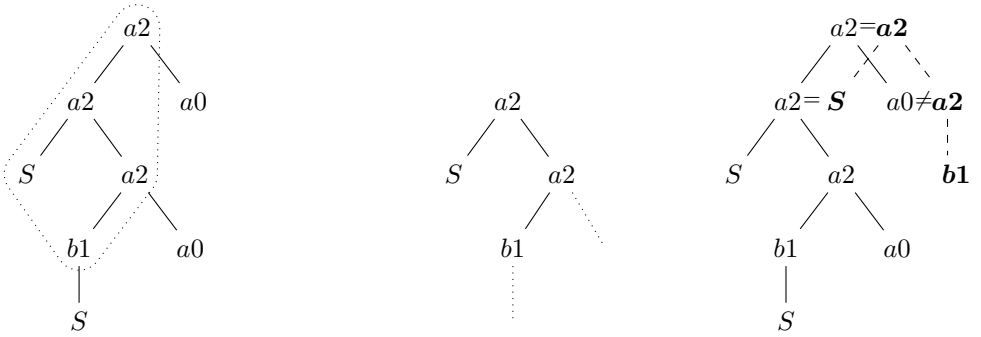
	1	2	3	4	5	6	7	8			
pref(p)	$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$			
pref(p)[2..5]	$a2$	$\vdash$		S			$\dashv$	$a2$			mismatch at position 8
pref(p)[3..5]	$\vdash$			S				$\dashv$	$a2$	$b1$	match
pref(p)[4..5]	$a2$	$b1$									mismatch at position 2
pref(p)[5..5]	$b1$										mismatch at position 1
pref(p)[6..5]											match because pref(p)[6..5] = ε

Table 1. Trace of naive computation of $pref(p)$ *matches* $pref(p)[j+1..5]$ for $1 \leq j \leq 5$ and $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$.

Example 8. Let $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$ be a prefix notation of a tree pattern p. In order to compute $\mathcal{B}(pref(p))[5]$ according to Definition 7 the computation of $pref(p)$ *matches* $pref(p)[j+1..5]$ for $1 \leq j \leq 5$ is necessary.

The minimal j for which $pref(p)$ *matches* $pref(p)[j+1..5]$ is 2, therefore, the $\mathcal{B}(pref(p))[5] = 5-2 = 3$. The process of computation of relation matches is depicted in Table 1.

The visualisation of the alignment of $pref(p)$ and $pref(p)[2..5]$, internally computed by the *matches* relation is depicted in Figure 2 on non-linearised tree pattern p and tree pattern p subgraph corresponding to $pref(p)[2..5]$. The other alignment visualisations can be depicted in a similar manner.



(a) Tree pattern p and the subgraph of p corresponding to the prefix of $pref(p)$ relevant to the computation of relation *matches*.

(b) The subgraph of p corresponding to $pref(p)[2..5]$.

(c) Visualisation of the alignment.

Figure 2. Visualisation of a single alignment of p and subgraph corresponding to $pref(p)[2..5]$ in computation of $pref(p)$ *matches* $pref(p)[j+1..5]$) for $1 \leq j \leq 5$.

The $\mathcal{B}(pref(p))$ can naively be implemented using Definition 7. The relation *matches* is easy to implement using iteration with the help of $sjt(pref(p))$. The naive computation requires $O(m^3)$ time, where m is the length of $pref(p)$.

The computation of $\mathcal{B}(pref(p))$ can also be done in quadratic time with respect to the length of $pref(p)$. All the factors beginning at *offset* are tested in one iteration through the pattern $pref(p)$ to determine the result of $pref(p)$ *matches* $pref(p)[offset+1..i]$ $\forall\, 1 \leq offset \leq i \leq m$. This approach avoids repeating some comparisons. One iteration through $pref(p)$ takes $O(m)$ time and has to be repeated $m-1$ times for different values of *offset*. The procedure of creating $\mathcal{B}(pref(p))$ is formalised in Algorithm 2.

3.2 Forward linearised tree pattern matching algorithm

One of the usages of the border array is in the Morris-Pratt algorithm. The string version of the algorithm consists of two alternating components – occurrence check and shift computation. Both can be adapted to trees with the later requiring the tree pattern border array defined.

The Algorithm 3 is a modification of the Morris-Pratt algorithm for strings. It uses index j into the $pref(p)$ and two indexes i and *offset* into the $pref(s)$. The index i holds the position of the current attempt and *offset* holds the position of currently compared symbol. Index *offset* is needed due to the "elasticity" of the pattern caused by the wildcards. The shift distances are precomputed but otherwise the computation is unchanged, however, the number of symbols that are not required to be matched again is derived from the border array and is limited by the first occurrence of the wildcard due to variability of the subtree in place of the wildcard.

Algorithm 2: Computation of tree pattern border array.

Input: A pattern tree $pref(p)$ (pattern) of size m and a vector of integers $sjt(pref(p))$
Output: A vector of integers $\mathcal{B}(pref(p))$ indexed as $[1..m]$

```
1  begin
2      for i := 1 to m do B(pref(p))[i] := 0
3      for offset := 1 to m do
4          i := 1                                   /* index into full pref(p) */
5          j := offset + 1                          /* index into a factor of pref(p) */
6          while True do
7              if i > m then
8                  while j ≤ m do
9                      B(pref(p))[j] := max(B(pref(p))[j], j − offset)
10                     j += 1
11                 end
12                 break
13             else if j > m then
14                 break
15             else if pref(p)[i] = pref(p)[j] then
16                 B(pref(p))[j] := max(B(pref(p))[j], j − offset)
17                 i += 1
18                 j += 1
19             else if pref(p)[i] = S ∨ pref(p)[j] = S then
20                 for k := j to sjt(pref(p))[j] − 1 do
21                     B(pref(p))[k] := max(B(pref(p))[k], k − offset)
22                 end
23                 i := sjt(pref(p))[i]                 /* skip S */
24                 j := sjt(pref(p))[j]
25             else
26                 break                                /* mismatch */
27             end
28         end
29     end
30 end
```

Theorem 9. *Given a tree pattern p in* prefix notation *$pref(p)$ and tree pattern border array $\mathcal{B}(pref(p))$ constructed by Algorithm 2, the Algorithm 3 does not skip any occurrence of the pattern p in an input tree t.*

Proof. Assume that the match attempt found a mismatch on j-th symbol of the pattern, therefore the $shift$ is either by single position if $j = 1$, which is always safe, or, according to the tree pattern border array, $j - \mathcal{B}(pref(p))[j-1] - 1$ if $j \geq 1$.

Assume, that for the shift where $j \geq 1$ there is a shorter shift by k positions, where $0 < k < j - \mathcal{B}(pref(p))[j-1] - 1$. It must therefore be possible to match $pref(p)[k..j-1]$ with the pattern $pref(p)$ itself. However, the shift for mismatch at j-th position is derived from the $\mathcal{B}(pref(p))[j-1]$, which according to the Definition 7 failed to match $pref(p)[k..j-1]$ with the pattern $pref(p)$ itself by Definition 6 for each $0 < k < j - \mathcal{B}(pref(p))[j-1] - 1$. □

3.3 Example

Example 10. Consider a tree pattern p and a subject s with their respective representations in prefix notation $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$ and $pref(s) = a2\ a2$

Algorithm 3: Forward tree pattern matching algorithm

Input: The subject tree in $pref(s)$ notation of size n, the tree pattern in $pref(p)$ notation of size m, and a vector of integers $sjt(pref(p))$
Input: A vector of integers $\mathcal{B}(pref(p))$
Result: Locations of occurrences of the tree pattern p in the subject tree s.

```
1  begin
2      Spos := min({j : pref(p)[j] = S ∧ 1 ≤ j ≤ m})
3      shift[1] := 1
4      for i := 2 to m + 1 do shift[i] := i − B(pref(p))[i − 1] − 1
5      i := 0
6      j := 1
7      while i ≤ n − m do
8          offset := i + j
9          while j ≤ m and offset ≤ n do
10             if pref(p)[j] = pref(s)[offset] then
11                 j += 1
12                 offset += 1
13             else if pref(p)[j] = S then
14                 offset := sjt(pref(s))[offset]
15                 j += 1
16             else
17                 break
18             end
19         end
20         if j > m then yield i + 1
21         i += shift(pref(p))[j]
22         j := max(1, min(Spos, j) − shift(pref(p))[j])
23     end
24 end
```

id	1	2	3	4	5	6	7	8	9
pref(p)	$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$	
$\mathcal{B}(pref(p))$	0	1	2	2	3	4	5	6	
$shift(pref(p))$	1	1	1	1	2	2	2	2	2

Table 2. The tree pattern border array $\mathcal{B}(pref(p))$ and $shift(pref(p))$ used in Algorithm 3.

$a2$ $a0$ $a2$ $b1$ $b0$ $a0$ $a0$ $a2$ $a2$ $a0$ $a2$ $b1$ $b0$ $a0$ $a0$. Table 2 shows the tree pattern border array and derived shift function values for pattern p and Table 3 shows the run of the tree pattern matching algorithm. Matches are at indexes 2 and 10.

3.4 Time complexity

Consider a pattern p of length m and a subject s of length n. The time complexity of the preprocessing phase (construction of $sjt(pref(p))$ and Algorithm 2) is $O(m^2)$.

The classical Morris-Pratt algorithm runs in linear time with respect to the subject size thanks to the saving some subject to pattern symbol comparisons arising from the border array properties. Since the Forward linearised tree pattern matching algorithm skips some parts of the subject tree where wildcards are and the information about symbols inside the skipped subtree is not known while doing so, the number of symbols not needed to be matched in the next pattern to subject alignment is limited by the first subtree wildcard in the tree pattern. Matching itself therefore takes $O(m \cdot n + occ)$

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
pref(s)	$a2$	$a2$	$a2$	$a0$	$a2$	$b1$	$b0$	$a0$	$a0$	$a2$	$a2$	$a0$	$a2$	$b1$	$b0$	$a0$	$a0$
sjt	18	10	9	5	9	8	8	9	10	18	17	13	17	16	16	17	18
1	$a2$	$a2$	$\vdash$			S		$\dashv$	$a2$								
2		$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$								
3				$a2$													
4					$a2$	$a2$											
5						$a2$											
6							$a2$										
7								$a2$									
8									$a2$								
9										$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$

Table 3. Algorithm 3 run for the subject and the pattern from Example 10.

in general and $\Theta(n + occ)$ time when the pattern tree does not contain any wildcard. Time required for construction of $sjt(pref(s))$ is included.

4 Some empirical results

An existing Forest FIRE toolkit and accompanying FIRE Wood graphical user interface [7,19] were extended with the implementation of the presented algorithm. Many tree pattern matching algorithms based on automata, like those described in [2,6,8,12] and others, are already implemented within the toolkit. Single pattern matching algorithm based on linearisations of both pattern tree and subject tree utilising a backward shift heuristics [20] is also present. Performance of the presented algorithm is compared with some of the best-performing algorithms in the toolkit based on automata, according to the results in [8], and with the algorithm based on linearisation of tree structures utilising a backward shift heuristics. The running time of the pattern preprocessing was not measured as it is done only once for all queries by the pattern on many subjects.

We have measured the following running times of searching phases: 1) our new forward tree pattern matching algorithm based on linearisations of pattern and subject tree (FLTPM); 2) an algorithm based on linearisation of pattern and subject tree utilising a backward shift heuristics (BLTPM); 3) an algorithm based on the use of a *deterministic frontier-to-root (bottom-up) tree automaton* constructed for the pattern (DFRTA); and 4) an algorithm based on the use of a *Aho-Corasick automaton* constructed for the pattern's stringpath set (AC). The construction of a subtree jump table is included in the running time for both FLTPM and BLTMP algorithms and these algorithms' running times were recorded with trees represented in their prefix notation. Additionally, a modified version of the DFRTA algorithm reading prefix notation of a subject was implemented in the Forest FIRE toolkit as another reference algorithm (DFRTA Prefix). Since this algorithm doesn't need subtree jump table, its construction isn't included in the running time of DFRTA Prefix algorithm.

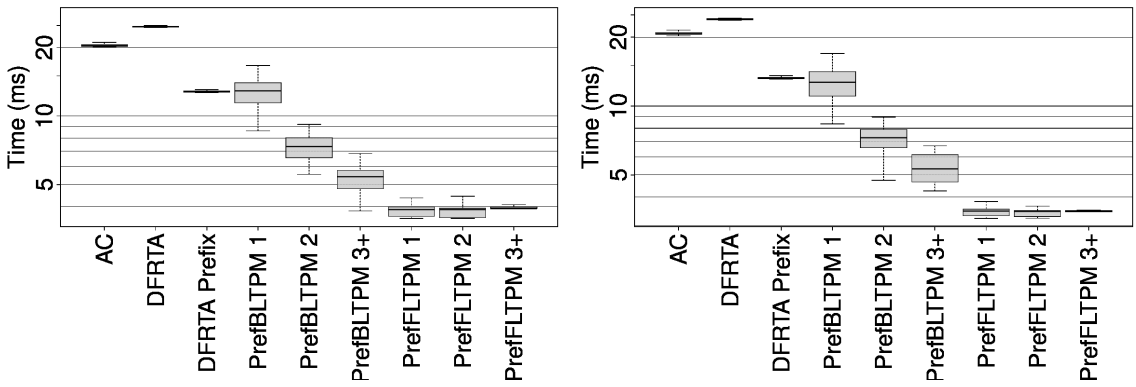
The performance of the new algorithm was measured using a pattern set previously used to measure the performance of preexistent algorithms in the Forest FIRE toolkit. This pattern set was obtained by taking the Mono project's X86 instruction set grammar and, for each grammar production, taking the tree in the production's right-hand side, and replacing any nonterminal occurrences by wildcard symbol occurrences. The resulting pattern set consists of 460 tree patterns of varying sizes.

Two sets of subject trees were used previously to measure the performance of Forest FIRE toolkit and the same two sets were used in the benchmarking of the new algorithm. The two subject sets were a set of 150 trees of approximately 500 nodes each and a set of 500 trees of approximately 150 nodes each.

As in the case of the BLTPM algorithm, the new algorithm is a single-pattern one. All chosen algorithms were executed with each pattern from the pattern set and each subject tree from two subject sets individually. The running times of the pattern matching algorithms were aggregated for a single tree pattern and all subject trees.

Benchmarking was conducted on a 2 GHz Intel Core i7 with 24 GB of RAM running OpenSUSE GNU/Linux version 15.2 using Java SE 11.

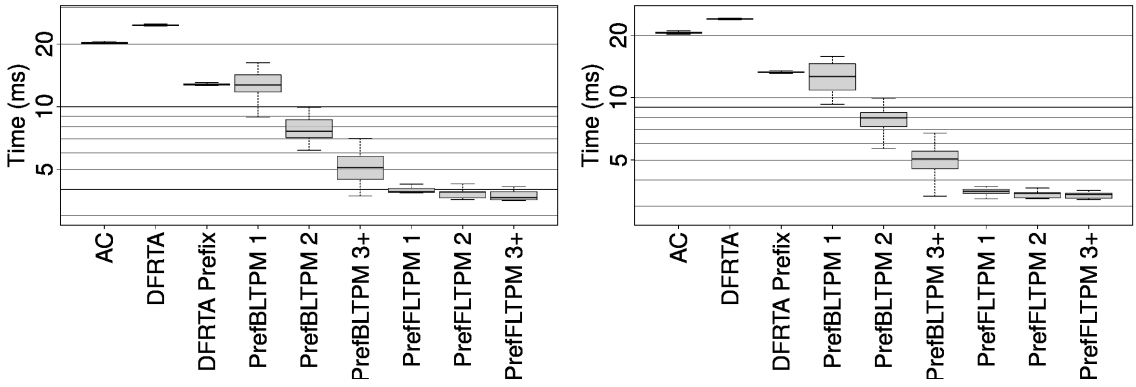
The linearised representations of subject trees and pattern trees were constructed in-memory and are linear in size with respect to the sizes of the subject trees and the pattern trees. The time required to construct the linear representation was not included in the running time of the searching phase. Also, because the search time was our primary concern, we do not consider memory use. Figure 3a and Figure 3b show the search times of tree patters with a wildcard symbol as boxplots. The figures of BLTPM and FLTPM algorithms were split to three based on the distance of the first wildcard symbol from the beginning of the pattern in its prefix representation to present this distance affects the running time. The distances are one symbol, two symbols, and three and more symbols. Similarly, Figure 4a and Figure 4b show the search times of tree patterns without a wildcard symbol as boxplots. The figures for BLTPM and FLTPM algorithms were split to three based on the length of the tree patterns, to patterns of length one, two, and three and more.



(a) Results on 150 trees of ca. 500 nodes each. (b) Results on 500 trees of ca. 150 nodes each.

Figure 3. Distributions of pattern matching times for the respective algorithms and patterns with wildcards.

The BLTPM algorithm generally runs faster for longer tree patterns without a wildcard or with a wildcard further from the beginning of the pattern whereas the FLTPM algorithm is unaffected by the wildcard position nor by the length of the pattern. The plots are clearly showing that on average, our new forward linearised tree pattern matching algorithm considerably outperforms the existing ones based on the automata approach for the single-pattern case (note the logarithmic scale) and even the backward linearised tree pattern matching algorithm.



(a) Results on 150 trees of ca. 500 nodes each. (b) Results on 500 trees of ca. 150 nodes each.

Figure 4. Distributions of pattern matching times for the respective algorithms and patterns without wildcards.

5 Concluding remarks

We presented a property of linearised trees similar to border arrays from strings. Using tree pattern border arrays, a new forward tree pattern matching algorithm similar to the Morris-Pratt algorithm for strings is defined. The algorithm was designed for trees represented in the prefix notation, but the idea can be adapted to other notations. The algorithm was empirically compared with other pattern matching algorithms and was shown to perform well in practice. Future work should focus on the identification of properties of the tree pattern border array to improve the preprocessing time and modification of the shift heuristics similar to the one used in the Knuth-Morris-Pratt algorithm. Future work should also include an investigation into a shift heuristics for multiple tree patterns, i.e., into a modification of the Aho-Corasick algorithm.

References

1. A. Aho and M. Corasick: *Efficient string matching: An aid to bibliographic search.* Commun. ACM, 18 06 1975, pp. 333–340.
2. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang: *Code generation using tree matching and dynamic programming.* ACM Trans. Program. Lang. Syst., 1989, pp. 491–516.
3. A. V. Aho and J. D. Ullman: *The theory of parsing, translation, and compiling*, Prentice-Hall, 1972.
4. R. Beal and D. Adjeroh: *Border array for structural strings*, in Combinatorial Algorithms, S. Arumugam and W. F. Smyth, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 189–205.
5. R. Beal and D. Adjeroh: *Border array for structural strings*, in Combinatorial Algorithms, S. Arumugam and W. F. Smyth, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 189–205.
6. D. R. Chase: *An improvement to bottom-up tree pattern matching*, in POPL, ACM Press, 1987, pp. 168–177.
7. L. Cleophas: *Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms*, in FSMNLP, J. Piskorski, B. W. Watson, and A. Yli-Jyrä, eds., vol. 19 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 191–198.
8. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Apr. 2008.
9. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: *Tree automata: Techniques and applications*, 2007, `http://www.grappa.univ-lille3.fr/tata/`.

10. T. Flouri, J. Janoušek, B. Melichar, C. S. Iliopoulos, and S. P. Pissis: *Tree template matching in ranked ordered trees by pushdown automata*, in Implementation and Application of Automata, B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, and D. Maurel, eds., vol. 6807 of Lecture Notes in Computer Science, Springer Verlag, 2011, pp. 273–281.
11. F. Gécseg and M. Steinby: *Tree Languages*, vol. 3 of Handbook of Formal Languages, Springer, 1997, pp. 1–68.
12. C. M. Hoffmann and M. J. O'Donnell: *Pattern matching in trees.* Journal of the ACM, 29(1) January 1982, pp. 68–95.
13. J. Janoušek: *Arbology: Algorithms on trees and pushdown automata*, PhD thesis, habilitation thesis, Brno University of Technology, 2010, submitted, 2010.
14. J. Janoušek, B. Melichar, R. Polách, M. Poliak, and J. Trávníček: *A full and linear index of a tree for tree patterns*, in Descriptional Complexity of Formal Systems, H. Jürgensen, J. Karhumäki, and A. Okhotin, eds., vol. 8614 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 198–209.
15. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM journal on computing, 6(2) 1977, pp. 323–350.
16. B. Melichar, J. Janoušek, and T. Flouri: *Arbology: trees and pushdown automata.* Kybernetika, 48, No.3 2012, pp. 402–428.
17. J. Morris Jr and V. Pratt: *A linear pattern-matching algorithm*, Technical Report 40, University of California, Berkeley, 1970.
18. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley-Pearson Education Limited, 2003.
19. R. Strolenberg: *ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms*, Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2007, `http://alexandria.tue.nl/extra1/afstversl/wsk-i/strolenberg2007.pdf`.
20. J. Trávníček, J. Janoušek, B. Melichar, and L. Cleophas: *On modification of boyer-moore-horspool's algorithm for tree pattern matching in linearised trees.* Theoretical Computer Science, 830-831 2020, pp. 60 – 90.