Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array

Jan Trávníček Robin Obůrka Tomáš Pecka Jan Janoušek

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
{Jan.Travnicek, oburkrob, Tomas.Pecka, Jan.Janousek}@fit.cvut.cz

PSC 2020
31. 8. 2020

Outline

Trees and Tree Notations

- An unranked alphabet
  $\mathcal{A} = \{a, b, \uparrow\}$
- Subject tree t_{1u} in the prefix bar notation

$pref\_bar(t_{1u}) = a\ a\ a \uparrow a\ a \uparrow\uparrow\uparrow a\ b \uparrow\uparrow\uparrow$

- A ranked alphabet
  $\mathcal{A} = \{a2, a1, a0, b0\}$
- And t_{1r} in the prefix notation

$pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ b0$

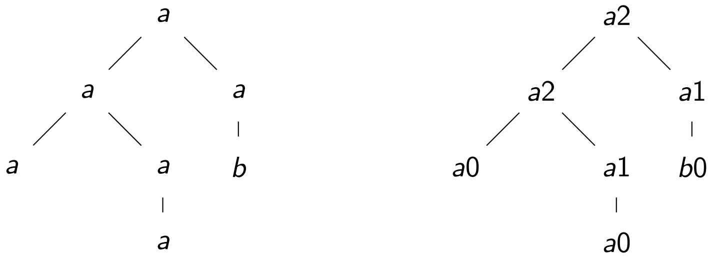


Figure: Subject tree t_{1u} over an unranked alphabet (left), and the same subject tree t_{1r} over a ranked alphabet (right)

Some Other Notations

On a ranked alphabet

- $\mathcal{A} = \{a2, a1, a0, b0\}$, t_{1r} in the postfix notation
  $post(t_{1r}) = a0\ a0\ a1\ a2\ b0\ a1\ a2$
- $\mathcal{A} = \{a2, a1, a0, \uparrow 2, \uparrow 1, \uparrow 0\}$, t_{1r} in the prefix ranked bar notation
  $pref\_ranked\_bar(t_{1r}) = a2\ a2\ a0 \uparrow 0\ a1\ a0 \uparrow 0 \uparrow 1 \uparrow 2\ a1\ b0 \uparrow 0 \uparrow 1 \uparrow 2$
- Euler tour traversal, ...

On an unranked alphabet

- $\mathcal{A} = \{a, (,)\}$, t_{1r} in the prefix bracketed notation
  $pref\_brac(t_{1u}) = a\ (\ a\ (\ a\ (\ )\ a\ (\ a\ (\ )\ )\ )\ a\ (\ b\ (\ )\ )\ )$
- $\mathcal{A} = \{a, \uparrow\}$, t_{1r} in the postfix bar notation
  $post\_bar(t_{1u}) = \uparrow\uparrow\uparrow a \uparrow\uparrow a\ a\ a \uparrow\uparrow b\ a\ a$

Trees Patterns

- A ranked alphabet of a tree template $\mathcal{A} = \{a2, a1, a0, S\}$
- Tree template p_{2r} in the prefix notation $pref(t_{1r}) = a2\ S\ a1\ S$
- Symbol S stands for any subtree.

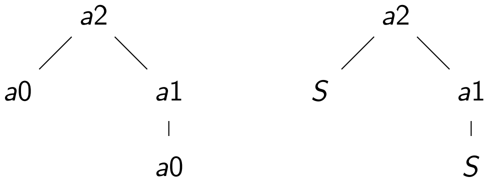


Figure: Subtree p_{1r} (left) of the t_{1r}, tree template p_{2r} (right)

Tree Pattern Matching

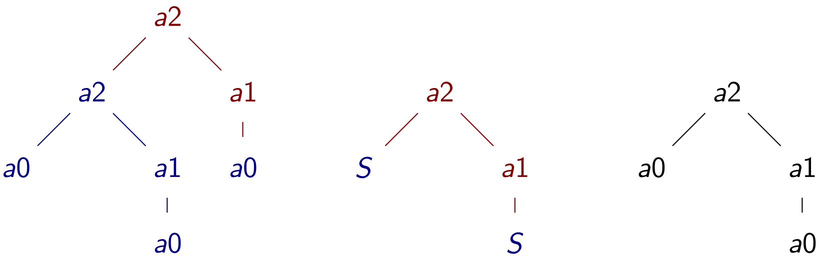


Figure: Tree t_{1r} over a ranked alphabet (left), tree template p_{2r} (centre) and subtree p_{1r} of t_{1r} (right)

- For many linearisations it holds that subtree s of a tree t, the linear respresenation of s is a substring of linear representation of t.

$pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$
$pref(p_{1r}) = a2\ a0\ a1\ a0$
$pref(p_{2r}) = a2\ S\ a1\ S$

Tree Pattern Matching

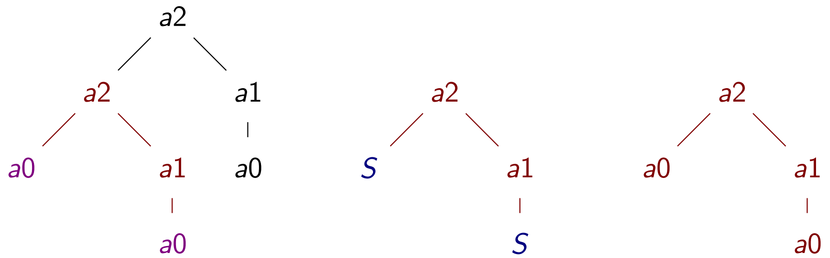


Figure: Tree t_{1r} over a ranked alphabet (left), tree template p_{2r} (centre) and subtree p_{1r} of t_{1r} (right)

- For many linearisations it holds that subtree s of a tree t, the linear respresenation of s is a substring of linear representation of t.

$pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$
$pref(p_{1r}) = a2\ a0\ a1\ a0$
$pref(p_{2r}) = a2\ S\ a1\ S$

Subtree Jump Table

A structure allowing quick jumps over subtrees of a given linearised tree.

Definition (subtree jump table for prefix notation $sjt(pref(t))$)

Let t and $pref(t) = \ell_1\ell_2 \dots \ell_n$, $n \geq 1$, 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 $\ell_i\ell_{i+1} \dots \ell_{j-1}$ is the prefix notation of a subtree of tree t, then $sjt(pref(t))[i] = j$, $1 \leq i < j \leq n+1$.

Table: Subtree jump table $sjt(pref(t_{1r}))$

id	1	2	3	4	5	6	7
$pref(t_{1r})$	$a2$	$a2$	$a0$	$a1$	$a0$	$a1$	$b0$
$sjt(pref(t_{1r}))$	8	6	4	6	6	8	8

Forward Pattern Matching

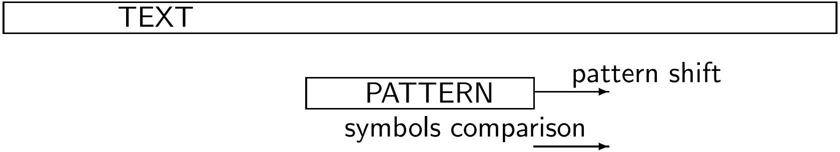


Figure: Graphical outline of a forward pattern matching algorithm

Searching for Subtrees

- A Morris-Pratt algorithm makes use of a border array table.
- A border of a string s a prefix of s that is also a suffix of s.
- The border array stores the length of the longest border for each prefix of s.

An alphabet $\mathcal{A} = \{a2, a0\}$
A string $s_1 = a2\, a0\, a2\, a0\, a0$

Table: The border array for string s_1

1	2	3	4	5
0	0	1	2	0

In order to look for subtrees in a tree, the Morris-Pratt algorithm can be used without changes. Actually, any string pattern matching algorithm can be used.

String Morris-Pratt algorithm

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.

```
begin
    i := 0, j := 1
    while i ≤ n − m do
        /* occurrence check loop */
        while j ≤ m and s[i + j] = p[j] do
            j += 1
        end
        if j > m then yield i + 1
        /* shift handling */
        if j ≠ 1 then
            i += j − B(p)[j − 1] − 1 /* j - 1st symbol failed or overflowed */
            j := B(p)[j − 1] + 1
        else
            i += 1
        end
    end
end
```

Alternative border-array definition

Definition (border array $\mathcal{B}(s)$)

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
$\mathcal{B}(s)[i] = max(\{0\} \cup \{k : s[1..k] \;=\; s[i-k+1..i] \wedge k \geq 1 \wedge i-k+1 > 1\})$.

Forward Tree Pattern Matching Algorithm

Modifications to the Morris-Pratt algorithm needed to use it for tree patterns.

- The occurrence check loop has to be modified to handle the wildcards,
- the border array needs to be modified to represent the same idea in tree patterns.

Matches

Definition (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:

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

Tree pattern border array

Definition (tree pattern border array $\mathcal{B}(pref(p))$)

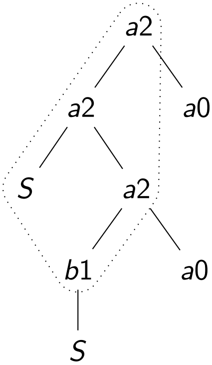
Let $pref(p)$ be a tree pattern in a prefix notation of length n. The $\mathcal{B}(pref(p))$ is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}(pref(p))[1] = 0$ and otherwise $\mathcal{B}(pref(p))[i] = max(\{0\} \cup \{k : pref(p)\ matches\ pref(p)[i-k+1..i] \wedge k \geq 1 \wedge i-k+1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

Visualization of the Matches Relation

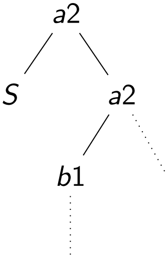
Table: Trace of naive computation of $pref(p)$ $matches$ $pref(p)[j+1..5]$ for $1 \leq j \leq 5$ and $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$.

	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 (pref(p)[6..5] $= \varepsilon$)

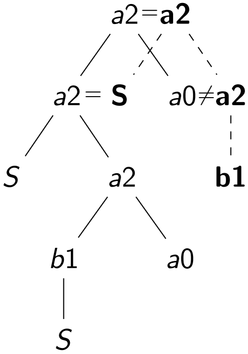
Bad Character Shift Table for Tree Templates



(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.

The border array example

Table: The tree pattern border array $\mathcal{B}(pref(p))$ for $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$.

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

Modification of the Occurrence Check loop

Due to the variable length of the subtree matched to the wildcard

- an offset to the subject is to be maintained,
- the subtree jump table is used to efficiently skip over subtrees.

Algorithm 2: (Fragment) modification of the Occurrence check.

```
offset := i + j
while j ≤ m and offset ≤ n do
    if pref(p)[j] = pref(s)[offset] then
        j += 1
        offset += 1
    else if pref(p)[j] = S then
        offset := sjt(pref(s))[offset]
        j += 1
    else
        break
    end
end
```

Modification of the Shift Handling

Modifications to the shift handling:

- the shift handling is mainly updated by switching to tree pattern border array
- the number of character that do not need to be matched is limited by the distance of the wildcard from the beginning of the pattern.

Algorithm 3: (Fragment) modification of the Shift Handling.

0a $Spos := min(\{k : pref(p)[k] = S \wedge 1 \leq k \leq m\})$
0b $shift[1] := 1$
0c **for** $k := 2$ **to** $m + 1$ **do** $shift[k] := k - \mathcal{B}(pref(p))[k-1] - 1$
/* because the k - 1st symbol failed or overflowed */

$\vdots$

10-15a $i \mathrel{+}= shift(pref(p))[j]$
10-15b $j := max(1, min(Spos, j) - shift(pref(p))[j])$

Sample run of the matching algorithm

Table: The $shift(pref(p))$ for $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$.

id	1	2	3	4	5	6	7	8	9
pref(p)	*a2*	*a2*	*S*	*a2*	*b1*	*S*	*a0*	*a0*	
shift(pref(p))	1	1	1	1	2	2	2	2	2

Table: Run for the subject $pref(s)$ and the pattern $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$.

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*

Complexities

The n is the size of the subject tree, the m is the size of the tree template and the $\mathcal{A}$ is the size of the alphabet.

- The tree pattern border array requires $\Theta(m)$ space.
- The preprocessing (computation of the tree pattern border array) takes $\mathcal{O}(m^2)$ time
- The algorithm runs in $\Omega(n)$ time in the best case and $\mathcal{O}(m \cdot n)$ time in the worst case if searching for tree templates and $\Theta(n)$ time if searching for subtrees.

Measurements

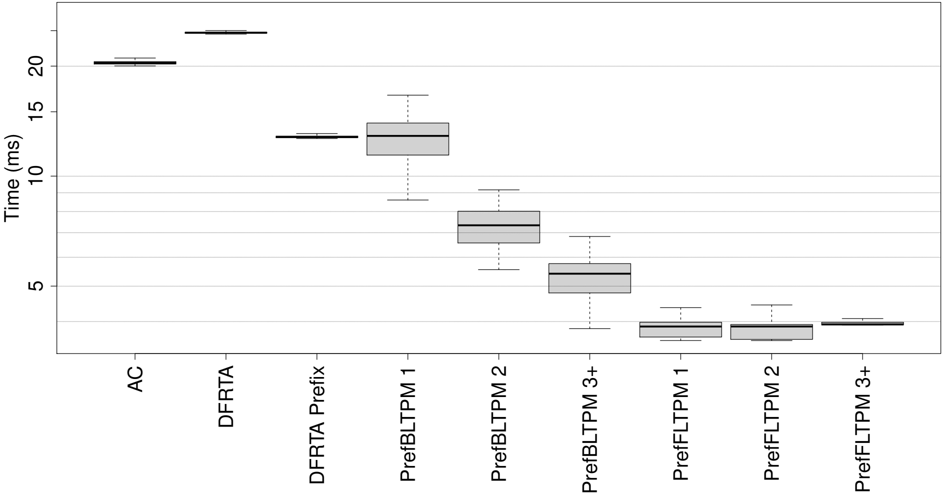


Figure: Results on 150 trees of ca. 500 nodes each with wildcards.

Measurements

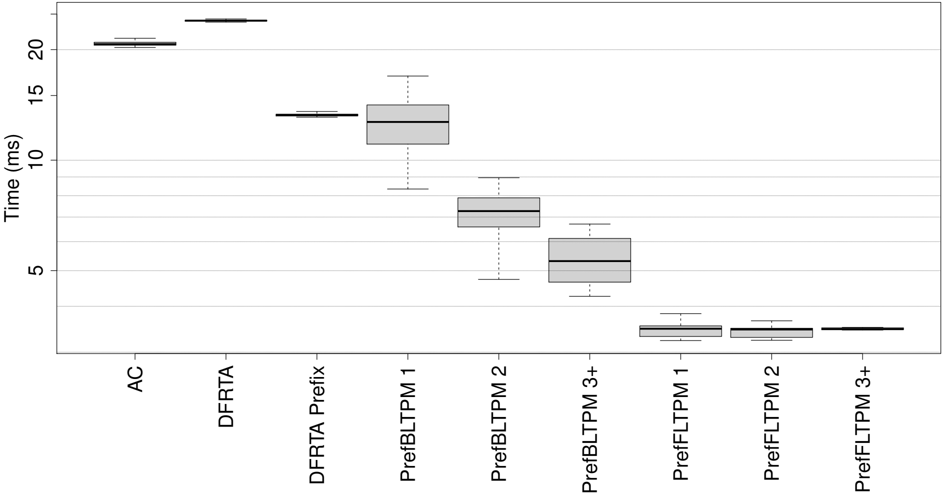


Figure: Results on 500 trees of ca. 150 nodes each with wildcards.

Measurements

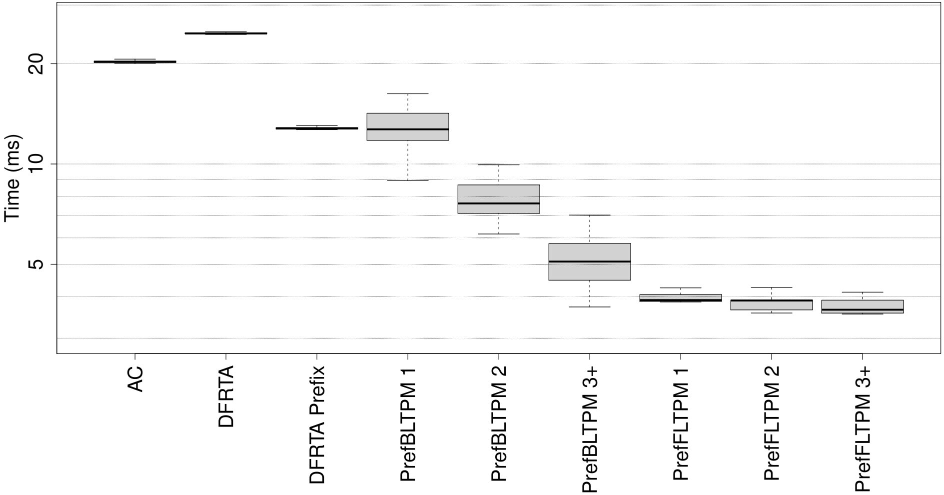


Figure: Results on 150 trees of ca. 500 nodes each without wildcards.

Measurements

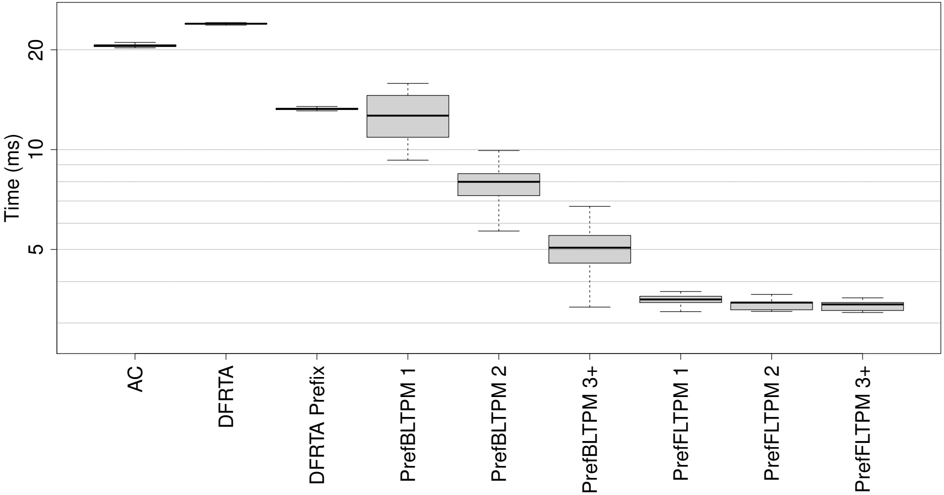


Figure: Results on 500 trees of ca. 150 nodes each without wildcards.

Conclusions

Results:

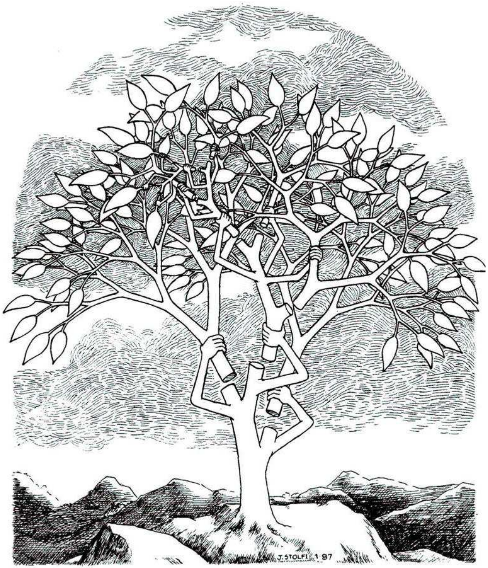
- A new tree pattern matching algorithm was presented,
- the algorithm is based on Morris-Pratt algorithm and uses an adaptation of the border array from string domain,
- the algorithm was implemented in the Forrest-FIRE toolkit and experimentally evaluated using the dataset used in its original evaluation.

Takeaway message:

- Many string processing algorithms can be modified (with some care) to process trees represented as strings using some linearisation schemes.

Future work:

- Adapt the Knuth-Morris-Pratt improvement to the presented algorithm.


J. STOLFI 1·87