Introduction to Unification Theory

Matching

Temur Kutsia

RISC, Johannes Kepler University of Linz, Austria
kutsia@risc.jku.at

Overview

Syntactic Matching

Advanced Topics

Overview

Syntactic Matching

Advanced Topics

Matching Problem

- Given: terms t and s.
- Find: a substitution σ such that $t\sigma = s$ (syntactic matching).
- Matching equation: $t \leq^? s$.
- σ is called a matcher.

Matching Problem

Example

- Matching problem: $f(x, y) \leq^? f(g(z), x)$.
  Matcher: $\sigma = \{x \mapsto g(z), y \mapsto x\}$.

Matching Problem

Example

- Matching problem: $f(x, y) \leq^? f(g(z), x)$.
  Matcher: $\sigma = \{x \mapsto g(z), y \mapsto x\}$.
- Matching problem: $f(x, x) \leq^? f(x, a)$.
  No matcher.

Matching Problem

Example

- Matching problem: $f(x, y) \leq^? f(g(z), x)$.
  Matcher: $\sigma = \{x \mapsto g(z), y \mapsto x\}$.
- Matching problem: $f(x, x) \leq^? f(x, a)$.
  No matcher.
- Matching problem: $f(g(x), x, y) \leq^? f(g(g(a)), g(a), b)$.
  Matcher: $\{x \mapsto g(a), y \mapsto b\}$.

Matching Problem

Example

- Matching problem: $f(x, y) \leq\!\cdot^{?} f(g(z), x)$.
  Matcher: $\sigma = \{x \mapsto g(z), y \mapsto x\}$.
- Matching problem: $f(x, x) \leq\!\cdot^{?} f(x, a)$.
  No matcher.
- Matching problem: $f(g(x), x, y) \leq\!\cdot^{?} f(g(g(a)), g(a), b)$.
  Matcher: $\{x \mapsto g(a), y \mapsto b\}$.
- Matching problem: $f(x) \leq\!\cdot^{?} f(g(x))$.
  Matcher: $\{x \mapsto g(x)\}$.

Relating Matching and Unification

- Matching can be reduced to unification.
- Simply replace in a matching problem $t \leq^? s$ each variable in s with a new constant.
- $f(x, y) \leq^? f(g(z), x)$ becomes the unification problem $f(x, y) \doteq^? f(g(c_z), c_x)$.
- c_z, c_x: new constants.
- The unifier: $\{x \mapsto g(c_z), y \mapsto c_x\}$.
- The matcher: $\{x \mapsto g(z), y \mapsto z\}$.
- When t is ground, matching and unification coincide.

Relating Matching and Unification

- Both matching and unification can be implemented in linear time.
- Linear implementation of matching is straightforward.
- Linear implementation of unification requires sophisticated data structures.
- Whenever efficiency is an issue, matching should be implemented separately from unification.

Overview

Syntactic Matching

Advanced Topics

Tree Pattern Matching

- Matching is needed in rewriting, functional programming, querying, etc.
- Often the following problem is required to be solved:
  - Given a ground term s (subject) and a term p (pattern)
  - Find all subterms in s to which p matches.
- Notation: $p \ll^? s$.
- In this lecture: An algorithm to solve this problem.
- Terms are represented as trees.

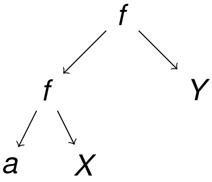
Matching

Working example:

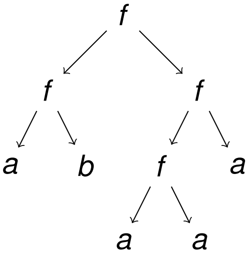
$$f(f(a, X), Y) \ll^{?} f(f(a, b), f(f(a, b), a)).$$

Tree Pattern Matching

Matching the pattern tree to the subject tree.



Pattern tree 1

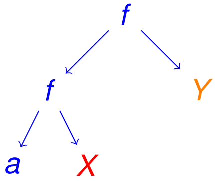


Subject tree

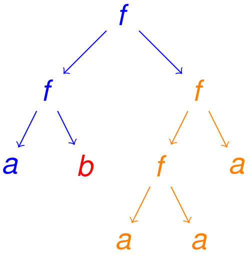
Tree Pattern Matching

Matching the pattern tree to the subject tree.

Pattern tree 1. First match:



Pattern tree 1

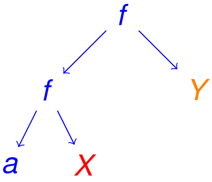


Subject tree

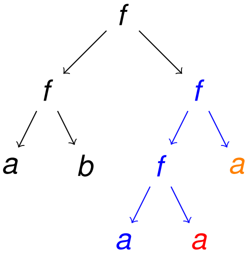
Tree Pattern Matching

Matching the pattern tree to the subject tree.

Pattern tree 1. Second match:



Pattern tree 1

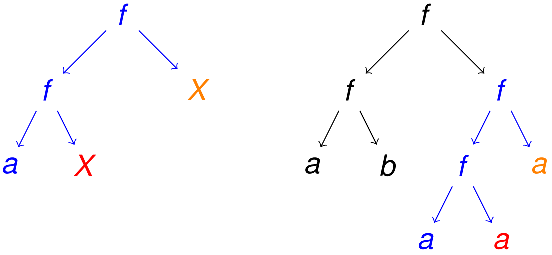


Subject tree

Tree Pattern Matching

Matching the pattern tree to the subject tree.

Pattern tree 2. Single match:



Pattern tree 2

Subject tree

Tree Pattern Matching

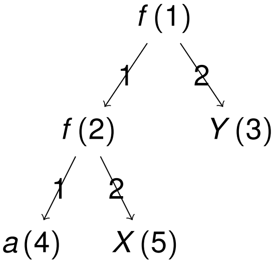
- Pattern tree 1 in the example is linear: Every variable occurs only once.
- Pattern tree 2 is nonlinear: X occurs twice.
- Two steps for nonlinear tree matching:
  1. Ignore multiplicity of variables (assume the pattern in linear) and do linear tree pattern matching.
  2. Verify that the substitutions computed for multiple occurrences of a variable are identical: check consistency.

Terms

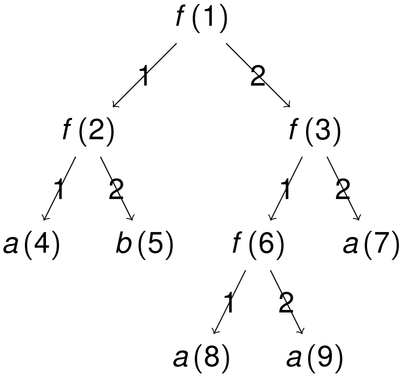
- $\mathcal{V}$: Set of variables.
- $\mathcal{F}$: Set of function symbols of fixed arity.
- $\mathcal{F} \cap \mathcal{V} = \emptyset$.
- Constants: 0-ary function symbols.
- Terms:
  - A variable or a constant is a term.
  - If $f \in \mathcal{F}$, f is n-ary, $n > 0$, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

Term Trees, Nodes, Node Labels, Edges, Edge labels

Example



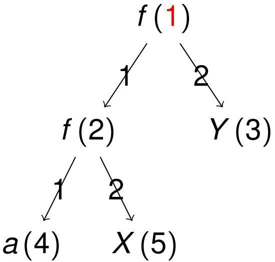
The tree for $f(f(a, X), Y)$

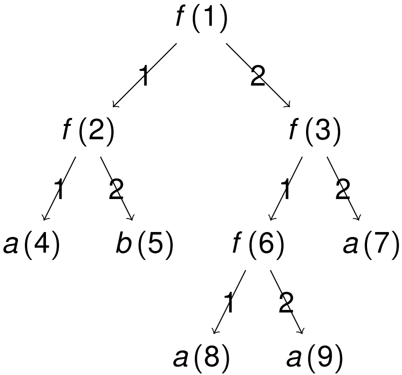


The tree for $f(f(a, b), f(f(a, a), a))$

Term Trees, Nodes, Node Labels, Edges, Edge labels

Example

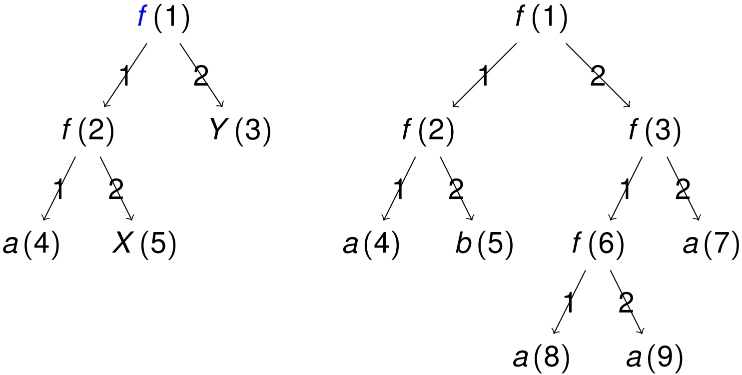




Node

Term Trees, Nodes, Node Labels, Edges, Edge labels

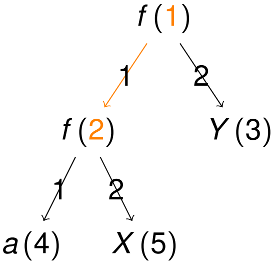
Example

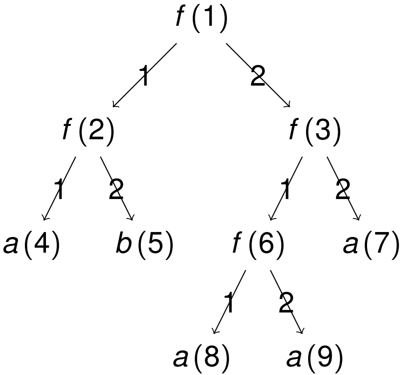


Node label

Term Trees, Nodes, Node Labels, Edges, Edge labels

Example

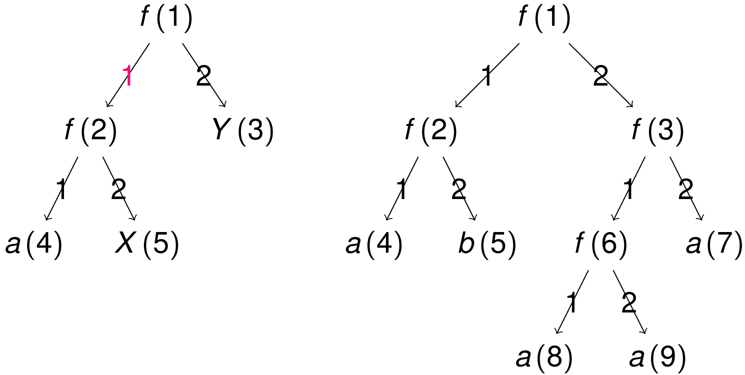




Edge

Term Trees, Nodes, Node Labels, Edges, Edge labels

Example



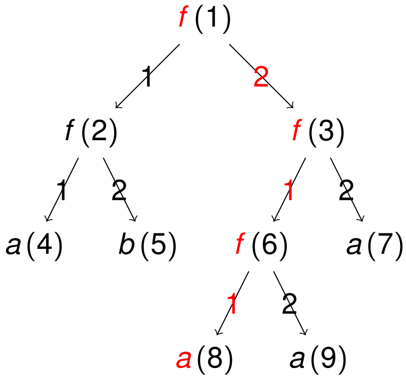
Edge label

Labeled Path

- Labeled path $lp(n_1, n_q)$ in a term tree from the node n_1 to the node n_q:
  A string formed by alternatively concatenating the node and edge labels from n_1 to n_q.

Labeled Path

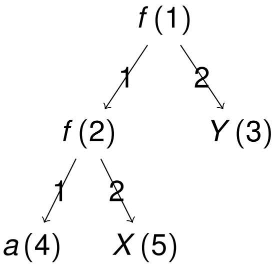
Example



Labeled path from 1 to 8: $lp(1, 8) = f2f1f1a$

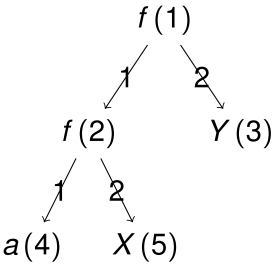
Euler Chains and Strings

- Euler chain for a term tree: a string of node labels obtained as follows:



Euler Chains and Strings

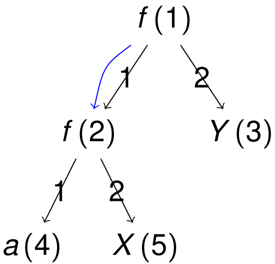
- Euler chain for a term tree: a string of node labels obtained as follows:



1

Euler Chains and Strings

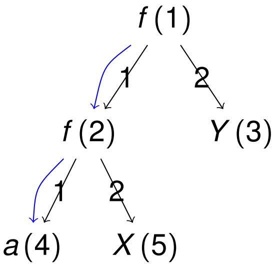
- Euler chain for a term tree: a string of node labels obtained as follows:



12

Euler Chains and Strings

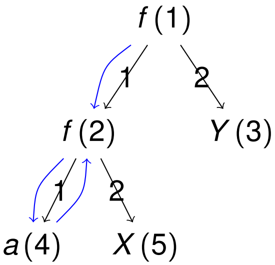
- Euler chain for a term tree: a string of node labels obtained as follows:



124

Euler Chains and Strings

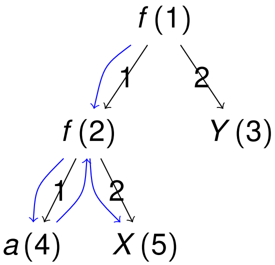
- Euler chain for a term tree: a string of node labels obtained as follows:



1242

Euler Chains and Strings

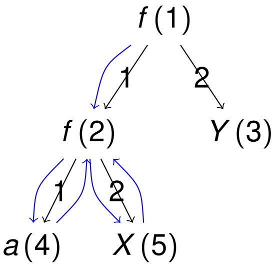
- Euler chain for a term tree: a string of node labels obtained as follows:



12425

Euler Chains and Strings

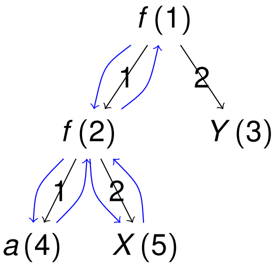
- Euler chain for a term tree: a string of node labels obtained as follows:



124252

Euler Chains and Strings

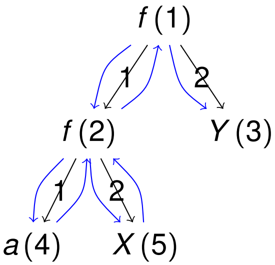
- Euler chain for a term tree: a string of node labels obtained as follows:



1242521

Euler Chains and Strings

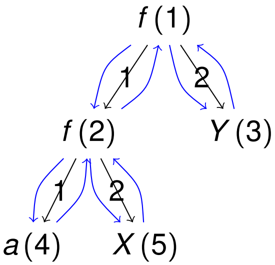
- Euler chain for a term tree: a string of node labels obtained as follows:



12425213

Euler Chains and Strings

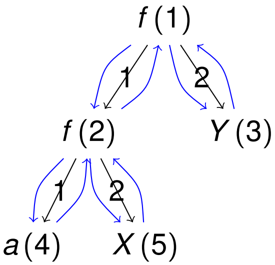
- Euler chain for a term tree: a string of node labels obtained as follows:



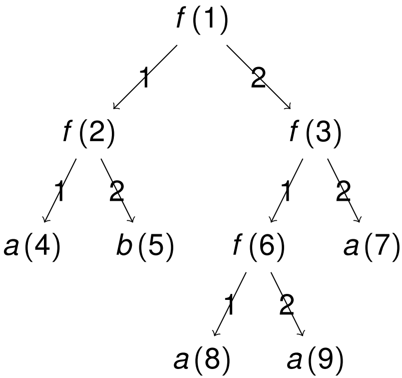
124252131

Euler Chains and Strings

- Euler chain for a term tree: a string of node labels obtained as follows:



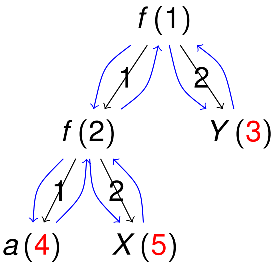
124252131

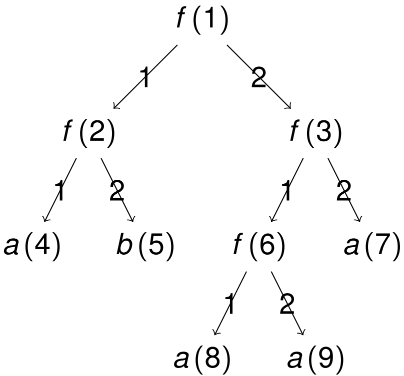


124252136869637 31

Euler Chains and Strings

- Properties of Euler chains





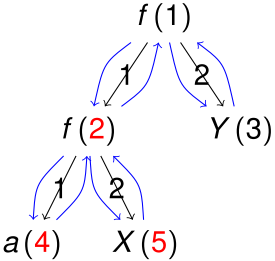
The leaves occur only once:

124252131

1242521368696373 1

Euler Chains and Strings

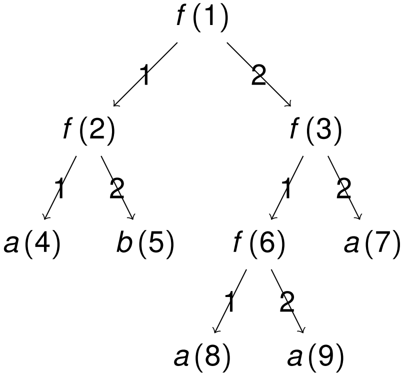
- Properties of Euler chains



The subchain between the first and last occurrence of a node:

The chain of the subtree rooted at that node:

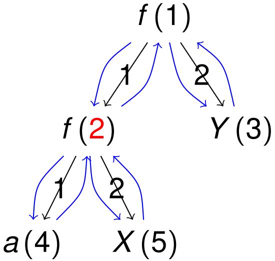
124252131

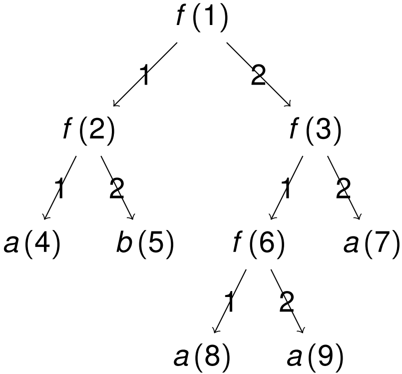


12425213686963731

Euler Chains and Strings

- Properties of Euler chains





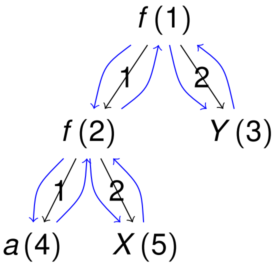
A node with n children occurs $n + 1$ times

124252131

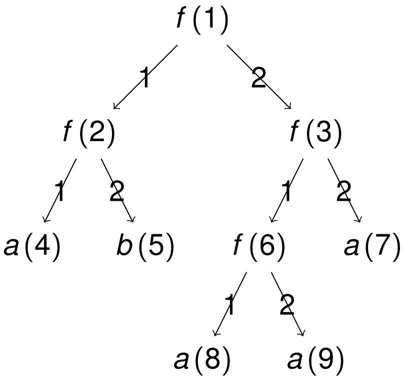
12425213686963731

Euler Chains and Strings

- Euler strings: Replace nodes in Euler chains with node labels.



ffafXffYf



ffafbffffafaffaff

Tree Pattern Matching: Idea

- Instead of using the tree structure, the algorithm operates on Euler chains and Euler strings.

Tree Pattern Matching: Idea

- Instead of using the tree structure, the algorithm operates on Euler chains and Euler strings.
- To declare a match of the pattern tree at a subtree of the subject tree, the algorithm
  - verifies whether their Euler strings are identical after replacing the variables in the pattern by Euler strings of appropriate terms.

Tree Pattern Matching: Idea

- Instead of using the tree structure, the algorithm operates on Euler chains and Euler strings.
- To declare a match of the pattern tree at a subtree of the subject tree, the algorithm
  - verifies whether their Euler strings are identical after replacing the variables in the pattern by Euler strings of appropriate terms.
- To justify this approach, Euler strings have to be related to the tree structures.

Tree Pattern Matching: Idea

- Instead of using the tree structure, the algorithm operates on Euler chains and Euler strings.
- To declare a match of the pattern tree at a subtree of the subject tree, the algorithm
  - verifies whether their Euler strings are identical after replacing the variables in the pattern by Euler strings of appropriate terms.
- To justify this approach, Euler strings have to be related to the tree structures.

Theorem

Two term trees are equivalent (i.e. they represent the same term) iff their corresponding Euler strings are identical.

Nonlinear Tree Pattern Matching: Ideas

Putting the ideas together:

1. Ignore multiplicity of variables (assume the pattern is linear) and do linear tree pattern matching.
2. Verify that the substitutions computed for multiple occurrences of a variable are identical: check consistency.
3. Instead of trees, operate on their Euler strings.

Notation

- s: Subject tree.
- p: Pattern tree.
- C_s and E_s: Euler chain and Euler string for the subject tree.
- C_p and E_p: Euler chain and Euler string for the pattern tree.
- n: Size of s.
- m: Size of p.
- k: Number of variables in p.
- K: The set of all root-to-variable-leaf pathes in p.

Step 1. Linear Tree Pattern Matching

- Let $v_1, \ldots, v_k$ be the variables in p.
- $v_1, \ldots, v_k$ appear only once in E_p, because
  - only leaves are labeled with variables,
  - each leaf appears exactly once in the Euler string, and
  - each variable occurs exactly once in p (linearity).

Step 1. Linear Tree Pattern Matching

We start with a simple algorithm.

- E_s is stored in an array.

Step 1. Linear Tree Pattern Matching

We start with a simple algorithm.

- E_s is stored in an array.
- Split E_p into $k+1$ strings, denoted $\sigma_1, \ldots, \sigma_{k+1}$, by removing variables.

Step 1. Linear Tree Pattern Matching

We start with a simple algorithm.

- E_s is stored in an array.
- Split E_p into $k+1$ strings, denoted $\sigma_1, \ldots, \sigma_{k+1}$, by removing variables.
  - $ffafXffYf$ splits into $\sigma_1 = ffaf$, $\sigma_2 = ff$, and $\sigma_3 = f$.

Step 1. Linear Tree Pattern Matching

We start with a simple algorithm.

- E_s is stored in an array.
- Split E_p into $k+1$ strings, denoted $\sigma_1, \ldots, \sigma_{k+1}$, by removing variables.
  - *ffafXffYf* splits into $\sigma_1 = \mathit{ffaf}$, $\sigma_2 = \mathit{ff}$, and $\sigma_3 = f$.
- Construct Boolean tables $M_1, \ldots, M_k$, each having $|E_s|$ entries:

$$M_i[j] = \begin{cases} 1 & \text{if there is a match for } \sigma_i \text{ in } E_s \text{ starting at pos. } j \\ 0 & \text{otherwise.} \end{cases}$$

Step 1. Linear Tree Pattern Matching

Example

- $E_p = \texttt{ffafXffYf}$, $\sigma_1 = \texttt{ffaf}$, $\sigma_2 = \texttt{ff}$, $\sigma_3 = \texttt{f}$,
  $E_s = \texttt{ffafbffffafaffaff}$.
- $M_1 = \texttt{10000001000010000}$ ($\texttt{ffafbffffafaffaff}$).
- $M_2 = \texttt{10000111000010010}$ ($\texttt{ffafbffffafaffaff}$).
- $M_3 = \texttt{11010111100011011}$ ($\texttt{ffafbffffafaffaff}$).

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rl rl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- We start from $M_1 = \mathtt{10000001000010000}$.

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{12425213686963731} \\
E_p = & \texttt{ffafXffYf} & E_s = & \texttt{ffafbffffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000111000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}
$$

- We start from $M_1 = \texttt{10000001000010000}$.
- The set of nodes where p matches s is a subset of the set of nodes with nonzero entries in M_1.

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- We start from $M_1 = \mathtt{10000001000010000}$.
- The set of nodes where p matches s is a subset of the set of nodes with nonzero entries in M_1.
- Take a nonzero entry position i in M_1 that corresponds to the first occurrence of a node in the Euler chain, $i = 1$.

Step 1. Linear Tree Pattern Matching

$$\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{12425213686963731} \\
E_p = & \texttt{ffafXffYf} & E_s = & \texttt{ffafbffffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000111000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}$$

- The replacement for X must be a string in E_s that starts at position $i + |\sigma_1| = 5$

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rl rl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{12425213686963731} \\
E_p = & \texttt{ffafXffYf} & E_s = & \texttt{ffafbffffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000111000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}
$$

- The replacement for X must be a string in E_s that starts at position $i + |\sigma_1| = 5$
- Moreover, this position must correspond to the first occurrence of a node in the Euler chain, because
  - variables can be substituted by subtrees only,
  - a subtree starts with the first occurrence of a node in the Euler chain.

If this is not the case, take another nonzero entry position in M_1.

Step 1. Linear Tree Pattern Matching

$p =$	f(f(a, X), Y)	$s =$	f(f(a, b), f(f(a, a), a))
$C_p =$	124252131	$C_s =$	12425213686963731
$E_p =$	ffafXffYf	$E_s =$	ffafbffffafaffaff
$\sigma_1 =$	ffaf	$M_1 =$	10000001000010000
$\sigma_2 =$	ff	$M_2 =$	10000111000010010
$\sigma_3 =$	f	$M_3 =$	11010111100011011

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.

Step 1. Linear Tree Pattern Matching

$p =$	`f(f(a,X),Y)`	$s =$	`f(f(a,b),f(f(a,a),a))`
$C_p =$	`124252131`	$C_s =$	`12425213686963731`
$E_p =$	`ffafXffYf`	$E_s =$	`ffafbffffafaffaff`
$\sigma_1 =$	`ffaf`	$M_1 =$	`10000001000010000`
$\sigma_2 =$	`ff`	$M_2 =$	`10000111000010010`
$\sigma_3 =$	`f`	$M_3 =$	`11010111100011011`

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.
- Let j be the position of the last occurrence from the previous item. Then $M_2[j+1]$ should be 1: σ_2 should match E_s at this position.

Step 1. Linear Tree Pattern Matching

$$\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{1242521368696 3731} \\
E_p = & \texttt{ffaf}\color{red}{\texttt{X}}\texttt{ffYf} & E_s = & \texttt{ffaf}\color{red}{\texttt{b}}\texttt{fffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{1000000100001 0000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000}\color{blue}{\texttt{1}}\texttt{1100001 0010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{1101011110001 1011}
\end{array}$$

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.
- Let j be the position of the last occurrence from the previous item. Then $M_2[j+1]$ should be 1: σ_2 should match E_s at this position.
- And proceed in the same way...

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{12425213686963731} \\
E_p = & \texttt{ffafXffYf} & E_s = & \texttt{ffafbffffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000111000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}
$$

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.
- Let j be the position of the last occurrence from the previous item. Then $M_2[j+1]$ should be 1: σ_2 should match E_s at this position.
- And proceed in the same way...

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{1242521}{\color{blue}\texttt{3}}\texttt{686963731} \\
E_p = & \texttt{ffaf}{\color{red}\texttt{X}}\texttt{ffYf} & E_s = & \texttt{ffaf}{\color{red}\texttt{b}}\texttt{ff}{\color{blue}\texttt{f}}\texttt{fafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000}{\color{blue}\texttt{1}}\texttt{11000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}
$$

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.
- Let j be the position of the last occurrence from the previous item. Then $M_2[j+1]$ should be 1: σ_2 should match E_s at this position.
- And proceed in the same way...

Step 1. Linear Tree Pattern Matching

$p =$	f(f(a, X), Y)	$s =$	f(f(a, b), f(f(a, a), a))
$C_p =$	124252131	$C_s =$	124252136869637 31
$E_p =$	ffafXffYf	$E_s =$	ffafbfffafaffaff
$\sigma_1 =$	ffaf	$M_1 =$	1000000100001000 0
$\sigma_2 =$	ff	$M_2 =$	10000111000010010
$\sigma_3 =$	f	$M_3 =$	11010111100011011

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.
- Let j be the position of the last occurrence from the previous item. Then $M_2[j + 1]$ should be 1: σ_2 should match E_s at this position.
- And proceed in the same way...

Step 1. Linear Tree Pattern Matching

$p =$	`f(f(a,X),Y)`	$s =$	`f(f(a,b),f(f(a,a),a))`
$C_p =$	`124252131`	$C_s =$	`12425213686963731`
$E_p =$	`ffafXffYf`	$E_s =$	`ffafbffffafaffaff`
$\sigma_1 =$	`ffaf`	$M_1 =$	`10000001000010000`
$\sigma_2 =$	`ff`	$M_2 =$	`10000111000010010`
$\sigma_3 =$	`f`	$M_3 =$	`11010111100011011`

- Replacement for X is a substring of E_s between the first and last occurrences of the node at position $i + |\sigma_1|$.
- Let j be the position of the last occurrence from the previous item. Then $M_2[j+1]$ should be 1: σ_2 should match E_s at this position.
- And proceed in the same way...

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{12425213686963731} \\
E_p = & \texttt{ffafXffYf} & E_s = & \texttt{ffafbffffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000111000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}
$$

- The first match found: $\texttt{X} \to \texttt{b}, \texttt{Y} \to \texttt{f(f(a,a),a)}$.

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{1242521368696373 1} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rl@{\qquad}rl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{124252136869 63731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{124252136869}\mathtt{63731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rl rl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \texttt{f(f(a,X),Y)} & s = & \texttt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \texttt{124252131} & C_s = & \texttt{12425213686963731} \\
E_p = & \texttt{ffafXffYf} & E_s = & \texttt{ffafbffffafaffaff} \\
\sigma_1 = & \texttt{ffaf} & M_1 = & \texttt{10000001000010000} \\
\sigma_2 = & \texttt{ff} & M_2 = & \texttt{10000111000010010} \\
\sigma_3 = & \texttt{f} & M_3 = & \texttt{11010111100011011}
\end{array}
$$

- The first match found: $\texttt{X} \to \texttt{b}, \texttt{Y} \to \texttt{f(f(a,a),a)}$.
- The next attempt

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt gives $\mathtt{X} \to \mathtt{a}, \mathtt{Y} \to \mathtt{a}$.

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{12425213686963731} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt gives $\mathtt{X} \to \mathtt{a}, \mathtt{Y} \to \mathtt{a}$.
- One more try...

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{124252136869637 31} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaff} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{10000001000010000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt gives $\mathtt{X} \to \mathtt{a}, \mathtt{Y} \to \mathtt{a}$.
- One more try...

Step 1. Linear Tree Pattern Matching

$$
\begin{array}{rlrl}
p = & \mathtt{f(f(a,X),Y)} & s = & \mathtt{f(f(a,b),f(f(a,a),a))} \\
C_p = & \mathtt{124252131} & C_s = & \mathtt{1242521368696373}\textcolor{red}{\mathtt{1}} \\
E_p = & \mathtt{ffafXffYf} & E_s = & \mathtt{ffafbffffafaffaf}\textcolor{blue}{\mathtt{f}} \\
\sigma_1 = & \mathtt{ffaf} & M_1 = & \mathtt{100000010000}\textcolor{blue}{\mathtt{1}}\mathtt{0000} \\
\sigma_2 = & \mathtt{ff} & M_2 = & \mathtt{10000111000010010} \\
\sigma_3 = & \mathtt{f} & M_3 = & \mathtt{11010111100011011}
\end{array}
$$

- The first match found: $\mathtt{X} \to \mathtt{b}, \mathtt{Y} \to \mathtt{f(f(a,a),a)}$.
- The next attempt gives $\mathtt{X} \to \mathtt{a}, \mathtt{Y} \to \mathtt{a}$.
- One more try... fail.
- The last 1 in C_s is not the first occurrence of 1.

Complexity of Linear Tree Pattern Matching

- The simple algorithm computes $k + 1$ Boolean tables.
- Each table has $|E_s| = n$ size.
- In total, construction of the tables takes $O(nk)$ time.
- Room for improvement: Do not compute them explicitly.

Suffix Number, Suffix Index

Ψ: finite set of strings.

- Suffix number of a string λ in Ψ: The number of strings in Ψ which are suffixes of λ.
- Suffix index of Ψ (denoted Ψ^*): The maximum among all suffix numbers of strings in Ψ.
- If $|\Psi| = 0$ then $\Psi^* = 1$.

Example

- $\Psi = \{ffffX, ffffb, fffb, ffb, fb\}$. $|\Psi| = 5$.
- Suffix number of *ffffX* in Ψ is 1.
- Suffix number of *fffb* in Ψ is 3.
- Suffix number of *ffffb* in Ψ is 4.
- Suffix index of Ψ is 4.

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.
- If X at node i in p matches a subtree at node w in s then

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.
- If X at node i in p matches a subtree at node w in s then
  - w is called a legal replacement for X,
  - $path_1 \circ lp(r_p, i') = lp(r_s, w)$. ($i'$: i labeled with ($lab(w)$).)

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.
- If X at node i in p matches a subtree at node w in s then
  - w is called a legal replacement for X,
  - $\mathit{path}_1 \circ \mathit{lp}(r_p, i') = \mathit{lp}(r_s, w)$. ($i'$: i labeled with ($\mathit{lab}(w)$).)
- If another variable Y at node j in p matches w (in another match) then

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.
- If X at node i in p matches a subtree at node w in s then
  - w is called a legal replacement for X,
  - $\mathit{path}_1 \circ \mathit{lp}(r_p, i') = \mathit{lp}(r_s, w)$. ($i'$: i labeled with ($\mathit{lab}(w)$).)
- If another variable Y at node j in p matches w (in another match) then
  - $\mathit{path}_2 \circ \mathit{lp}(r_p, j') = \mathit{lp}(r_s, w)$.

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.
- If X at node i in p matches a subtree at node w in s then
  - w is called a legal replacement for X,
  - $path_1 \circ lp(r_p, i') = lp(r_s, w)$. ($i'$: i labeled with ($lab(w)$).)
- If another variable Y at node j in p matches w (in another match) then
  - $path_2 \circ lp(r_p, j') = lp(r_s, w)$.
- Therefore, $lp(r_p, j')$ is a suffix of $lp(r_p, i')$, or vice versa.

Complexity of Linear Tree Pattern Matching

How many replacements at most are possible (independent of the algorithm)?

- Assume p matches s at some node.
- If X at node i in p matches a subtree at node w in s then
  - w is called a legal replacement for X,
  - $path_1 \circ lp(r_p, i') = lp(r_s, w)$. ($i'$: i labeled with ($lab(w)$).)
- If another variable Y at node j in p matches w (in another match) then
  - $path_2 \circ lp(r_p, j') = lp(r_s, w)$.
- Therefore, $lp(r_p, j')$ is a suffix of $lp(r_p, i')$, or vice versa.
- Hence, the subtree at w can be substituted at most K^* times over all matches and the number of all legal replacements that can be computed over all matches is $O(nK^*)$.

Complexity of Linear Tree Pattern Matching

Bound on the number of replacements computed by the simple algorithm:

- Assume $e_1, \ldots, e_w$ are Euler strings of subtrees in s rooted at nodes $i_1, \ldots, i_w$.
- Assume the string $\sigma_1 \circ e_1 \circ \cdots \circ e_w \circ \sigma_{w+1}$ matches a substring of E_s at position l.
- l is the position that corresponds to the first occurrence of a node j in s, i.e. $C_s[l] = j$ and $C_s[l'] \neq j$ for all $l' < l$.
- For each $1 < q < w$, $lp(r_p, v_q) = lp(j, i_q)$ (v's are the corresponding variable nodes in p.)
- The strings σ_1, $\sigma_1 \circ e_1$, $\sigma_1 \circ e_1 \circ \sigma_2, \ldots$ are computed incrementally.
- We have a match at j if we compute $\sigma_1 \circ e_1 \circ \cdots \circ e_k \circ \sigma_{k+1}$, i.e. legal replacements for all variables in p.

Complexity of Linear Tree Pattern Matching

Bound on the number of replacements computed by the simple algorithm:

- In case of failed match attempt, we would have computed at most one illegal replacement.
- Hence, the total number of illegal replacements computed over match attempts at all nodes can be $O(n)$ at most.
- Therefore, the upper bound of the replacements computed by the algorithm is $O(nK^*)$.

Complexity of Linear Tree Pattern Matching

Bound on the number of replacements computed by the simple algorithm:

- In case of failed match attempt, we would have computed at most one illegal replacement.
- Hence, the total number of illegal replacements computed over match attempts at all nodes can be $O(n)$ at most.
- Therefore, the upper bound of the replacements computed by the algorithm is $O(nK^*)$.

That's fine, but how to keep the time-bound of the algorithm proportional to the number of replacements?

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Do not spend more than $O(1)$ between replacements, without computing the tables explicitly.
- Do a replacement in $O(1)$.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Do not spend more than $O(1)$ between replacements, without computing the tables explicitly.
- Do a replacement in $O(1)$.
- Doing a replacement in $O(1)$ is easy:
  - Store an Euler string in an array along with a pointer from the first occurrence of a node to its last occurrence.
  - Check whether the replacement begins at the first occurrence of a node.
  - If yes, skip to its last occurrence.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Do not spend more than $O(1)$ between replacements, without computing the tables explicitly.
- Do a replacement in $O(1)$.
- Doing a replacement in $O(1)$ is easy:
  - Store an Euler string in an array along with a pointer from the first occurrence of a node to its last occurrence.
  - Check whether the replacement begins at the first occurrence of a node.
  - If yes, skip to its last occurrence.
- Not spending more than $O(1)$ between replacements, without computing the tables, needs more preprocessing.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- What happens in the steps preceding the replacement for v_{i+1}, after computing a replacement for v_i?

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- What happens in the steps preceding the replacement for v_{i+1}, after computing a replacement for v_i?
- Determine whether pattern string σ_{i+1} matches E_s at the position following the replacement for v_i.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- What happens in the steps preceding the replacement for v_{i+1}, after computing a replacement for v_i?
- Determine whether pattern string σ_{i+1} matches E_s at the position following the replacement for v_i.
- Had we computed the tables, this can be done in $O(1)$, but how to achieve the same without the tables?

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:
  - Preprocess the pattern strings to produce an automaton that recognizes every instance of these $k+1$ strings.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:
  - Preprocess the pattern strings to produce an automaton that recognizes every instance of these $k+1$ strings.
  - Use the automaton to recognize these strings in E_s.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:
  - Preprocess the pattern strings to produce an automaton that recognizes every instance of these $k+1$ strings.
  - Use the automaton to recognize these strings in E_s.
  - With every position in an array containing E_s, store the state of the automaton on reading the symbol in that position.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:
  - Preprocess the pattern strings to produce an automaton that recognizes every instance of these $k+1$ strings.
  - Use the automaton to recognize these strings in E_s.
  - With every position in an array containing E_s, store the state of the automaton on reading the symbol in that position.
  - In order to decide whether a pattern string σ_i matches the substring of E_s at position j, look at the state of the automaton in position $j + |\sigma_i| - 1$.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:
  - Preprocess the pattern strings to produce an automaton that recognizes every instance of these $k+1$ strings.
  - Use the automaton to recognize these strings in E_s.
  - With every position in an array containing E_s, store the state of the automaton on reading the symbol in that position.
  - In order to decide whether a pattern string σ_i matches the substring of E_s at position j, look at the state of the automaton in position $j + |\sigma_i| - 1$.
  - Lookup from the array in $O(1)$ time.

Complexity of Linear Tree Pattern Matching

Keep the time-bound of the algorithm proportional to the number of replacements:

- Problem: Given a position in E_s and a string σ_i, $1 \leq i \leq k+1$, decide in $O(1)$ whether σ_i matches E_s in that position.
- Idea:
  - Preprocess the pattern strings to produce an automaton that recognizes every instance of these $k+1$ strings.
  - Use the automaton to recognize these strings in E_s.
  - With every position in an array containing E_s, store the state of the automaton on reading the symbol in that position.
  - In order to decide whether a pattern string σ_i matches the substring of E_s at position j, look at the state of the automaton in position $j + |\sigma_i| - 1$.
  - Lookup from the array in $O(1)$ time.

How to preprocess the pattern strings?

Modifying the Linear Tree Pattern Matching

Pattern string preprocessing:

- The $k + 1$ pattern strings $\sigma_1, \ldots, \sigma_{k+1}$ are preprocessed to produce an automation that recognizes every instance of these strings.
- Method: Aho-Corasick (AC) algorithm.
- The AC algorithm constructs the desired automaton in time proportional to the sum of the lengths of all pattern strings.

Modifying the Linear Tree Pattern Matching

What does a Aho-Corasick automaton for a set of pattern strings $\sigma_1, \ldots, \sigma_{k+1}$ do?

- Takes the subject string E_s as input.
- Outputs the locations in E_s at which the σ's appear as substrings, together with the corresponding σ's.
- For example, a Aho-Corasick automaton for the strings *he, she, his, hers* returns on the input string *ushers* the locations 4 (match for *she* and *he*) and 6 (match for *hers*).

Modifying the Linear Tree Pattern Matching

Aho-Corasick automaton

- consists of a set of states, represented by numbers,
- processes the subject string by successively reading symbols in it, making state transitions and occasionally emitting output,
- is controlled by three functions:
  1. a goto function g,
  2. a failure function f,
  3. a output function *output*.

Modifying the Linear Tree Pattern Matching

Construction of the Aho-Corasick automation:

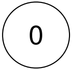
- Determine the states and the goto function.
- Compute the failure function.
- Computation of the output function begins on the first step and is completed on the second.

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : $\mathit{states} \times \mathit{letters} \to \mathit{states} \cup \{\mathit{fail}\}$:

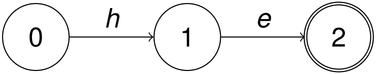


Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{$*fail*$\}$:



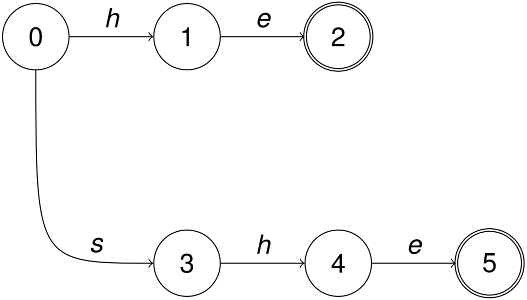
$output(2) = \{he\}$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function $g : \mathit{states} \times \mathit{letters} \rightarrow \mathit{states} \cup \{\mathit{fail}\}$:



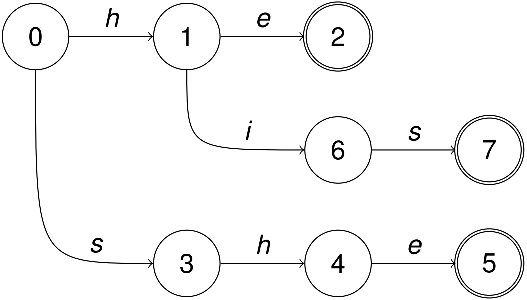
$\mathit{output}(2) = \{\mathit{he}\}$
$\mathit{output}(5) = \{\mathit{she}\}$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function $g : states \times letters \to states \cup \{fail\}$:



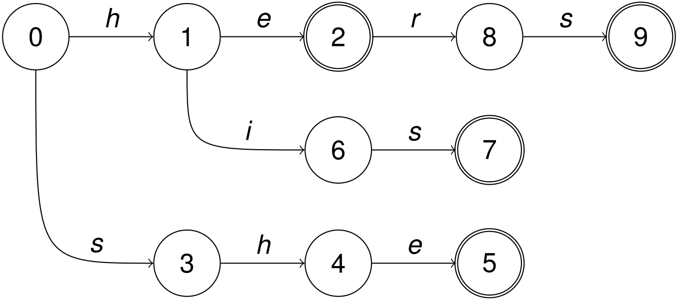
$output(2) = \{he\}$
$output(5) = \{she\}$
$output(7) = \{his\}$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function $g : \mathit{states} \times \mathit{letters} \rightarrow \mathit{states} \cup \{\mathit{fail}\}$:



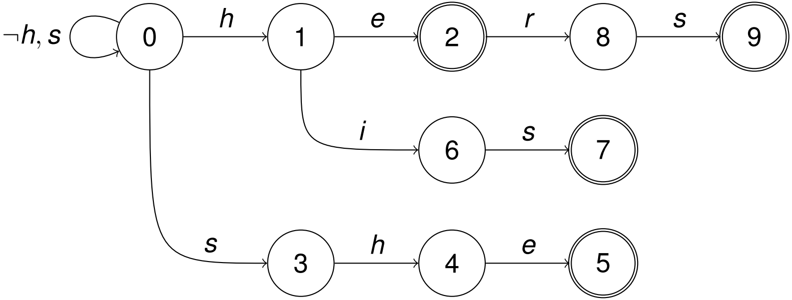
$\mathit{output}(2) = \{\mathit{he}\}$
$\mathit{output}(5) = \{\mathit{she}\}$
$\mathit{output}(7) = \{\mathit{his}\}$
$\mathit{output}(9) = \{\mathit{hers}\}$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function $g : states \times letters \rightarrow states \cup \{fail\}$:



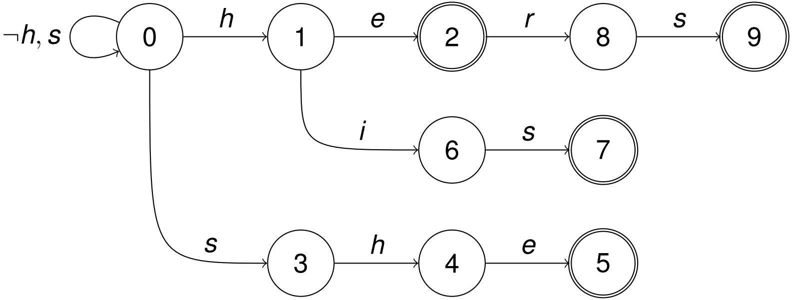
$output(2) = \{he\}$
$output(5) = \{she\}$
$output(7) = \{his\}$
$output(9) = \{hers\}$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : $states \times letters \to states \cup \{fail\}$:



$output(2) = \{he\}$
$output(5) = \{she\}$
$output(7) = \{his\}$
$output(9) = \{hers\}$

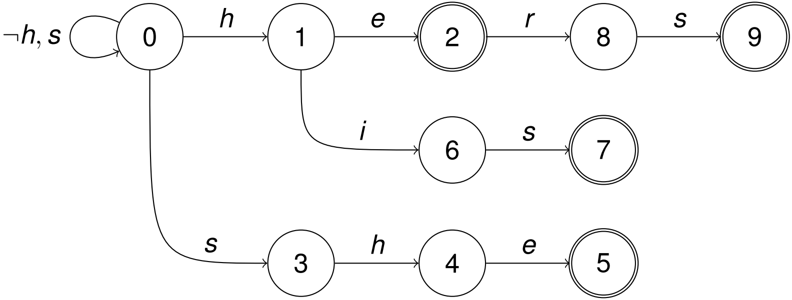
$f(1) = f(3) = 0$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{fail\}$:



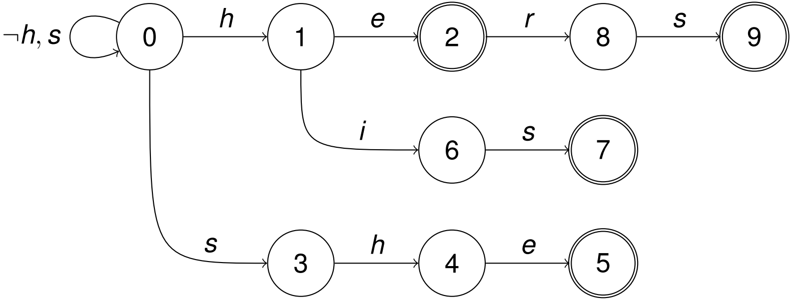
$$output(2) = \{he\} \qquad f(1) = f(3) = 0$$
$$output(5) = \{she\} \qquad f(2) = 0$$
$$output(7) = \{his\}$$
$$output(9) = \{hers\}$$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{$*fail*$\}$:



$output(2) = \{he\}$
$output(5) = \{she\}$
$output(7) = \{his\}$
$output(9) = \{hers\}$

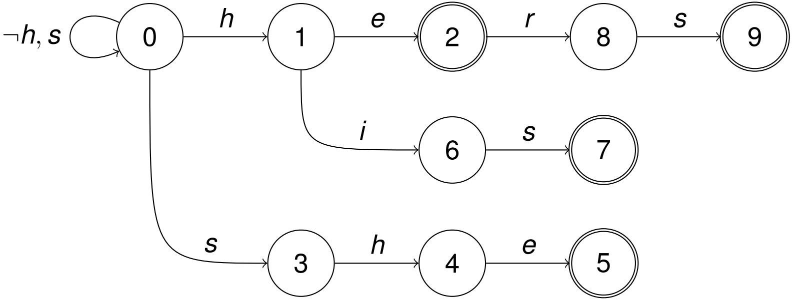
$f(1) = f(3) = 0$
$f(2) = 0$
$f(6) = 0$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function $g : \mathit{states} \times \mathit{letters} \rightarrow \mathit{states} \cup \{\mathit{fail}\}$:



$\mathit{output}(2) = \{he\}$ $\qquad f(1) = f(3) = 0$

$\mathit{output}(5) = \{she\}$ $\qquad f(2) = 0$

$\mathit{output}(7) = \{his\}$ $\qquad f(6) = 0$

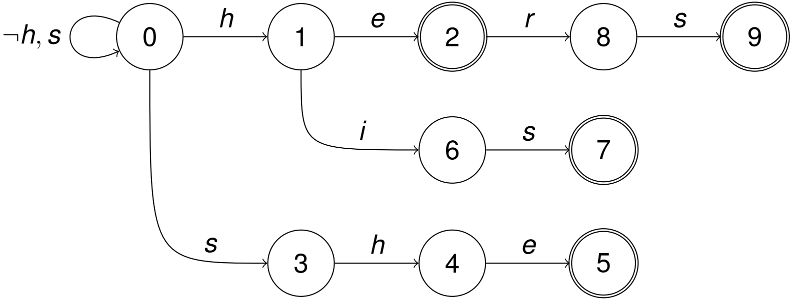
$\mathit{output}(9) = \{hers\}$ $\qquad f(4) = 1$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{$*fail*$\}$:



$output(2) = \{he\}$ $\quad f(1) = f(3) = 0 \quad f(8) = 0$

$output(5) = \{she\}$ $\quad f(2) = 0$

$output(7) = \{his\}$ $\quad f(6) = 0$

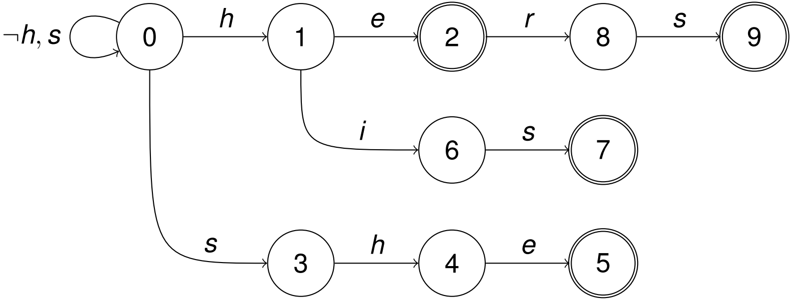
$output(9) = \{hers\}$ $\quad f(4) = 1$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{fail\}$:



$output(2) = \{he\}$
$output(5) = \{she\}$
$output(7) = \{his\}$
$output(9) = \{hers\}$

$f(1) = f(3) = 0$
$f(2) = 0$
$f(6) = 0$
$f(4) = 1$

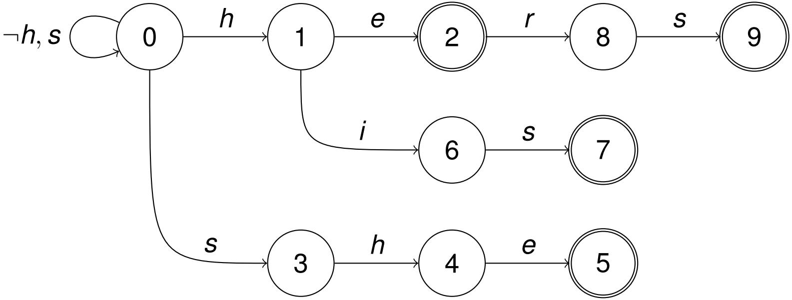
$f(8) = 0$
$f(7) = 3$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{fail\}$:



$output(2) = \{he\}$ $\quad f(1) = f(3) = 0$ $\quad f(8) = 0$

$output(5) = \{she, he\}$ $\quad f(2) = 0$ $\quad f(7) = 3$

$output(7) = \{his\}$ $\quad f(6) = 0$ $\quad f(5) = 2$

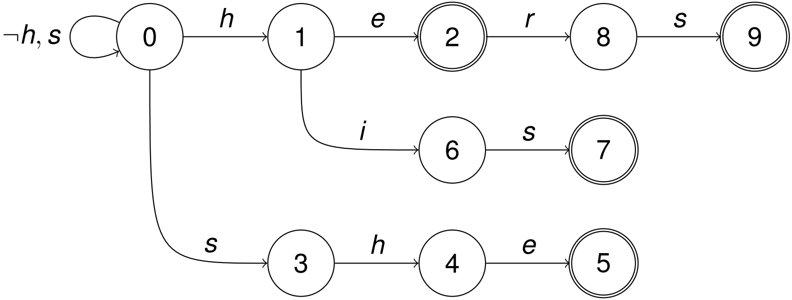
$output(9) = \{hers\}$ $\quad f(4) = 1$

Modifying the Linear Tree Pattern Matching

Example

Construction of the Aho-Corasick automation for the pattern strings *he, she, his, hers*.

The goto function g : *states* $\times$ *letters* $\rightarrow$ *states* $\cup \{fail\}$:

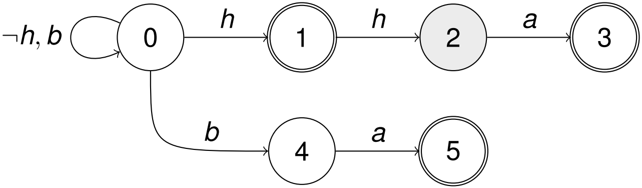


$output(2) = \{he\}$	$f(1) = f(3) = 0$	$f(8) = 0$
$output(5) = \{she, he\}$	$f(2) = 0$	$f(7) = 3$
$output(7) = \{his\}$	$f(6) = 0$	$f(5) = 2$
$output(9) = \{hers\}$	$f(4) = 1$	$f(9) = 3$

Modifying the Linear Tree Pattern Matching

Example

AC automation for the pattern strings *hha, h, ba*.



$output'(1) = \{h\}$ $\quad output'(2) = \{h\}$

$output'(3) = \{hha\}$

$output'(5) = \{ba\}$

$f(1) = 0 \quad f(5) = 0$

$f(4) = 0 \quad f(3) = 0$

$f(2) = 1$

$1, 3, 5$: primary accepting states.

2 : secondary accepting state (*h* is a suffix of *hh*).

Modifying the Linear Tree Pattern Matching

- For each secondary accepting state there is a unique primary accepting state with exactly the same output set.
- Modify construction of AC automaton by maintaining pointers from secondary accepting states to the corresponding accepting states.

Modifying the Linear Tree Pattern Matching

- Construction of Aho-Corasick automaton takes $O(m)$ time.
- The output set is represented as a linked list, which is inappropriate for our purpose.
- Given an arbitrary string, we want to determine in constant time whether it is in the output set.

Modifying the Linear Tree Pattern Matching

- Construction of Aho-Corasick automaton takes $O(m)$ time.
- The output set is represented as a linked list, which is inappropriate for our purpose.
- Given an arbitrary string, we want to determine in constant time whether it is in the output set.
- Idea: Copy the output set into an array.

Modifying the Linear Tree Pattern Matching

- Construction of Aho-Corasick automaton takes $O(m)$ time.
- The output set is represented as a linked list, which is inappropriate for our purpose.
- Given an arbitrary string, we want to determine in constant time whether it is in the output set.
- Idea: Copy the output set into an array.
- Question: How many elements do we have to copy?

Modifying the Linear Tree Pattern Matching

- Construction of Aho-Corasick automaton takes $O(m)$ time.
- The output set is represented as a linked list, which is inappropriate for our purpose.
- Given an arbitrary string, we want to determine in constant time whether it is in the output set.
- Idea: Copy the output set into an array.
- Question: How many elements do we have to copy?
- Answer: As many as in the output sets of all primary accepting states, which is $O(m)$ (because any string in a primary accepting state is a suffix of the longest string in this state.)

Modifying the Linear Tree Pattern Matching

Linear Tree Pattern Matching:

1. Construct Aho-Corasick automaton for the pattern strings.
2. Visit each primary accepting state and copy its output set into a boolean array.
3. Scan the E_s with this automaton.
4. During this process, with each entry in E_s store the state of automaton upon reading the function symbol in that entry.
5. If this state is a secondary accepting state, instead of it store the corresponding primary accepting state.
6. To determine whether there is a match for a pattern string σ at the position i requires verifying the state associated with the $i + |\sigma| - 1$'th entry in E_s:
   - This should be a primary accepting state and
   - Its output set should contain σ.

Consistency Checking

For nonlinear patterns the computed replacements have to be checked for consistency.

- Idea: Assign integer codes (from 1 to n) to the nodes in the subject tree.
- Two nodes get the same encoding iff the subtrees rooted at them are identical.
- Such an encoding can be computed in $O(n)$.

Consistency Checking

Computing the encoding:

- Bottom up: First, sort the leaves with respect to their labels and take the ranks as the integers for encoding. Duplicates are assigned the same rank.
- Suppose the encoding for all nodes up to the height i is computed.
- Computing the encoding of the nodes at height $i + 1$:
  - Assign to each node v at the level $i + 1$ a vector $\langle f, j_1, \ldots, j_n \rangle$.
  - f is the label of v and j_i is the encoding of its i's child.
  - The vectors assigned to all nodes at $i + 1$ are radix sorted.
  - If the rank of v is α and the largest encoding among the nodes at level i is β, then the encoding for v is $\alpha + \beta$.

Consistency Checking

Checking consistency:

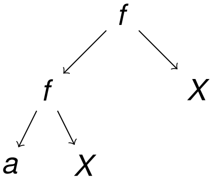
- Consistency of replacements is checked as they are computed.
- For each variable in the pattern, the encoding for the replacement of its first occurrence is computed and is entered into a table.
- For the next occurrence of the same variable, compare encoding of its replacement to the one in the table.
- If the check succeeds, proceed further. Otherwise report a failure and start matching procedure at another position in E_s.
- These steps do not increase the complexity of the algorithm.

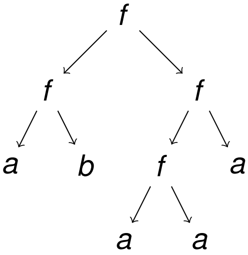
The Last Word

Nonlinear tree pattern matching can be done in $O(nK^*)$ time.

Example

$f(f(a, X), X)$ $\qquad$ $f(f(a, b), f(f(a, a), a))$

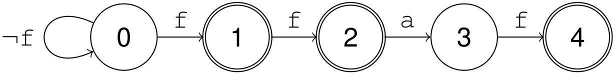




Pattern tree p $\qquad$ Subject tree s

Example (Cont.)

- $E_p = \texttt{ffafXffXf}$
- AC automation for the pattern strings `ffaf, ff, f`.



$output(1) = \{\texttt{f}\}$
$output(2) = \{\texttt{ff}, \texttt{f}\}$
$output(4) = \{\texttt{ffaf}, \texttt{f}\}$
$failure(1) = failure(3) = 0$
$failure(2) = failure(4) = 1$

	`ffaf`	`ff`	`f`
O_1	F	F	T
O_2	F	T	T
O_4	T	F	T

Example (Cont.)

$\sigma_1 = \texttt{ffaf}$, $\sigma_2 = \texttt{ff}$, $\sigma_3 = \texttt{f}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
C_s	1	2	4	2	5	2	1	3	6	8	6	9	6	3	7	3	1
E_s	f	f	a	f	b	f	f	f	f	a	f	a	f	f	a	f	f
IsFirst	T	T	T	F	T	F	F	T	T	T	F	T	F	F	T	F	F
lastptr	17	6	3	–	5	–	–	16	13	8	–	9	–	–	15	–	–
IsLast	F	F	T	F	T	T	F	F	F	T	F	T	T	F	T	T	T
state	1	2	3	4	0	1	2	2	2	3	4	0	2	2	3	4	2

- In the first row, the numbers from `1` to `17` - array indices.
- C_s and E_s - the Euler chain and the Euler string for s.
- For an index `i`,
  - $\mathit{IsFirst}[\texttt{i}] = \texttt{T}$ iff $C_s[\texttt{i}]$ occurs first time in C_s.
  - if $\mathit{IsFirst}[\texttt{i}] = \texttt{T}$ then $\mathit{lastptr}[\texttt{i}] = \texttt{j}$ where `j` is the index of the last occurrence of the number $C_s[\texttt{i}]$ in C_s.
  - $\mathit{IsLast}[\texttt{i}] = \texttt{T}$ iff $C_s[\texttt{i}]$ occurs last time in C_s.
  - $\mathit{state}[\texttt{i}]$ is the state of the automaton after reading $E_s[\texttt{i}]$.

Reference

R. Ramesh and I. V. Ramakrishnan.
Nonlinear pattern matching in trees.
J. ACM, 39(2):295–316, 1992.