String matching

The finite automaton approach. The Aho-Corasick algorithm. Suffix trees. Ukkonen algorithm

November 2020

- An alphabet Σ is a finite set of characters.
- A string S of length $n \geq 0$ is an array $S[1..n]$ of characters from Σ. We write $|S|$ for the length of S. Thus, $|S| = n$
- $S[i]$ is the character of S at position i
- $S[i..j]$ represents the substring of S form position i to position j inclusively.

Example

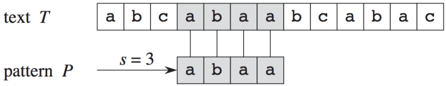
If $S = \texttt{alphabet}$ then $|S| = 8$, $S[1] = \texttt{a}$, $S[2] = \texttt{b}$, $S[1..4] = \texttt{alph}$, $S[3..7] = \texttt{phabe}$

ASSUMPTIONS:

- Σ : finite set of characters (an alphabet).
  E.g., $\Sigma = \{a, b, \ldots, z\}$
- $P[1..m]$: array of $m > 0$ characters from Σ (the pattern)
- $T[1..n]$: array of $n > 0$ characters from Σ (the text)

We say that P occurs with shift s in T (or, equivalently, that P occurs beginning at position $s + 1$ in T) if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$ (that is, if $T[s + j] = P[j]$, for $1 \leq j \leq m$).

EXAMPLE:



The string matching problem

Given a pattern $P[1..m]$ and a text $T[1..n]$

Find all shifts s where P occurs in T.

Terminology and notation:

- Σ^*=the set of all strings of characters from Σ
- If $x, y \in \Sigma^*$ then
  - $x\,y$:=the concatenation of x with y
  - $|x| :=$ the length (number of characters) of x
  - ϵ :=the zero-length empty string
  - x is prefix of y, notation $x \sqsubseteq y$, if $y = x\,w$ for some $w \in \Sigma^*$.
    x is suffix of y, notation $x \sqsupseteq y$, if $y = w\,x$ for some $w \in \Sigma^*$.

  Example: ab $\sqsubseteq$ abcca

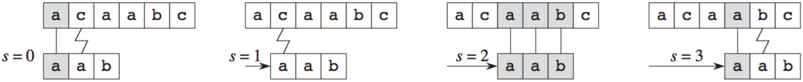
REMARKS

1. $x \sqsupseteq y$ if and only if $x\,a \sqsupseteq y\,a$.
2. Every string is either ϵ, or of the form wa where $a \in \Sigma$ and w a string.

The naive string matching algorithm

NAIVESTRINGMATCHER(T, P)
$n := T.length$
$m := P.length$
for $s = 0$ **to** $n - m$
    if $P[1..m] == T[s + 1..s + m]$
        print "pattern occurs with shift" s

EXAMPLE:



(a) (b) (c) (d)

- Time complexity: $O((n - m + 1)\, m)$
  - Several character comparison are performed repeatedly
  - **Can we do better?**

String matching with finite automata

Definition (Finite automaton)

A finite automaton is a 5-tuple $\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ where

- Q : finite set of states
- $q_0 \in Q$: the start state
- $A \subseteq Q$: distinguished set of accepting states
- Σ:=finite set of characters (the input alphabet)
- $\delta : Q \times \Sigma \to Q$ is the transition function

String matching with finite automata

Definition (Finite automaton)

A finite automaton is a 5-tuple $\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ where

- Q : finite set of states
- $q_0 \in Q$: the start state
- $A \subseteq Q$: distinguished set of accepting states
- Σ:=finite set of characters (the input alphabet)
- $\delta : Q \times \Sigma \to Q$ is the transition function

Alternative representations of a finite automaton:

1. Tabular representation of δ
2. state-transition diagram

(see next slide)

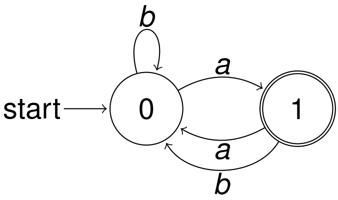
Alternative representations of a finite automaton

$\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ where
$Q = \{0, 1\}, q_0 = 0, A = \{1\}, \Sigma = \{a, b\}$

- Tabular representation:

δ	a	b
$\rightarrow 0$	1	0
$\leftarrow 1$	0	0

- State-transition diagram:



Acceptance by finite automata

ASSUMPTION: $\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ is a finite automaton.

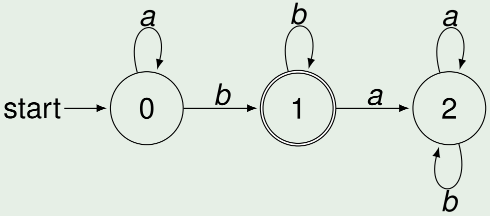
- Define inductively $\phi : \Sigma^* \to Q$, as follows:

$$\phi(\epsilon) := q_0,$$
$$\phi(wa) := \delta(\phi(w), a).$$

We say that w is accepted by $\mathcal{A}$ if $\phi(w) \in A$.

Example

The following finite automaton accepts all (and only) words of the form $a^m b^n$ where $m \geq 0$, $n \geq 1$:



REMARK: The time complexity of computing $\phi(w)$ is $O(n)$ where $n = |w|$.

A finite automaton for the string matching problem

Main ideas

- Define a finite automaton $\mathcal{A}$ such that $T[1..i]$ is accepted by $\mathcal{A}$ if and only if it has suffix P (that is, $P \sqsupseteq T[1..i]$).
- $\mathcal{A}$ can be defined in a preprocessing step of $P[1..m]$
  - To understand the construction of $\mathcal{A}$, we shall define the **suffix function** σ corresponding to pattern P:

- Define a finite automaton $\mathcal{A}$ such that $T[1..i]$ is accepted by $\mathcal{A}$ if and only if it has suffix P (that is, $P \sqsupseteq T[1..i]$).
- $\mathcal{A}$ can be defined in a preprocessing step of $P[1..m]$
  - To understand the construction of $\mathcal{A}$, we shall define the **suffix function** σ corresponding to pattern P:

Definition

The suffix function corresponding to pattern $P[1..m]$ is the function $\sigma : \Sigma^* \to \{0, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x. Formally:

$$\sigma(x) := \max\{k \mid 0 \leq k \leq m \text{ and } P[1..k] \sqsupseteq x\}.$$

A finite automaton for the string matching problem
Main ideas

- Define a finite automaton $\mathcal{A}$ such that $T[1..i]$ is accepted by $\mathcal{A}$ if and only if it has suffix P (that is, $P \sqsupseteq T[1..i]$).
- $\mathcal{A}$ can be defined in a preprocessing step of $P[1..m]$
  - To understand the construction of $\mathcal{A}$, we shall define the **suffix function** σ corresponding to pattern P:

Definition

The suffix function corresponding to pattern $P[1..m]$ is the function $\sigma : \Sigma^* \to \{0, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x. Formally:

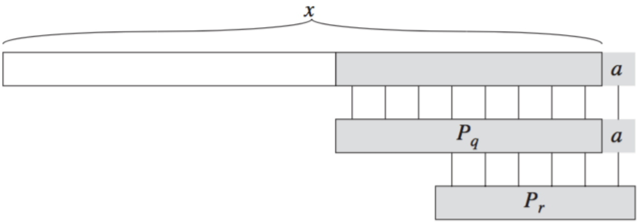
$$\sigma(x) := \max\{k \mid 0 \leq k \leq m \text{ and } P[1..k] \sqsupseteq x\}.$$

EXAMPLES: If $P = \texttt{ab}$ then $\sigma(\epsilon) = 0$, $\sigma(\texttt{cca}\underline{\texttt{a}}) = 1$, $\sigma(\texttt{ac}\underline{\texttt{ab}}) = 2$.

Suffix-function recursion lemma

For any string x and character $a \in \Sigma$, if $q = \sigma(x)$, then $\sigma(x\,a) = \sigma(P[1..q]\,a)$.

A graphical illustration of a proof of this Lemma is shown below:



The finite automaton corresponding to a pattern

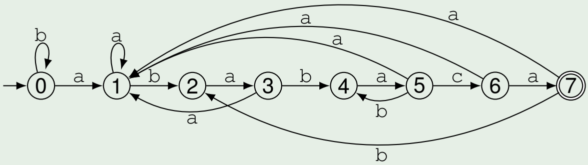
ASSUMPTION: $P[1..m]$ is the given pattern,
The corresponding finite automaton is $\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ where:

- $Q = \{0, 1, 2, \ldots, m\}$
- $q_0 = 0$
- $A = \{m\}$

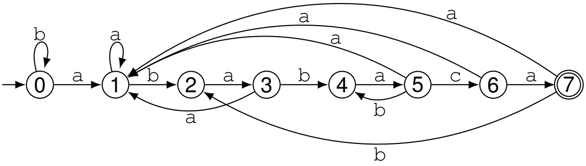
$\delta(q, a) = \sigma(P[1..q]\, a)$

Example

The finite automaton corresponding to $P[1..7] =$ `ababaca` is



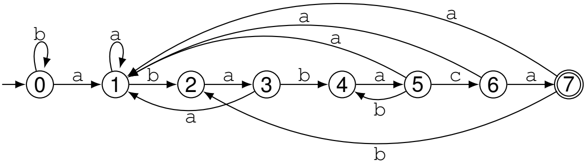
The missing transitions from a node point to state 0.



i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

The finite automaton corresponding to a pattern

Illustrated example



i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

The remaining question is:

How to compute the state transition function δ of $\mathcal{A}$?

Computing the transition function

A naive implementation (pseudocode)

```
ComputeTransitionFunction(P, Σ)
m := P.length
for q := 0 to m
    for each character a ∈ Σ
        k := min(m, q + 1) + 1
        repeat
            k := k − 1
        until P[1..k] ⊐ P[1..q] a
        δ(q, a) := k
return δ
```

Time complexity: $O(m^3 |\Sigma|)$.

There are better algorithms, which can compute δ with time complexity $O(m |\Sigma|)$.

We assume given

- $T[1..m]$ called text
- A finite set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$

Find **all** positions where some $P \in \mathcal{P}$ occurs in T.

We assume given

- $T[1..m]$ called text
- A finite set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$

Find **all** positions where some $P \in \mathcal{P}$ occurs in T.

USEFUL AUXILIARY NOTIONS

1. keyword tree $\mathcal{K}$ of the set $\mathcal{P}$
2. failure links between the nodes of $\mathcal{K}$

The keyword tree of a set of patterns $\mathcal{P} = \{P_1, \ldots, P_z\}$ is a tree $\mathcal{K}$ which satisfies 3 conditions:

1. every edge is labeled with exactly 1 character.
2. Distinct edges which leave from a node are labeled with distinct characters.
3. Every pattern $P_i \in \mathcal{P}$ gets mapped to a unique node v of $\mathcal{K}$ as follows: the string of characters along the branch from root to node v is P_i, and every leaf node of $\mathcal{K}$ is the mapping of a pattern from $\mathcal{P}$.

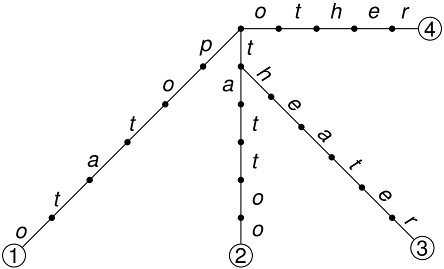
The keyword tree of a set of patterns $\mathcal{P} = \{P_1, \ldots, P_z\}$ is a tree $\mathcal{K}$ which satisfies 3 conditions:

1. every edge is labeled with exactly 1 character.
2. Distinct edges which leave from a node are labeled with distinct characters.
3. Every pattern $P_i \in \mathcal{P}$ gets mapped to a unique node v of $\mathcal{K}$ as follows: the string of characters along the branch from root to node v is P_i, and every leaf node of $\mathcal{K}$ is the mapping of a pattern from $\mathcal{P}$.

NOTATION: for every node $v \in \mathcal{K}$, $\mathcal{L}(v)$ is the string of characters along the branch of $\mathcal{K}$ from root to node v.

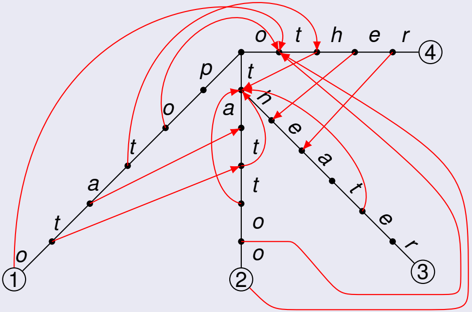
1. Keyword tree

Example for $\mathcal{P} = \{potato, tattoo, theater, other\}$



Let $\mathcal{K}$ be the keyword tree for $\mathcal{P} = \{P_1, \ldots, P_z\}$. Every node v of $\mathcal{K}$ has only one failure link to the node n_v of $\mathcal{K}$ which has the following property: $\mathcal{L}(n_v)$ is the longest proper suffix of $\mathcal{L}(v)$ which is a prefix of a pattern from $\mathcal{P}$.

Example for $\mathcal{P} = \{potato, tattoo, theater, other\}$



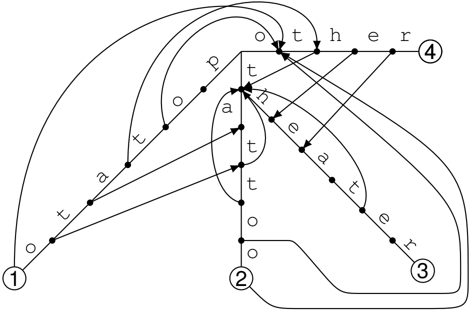
the failure links which are not depicted, go to the root of $\mathcal{K}$

Aho-Corasick algorithm

Allows to find all occurrences of $\mathcal{P}$ in $T[1..m]$ in time $O(m)$. It relies on the keyword tree $\mathcal{K}$ for $\mathcal{P}$ and its failure links.
The characters of $T[1..m]$ are read from left to right:

1. crt :=root of $\mathcal{K}$
   $i := 1$
2. If $\mathcal{L}(crt) = P_j$ or there is a sequence of failure links $crt \to \ldots \to w$ with $\mathcal{L}(w) = P_j$
   - signal "P_j occurs at position i in T"
3. If $i = m$ then STOP.
4. If $T[i] = c$ and there is an edge $crt \overset{c}{-} v$ then $i := i + 1, crt := v$, goto 2.
5. If $T[i] = c$ and there is no edge $crt \overset{c}{-} v$ then let $crt \to \ldots \to v$ the shortest sequence of failure links such that $\exists v \overset{c}{-} w$ an let $crt := v$.
   If no such sequence exists, let $crt :=$ root of $\mathcal{K}$.
6. goto 2.

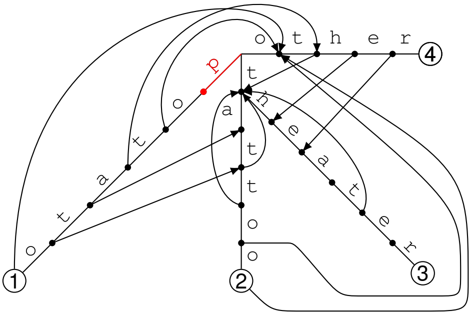
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$



potheater

Aho-Corasick algorithm

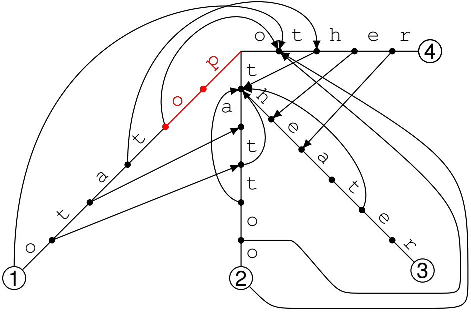
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$




```
potheater
△
```

Aho-Corasick algorithm

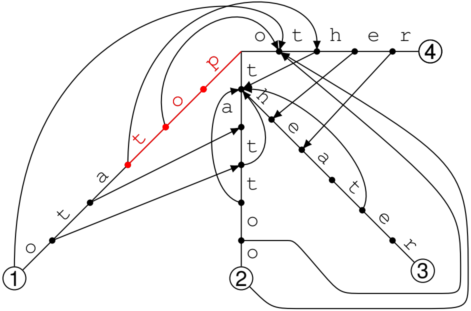
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$



potheater
△ △

Aho-Corasick algorithm

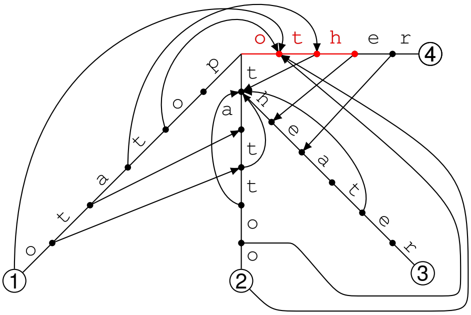
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$




```
potheater
△ △ △
```

Aho-Corasick algorithm

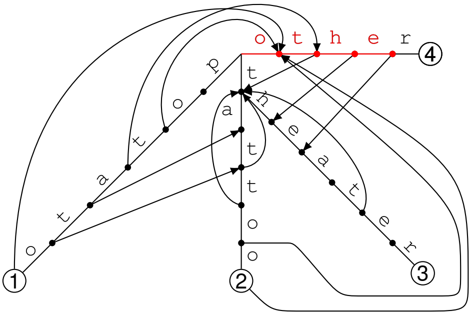
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$




```
potheater
△△△△
```

Aho-Corasick algorithm

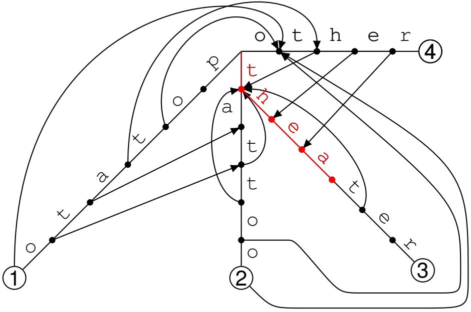
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$




```
potheater
△△△△△
```

Aho-Corasick algorithm

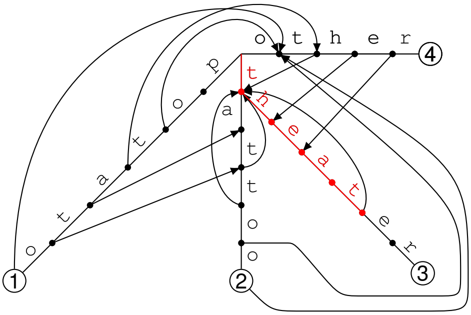
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$



potheater
△△△△△△

Aho-Corasick algorithm

Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$

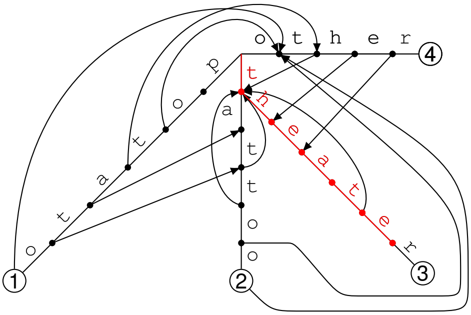


potheater

△△△△△△△

Aho-Corasick algorithm

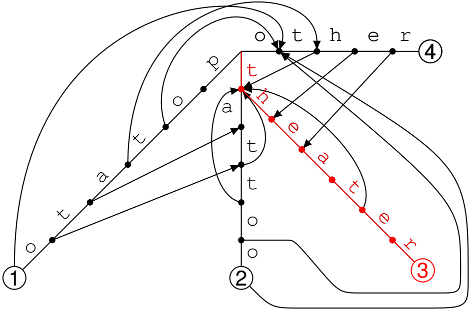
Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$




```
potheater
△△△△△△△△△
```

Aho-Corasick algorithm

Illustrated example: $\mathcal{P} = \{potato, tattoo, theater, other\}$, $T = potheater$



potheater

△△△△△△△△△

$\Rightarrow$ detected occurrence of $P_3 =$ theater

Aho-Corasick algorithm

The construction of the suffix tree and of the failure links in time $O(n)$

$\mathcal{P} = \{P_1, \ldots, P_z\}$, $n := |P_1| + \ldots + |P_z|$

- The keyword tree $\mathcal{K}$ for $\mathcal{P}$ is built by adding repeatedly the edges for $P_1, \ldots, P_z$ to an initially empty tree.
  - The addition of the edges for P_i has runtime complexity $O(|P_i|)$

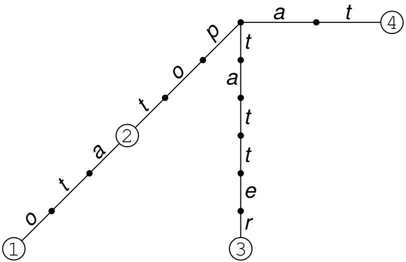
  $\Rightarrow$ the construction of $\mathcal{K}$ has runtime complexity $O(|P_1| + \ldots + |P_z|) = O(n)$
- The failure links are added to each node of $\mathcal{K}$ in the order of a breadth-first traversal: If r is the root of $\mathcal{K}$ then
  - add a failure link for the root of $\mathcal{K}$: $r \rightarrow r$
  - for the nodes of v at tree depth 1: add failure links $v \rightarrow r$
  - if v is a node at depth $k > 1$, then let
    - v' be the parent of v
    - x be the label of $v - v'$
    - $\pi : v' \rightarrow v_1 \rightarrow \ldots v_i$ be the shortest sequence of failure links such that there is an edge $v_i - w$ in $\mathcal{K}$ with label x

    If π exists: add the failure link $v \rightarrow w$

    If π does not exist: add the failure link $v \rightarrow r$

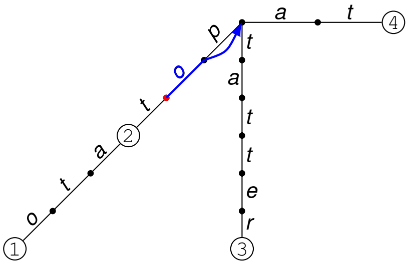
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



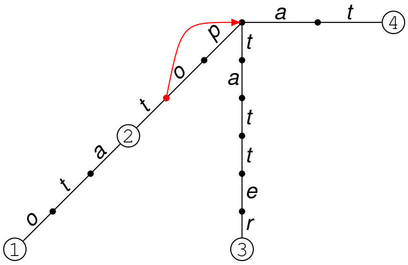
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



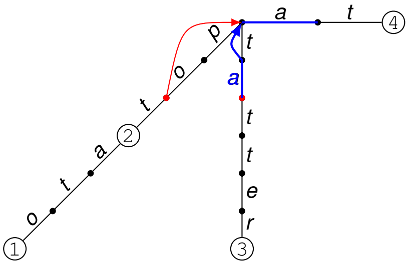
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



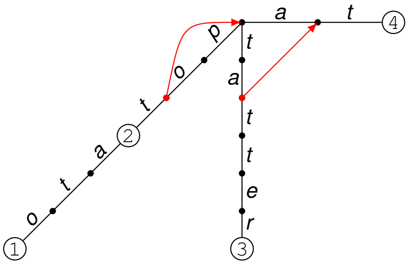
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



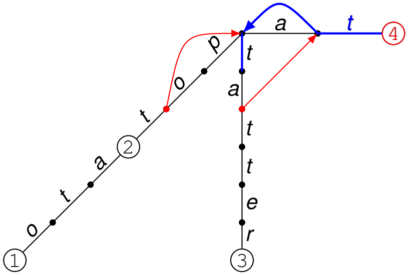
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



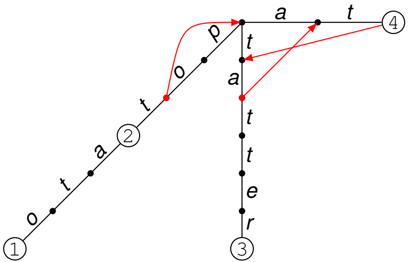
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



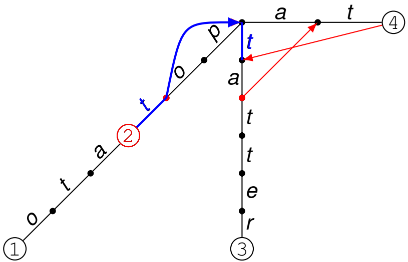
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



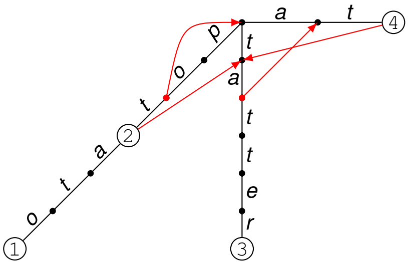
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



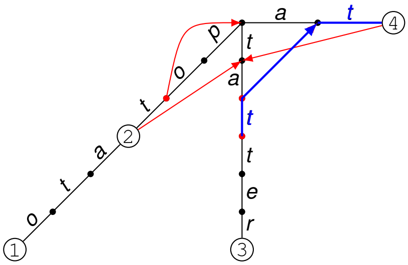
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



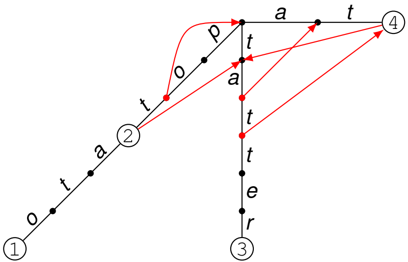
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



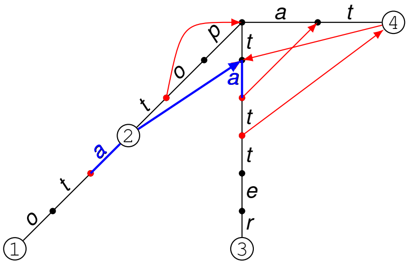
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



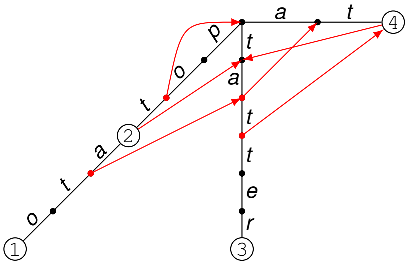
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



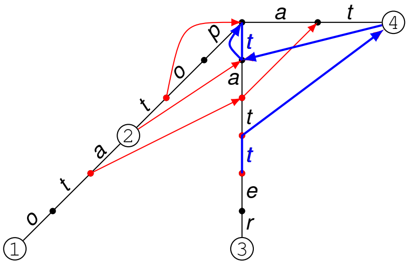
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



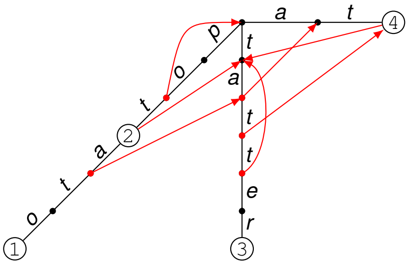
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



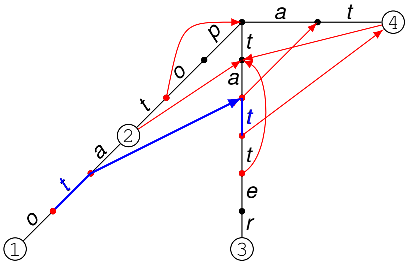
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



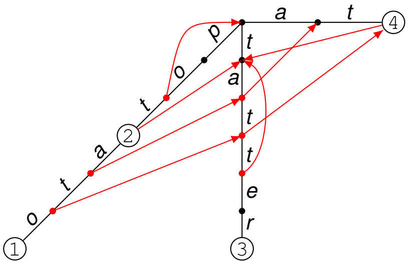
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



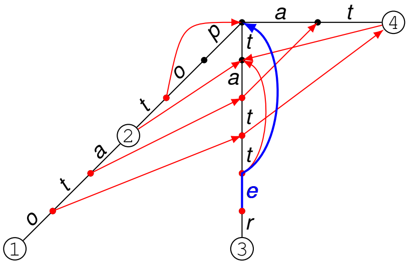
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



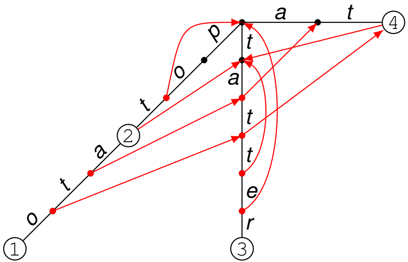
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



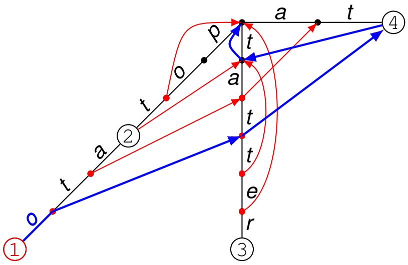
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



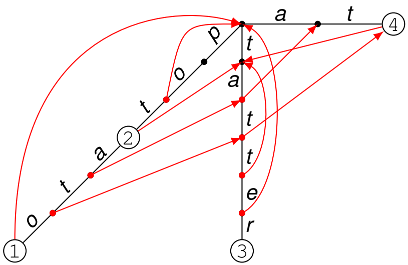
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



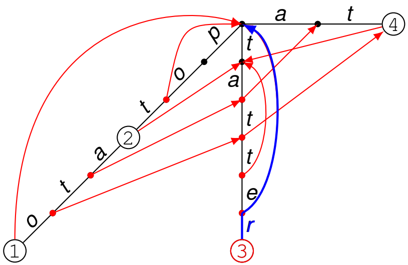
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



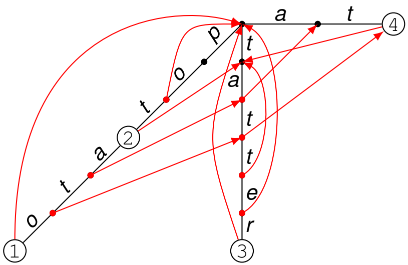
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



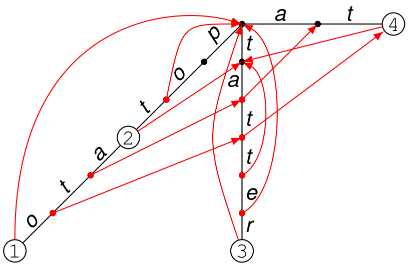
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



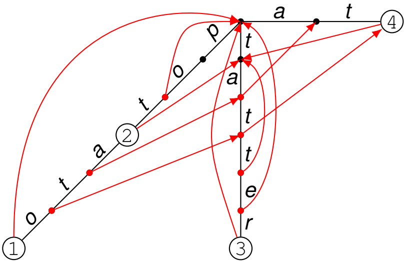
Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



Addition of failure links to a keyword tree

Illustrated example for the keyword tree of $\mathcal{P} = \{potato, pot, tatter, at\}$



REMARK: The runtime complexity of this algorithm for the computation of failure links is $O(n)$, where $n = |P_1| + \ldots + |P_z|$

- A proof of this fact can be found in the recommended bibliography.

Suffix trees
What are they?

- A tree-like data structure for a large string (the text $T[1..n]$), which can be built in time $O(n)$
  - it is a compact representation of all suffixes of text T.
- It allows to find all occurrences of a pattern $P[1..m]$ in T in time $O(m+k)$ where k is the number of occurrences of P in T.

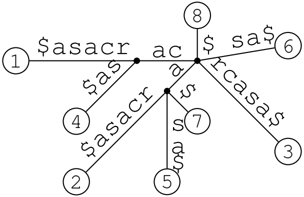
REMARKS

1. The algorithm which builds the suffix tree of $T[1..n]$ in linear time $O(n)$ was discovered by Wiener in 1973.
   - Donald Knuth called it "the algorithm of 1973" – he thought the suffix tree can not be built in linear time.
2. Suffix trees have many other interesting applications.

The suffix tree of a string $S[1..n]$ is a tree with the following properties:

1. It has exactly n leaf nodes, labeled with numbers $1, 2, \ldots, n$.
2. Except for the root, every internal node has at lest two children.
3. Every edge is labeled with a nonempty substring of S.
4. Edges from same node to different children are labeled with substrings that start with different characters.
5. The string produced by concatenating the labels of the edges from the root node to a leaf node i is the suffix $S[i..n]$.

S =carcasa$ has length 8, thus 8 suffixes.
The suffix tree of S is



Remarks

1. Some strings have no suffix trees.
2. If the last character of S occurs only once in S, then S has a suffix tree.

   From now on, we will assume S satisfies this condition.

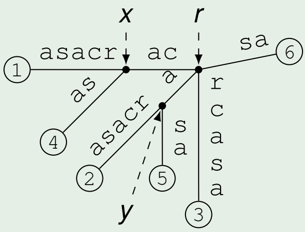
Auxiliary notions

Let $\mathcal{T}$ be the suffix tree of a string $S[1..n]$, and $\alpha = S[i..j]$ a substring of S.

- The label $\mathcal{L}(x)$ of a node x of $\mathcal{T}$ is the string produced by concatenating the labels of edges from root to x.
- The position $pos_{\mathcal{T}}(\alpha)$ of α in $\mathcal{T}$ is defined as follows: Let x be the node of $\mathcal{T}$ such that $\mathcal{L}(x)$ is the shortest node label with prefix α. (Note: x can be foud in $|\alpha|$ steps)
  1. If $\mathcal{L}(x) = \alpha$, then $pos_{\mathcal{T}}(\alpha) := x$
  2. Otherwise, let y be the parent node of x in $\mathcal{T}$ and β the substring such that $\alpha = \mathcal{L}(y)\beta$. In this case, $pos_{\mathcal{T}}(\alpha)$ is the triple $\langle y, x, \beta\rangle$.
     - Intuition: The position of α în $\mathcal{T}$ is between nodes y and x of $\mathcal{T}$.

Example

String positions in the suffix tree of string $S = \texttt{carcasa}$

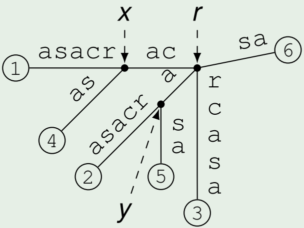


$$pos_T(\lambda) = r$$
$$pos_T(\texttt{c}) = \langle r, x, \texttt{c} \rangle$$
$$pos_T(\texttt{ca}) = x$$
$$pos_T(\texttt{car}) = \langle x, ①, \texttt{r} \rangle$$
$$pos_T(\texttt{carcasa}) = ①$$
$$pos_T(\texttt{arc}) = \langle y, ②, \texttt{rc} \rangle$$
$$pos_T(\texttt{sa}) = ⑥$$

The node depth $d_{\mathcal{T}}(\alpha)$ of substring α of S in the suffix tree $\mathcal{T}$ of S is:

1. if $pos_{\mathcal{T}}(\alpha)$ is a node y, then $d_{\mathcal{T}}(\alpha)$ is the number of nodes from root of $\mathcal{T}$ to y. The root and node y are counted as well.
2. $pos_{\mathcal{T}}(\alpha) = \langle y, x, \beta \rangle$ then $d_{\mathcal{T}}(\alpha)$ is the number of nodes from root of $\mathcal{T}$ to y, except y. The root is counted, but node y is not.

Example



$d_{\mathcal{T}}(\texttt{ca}) = 1$

$d_{\mathcal{T}}(\texttt{carc}) = 2$

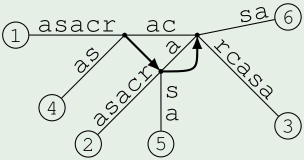
$d_{\mathcal{T}}(\texttt{carcasa}) = 2$

Suffix trees have a remarkable property:

> For every interior node x different from root, there is another interior node y such that $\mathcal{L}(y)$ is obtained from $\mathcal{L}(x)$ by dropping its first character.

y is called the suffix link of x, and is denoted by $suf(x)$.

Example (Suffix links in the suffix tree of `carcasa`)



Main idea: Instead of labeling the edges with substrings $S[i..j]$, we can label them with pairs of integers $\langle i, j\rangle$

$\Rightarrow$ edge labels of variable size (substrings) are replaced by edge labels of constant size (pair of integer indices in S)

Example (Suffix tree for the string `axabxb`)

The suffix tree $\mathcal{T}$ of a string $S[1..n]$ has

- n leaf nodes
- except for the root, every internal node has at least 2 children
- the root node may have 1 child.

Therefore:

- $\mathcal{T}$ has at most n internal nodes.
- $\mathcal{T}$ has at most $2 \cdot n$ edges

$\Rightarrow$ the size of $\mathcal{T}$ is $O(n)$.

Fact: The suffix tree and suffix links of a text $S[1..n]$ can be constructed in time $O(n)$

1. Such an algorithm was first described by Wiener, in 1973.
2. A simpler linear-time algorithm was proposed by Ukkonen; it is described in Chapter 6 of the book

   Dan Gusfield, *Algorithms of Strings, trees, and sequences.* Cambridge University Press, 1997.

Generalized suffix trees

What are they?

Let $\mathcal{S} = \{S_1, \ldots, S_p\}$ a set of p non-empty strings.

- We assume w.l.o.g. that every string S_j ends with a specific character z_j which occurs nowhere else.

The generalized suffix tree of $\mathcal{S}$ is a tree with the following properties:

1. It has $|S_1| + \ldots + |S_p|$ leaves, with labels from the set $\{j{:}i \mid 1 \leq j \leq p, 1 \leq i \leq |S_j|\}$
2. All internal nodes, except the root, have ar least 2 children.
3. Every edge is labeled with a nonempty substring of strings from $\mathcal{S}$.
4. Edges from same node to different children are labeled with substrings that start with different characters.
5. $\mathcal{L}(j{:}i) = S_j[i..n_j]$ where $n_j = |S_j|$.

Like for suffix tree, we define a compact representation of generalized suffix trees:

We replace every edge label $S_j[k..\ell]$ with the constant-size label $j{:}\langle k, \ell \rangle$

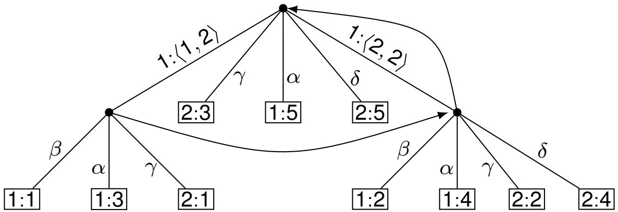
1. We build suffix tree $\mathcal{G}_1$ of S_1 with Ukkonen alg. in $O(|S_1|)$ time
   - we label edges with $1{:}\langle k, \ell\rangle$ instead of $\langle k, \ell\rangle$, and leaves with $1{:}i$ instead of i.
2. For $m := 2$ to p, we build the generalized suffix tree $\mathcal{G}_m$ of set of strings $\{S_1, \ldots, S_m\}$ as follows:
   - Traverse $\mathcal{G}_{m-1}$ from root, to find longest prefix $S_m[1..j]$ which has a position in $\mathcal{G}_{m-1}$.

     $S_m[1..j]$ is longest prefix of S_m which is prefix of a suffix of a string from $\{S_1, \ldots, S_{m-1}\}$
   - Start extending G_{m-1} from that position, until we produce $\mathcal{G}_m$

$\Rightarrow \mathcal{G}_p$ is a suffix tree of $\mathcal{S} = \{S_1, \ldots, S_p\}$, built in $O(n)$ time, where $n = |S_1| + \ldots + |S_p|$

Generalized suffix trees
Example

The generalized suffix tree of $\mathcal{S} = \{\texttt{cocos}, \texttt{comod}\}$ is



where $\alpha = \langle 1, 5, 5\rangle$, $\beta = 1{:}\langle 3, 5\rangle$, $\gamma = 2{:}\langle 3, 5\rangle$, $\delta = 2{:}\langle 5, 5\rangle$.

Given text $S[1..n]$ and pattern $P[1..m]$, find all occurrences of P in S.

1. Construct the suffix tree $\mathcal{T}$ of S in time $O(n)$
2. Find $pos_P(\mathcal{T})$ in time $O(m)$. Suppose $pos_P(\mathcal{T})$ is y or $\langle x, y, \beta \rangle$.
3. Find all leaf nodes of $\mathcal{T}$ below node y.
   - Every occurrence of P in S is a prefix of a suffix $P[j..n]$ of S, where j is the label of such a leaf node.
   - If there are k occurrences of P in S, there are k such leaf nodes. These leaf nodes can be found in $O(k)$ time.

Properties of string matching with (generalized) suffix trees:

1. Finding all occurrences of $P[1..m]$ in a text $S[1..n]$ takes $O(n + m + k)$ time
   - If the suffix tree of S is precomputed, then finding all occurrences of P in S takes $O(m + k)$ time
   - This method is useful if we search often in the same text S (representation of a large database)
2. Finding all occurrences of $P[1..m]$ in all texts of a set $\mathcal{S} = \{S_1, \ldots, S_p\}$ takes $O(n + m + k)$ time where $n = |S_1| + \ldots + |S_p|$

Given two texts S_1 and S_2,

Find the longest substrings common to S_1 and S_2.

Answer:

1. Build the generalized suffix tree $\mathcal{G}$ of $\{S_1, S_2\}$ and mark its internal nodes that have leaf descendants for suffixes of both S_1 and S_2

   Can be done in time $O(n)$ where $n = |S_1| + |S_2|$
2. Traverse the internal nodes of $\mathcal{G}$, and compute the character depth of those which are marked.
   - Note: their character depth is the length of a common substring of S_1 and S_2

Overall computation time: $O(n)$

References

- Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to Algorithms*. Third Edition. Chapter 32. The MIT Press. 2009.
- D. Gusfield: *Algorithms on Strings, Trees, and Sequences*. Published by *Press Syndicate of the University of Cambridge*. 1997.