Lecture 5

Algorithms on strings

Today’s plan

- Notations
- Pattern matching: naive and Knuth-Morris-Pratt algorithms
- Trie
- Aho-Corasick algorithm
- Suffix tree
- Applications of suffix trees

Algorithms on strings: real world

- Bioinformatics

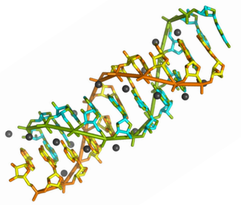
What is the gene responsible for a decease X? Does a genome contain a given gene? How the species developed?…

- Information retrieval

Search engines, plagiarism detection, news analysis, philology, …

- Cyber security

Detection of malicious patterns



DuckDuckGo

Some basic definitions

Alphabet Σ: a finite set

Elements of Σ are called **letters** (or characters, or symbols). We denote letters by small letters ($a, b, c, \ldots$). In practice, letters can be letters of natural languages, words, numbers, …

String (or word) = a finite sequence of letters. We denote strings by capital letters: $P, S, T, \ldots$

Some basic definitions

Length of a string $S = s_1 s_2 \ldots s_n$, where $s_i \in \Sigma$ are letters, is denoted by $|S|$ and defined to be equal to n .

For $1 \leq i \leq j \leq |S|$, we say that $S[i, j] = s_i s_{i+1} \ldots s_j$ is a **substring** of S.

If $i = 1$, we say that the substring $S[i, j]$ is a **prefix** of S (sometimes denoted by $S[\,..\,i]$). If $j = |S|$, we say that $S[i, j]$ is a **suffix** of S (sometimes denoted by $S[i\,..\,]$).

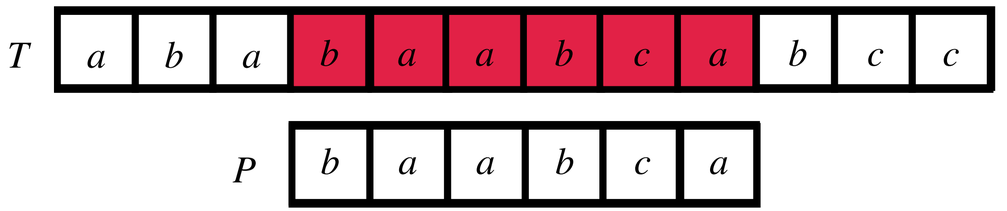
We say that $S[i, j]$ is an **occurrence** of a string X in S if $S[i, j] = X$.

Knuth-Morris-Pratt algorithm

An illustration of Christine de Pizan writing in her study, from *The Book of the Queen* (Harley MS 4431, f. 4r) ©The British Library Board

Pattern matching problem

Given: a string P of length m (pattern), a string T of length $n \geq m$ (text)



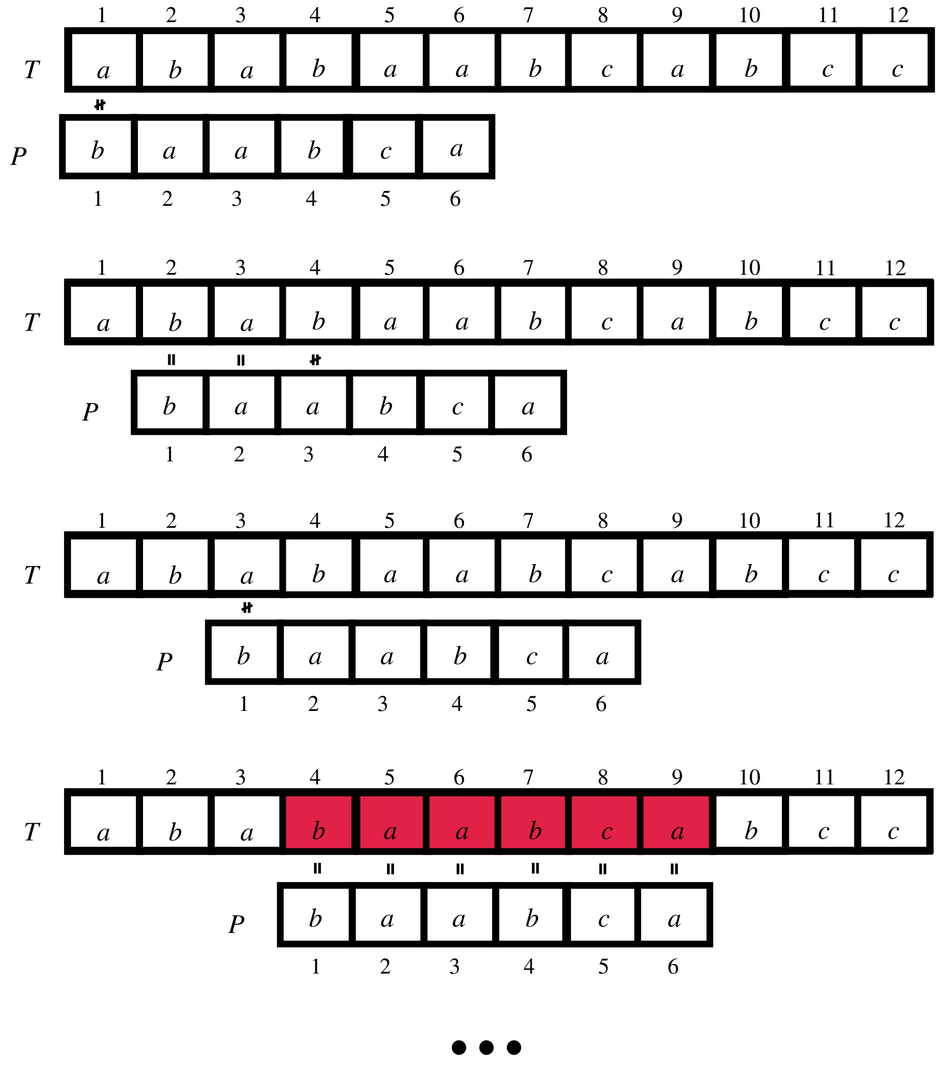
Return: all occurrences of the pattern P in the text T

(specified by their starting positions)

Naive algorithm

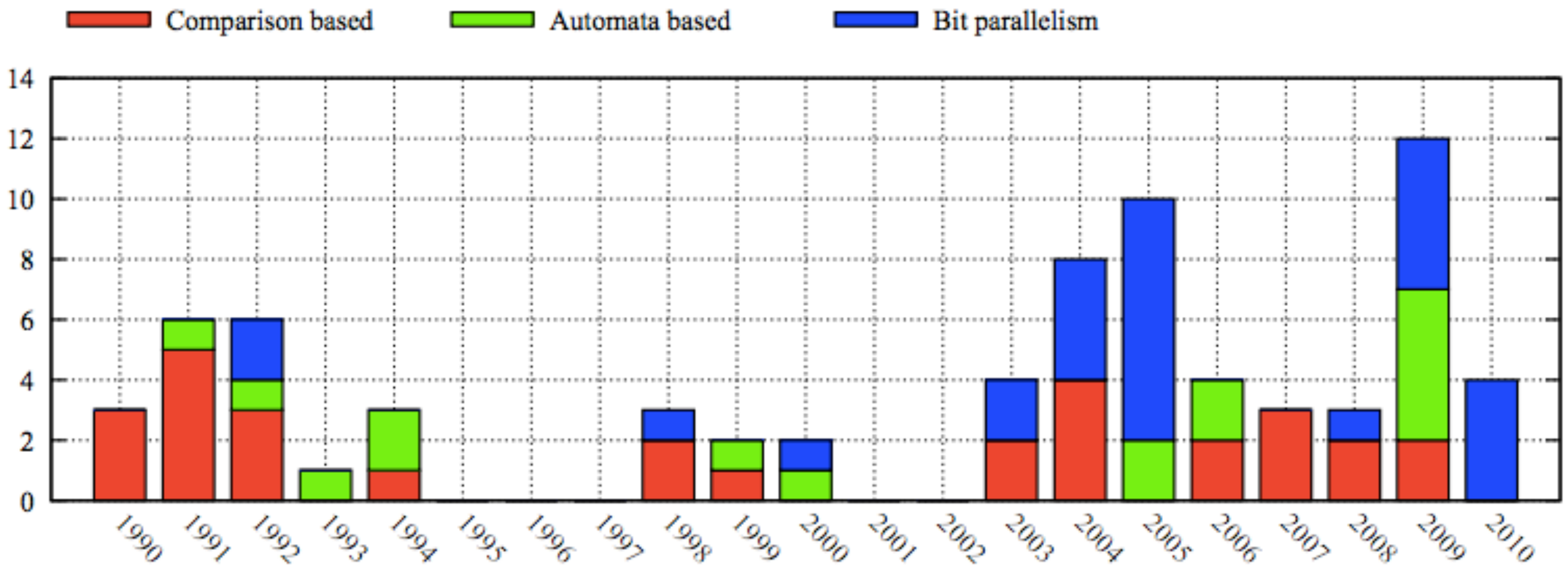
```
for i = 1 to n-m:
        occ = True
        for j = 1 to m:
            if T[i+j-1] != P[j]:
                occ = False
        if occ:
            report i
```

Time = O(nm), Space = O(n+m)



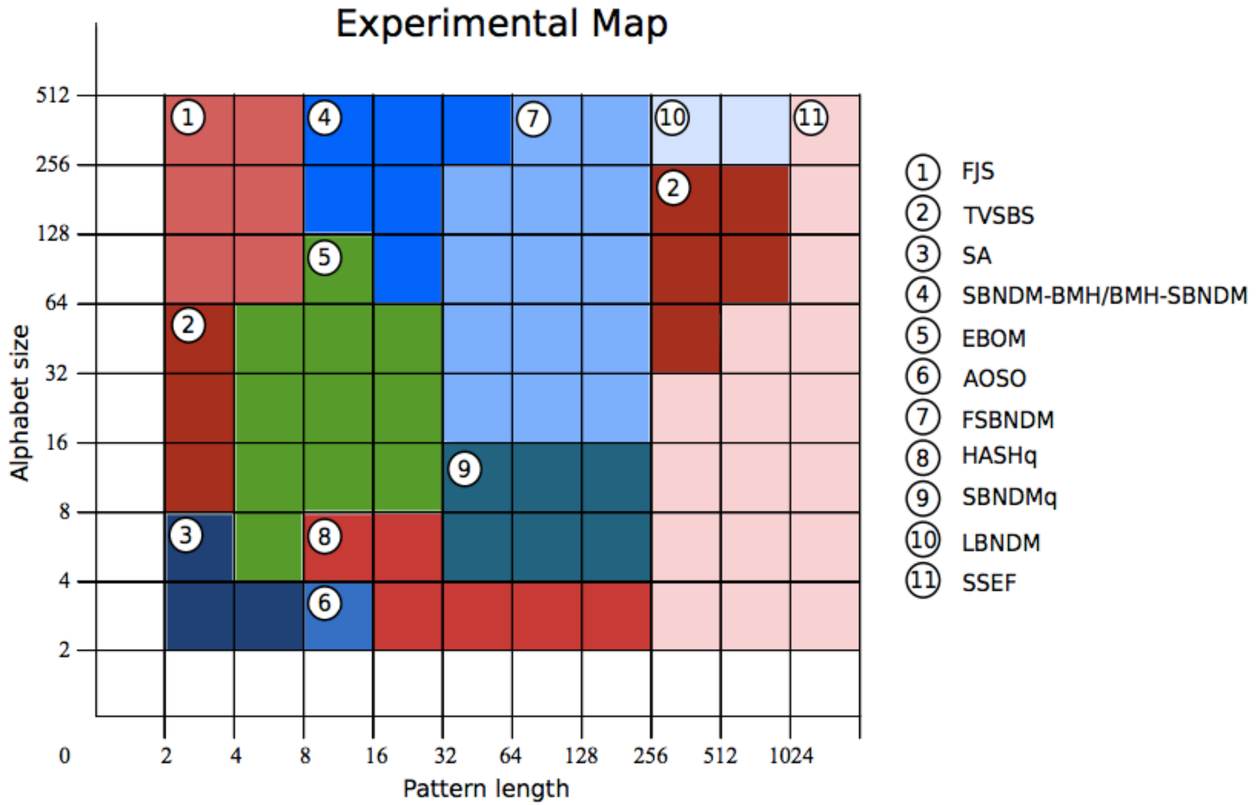
• • •

Pattern matching algorithms



Pattern matching algorithms invented in 1990-2010. Image from S. Faro, T. Lecroq
"The Exact String Matching Problem: a Comprehensive Experimental Evaluation"

Pattern matching algorithms



Experimental comparison of pattern matching algorithms invented in 1990-2010. Image from S. Faro, T. Lecroq "The Exact String Matching Problem: a Comprehensive Experimental Evaluation"

C code: https://www-igm.univ-mlv.fr/~lecroq/string/

Knuth-Morris-Pratt algorithm

Naive algorithm

- Time = O(nm)
- Space = O(n+m) [extra space O(1)]

checks every position of T

Knuth-Morris-Pratt

- Time = O(n+m)
- Space = O(n+m) [extra space O(m)]

tries to skip some positions

Borders and border array

Border of a string P: proper prefix ($\neq P$) of P that is equal to a suffix of P

Example: borders of $P = abcabcabc$ are abc and $abcabc$

Border array of a string P: array B of length $|P|$ such that $B[i]$ is the maximal length of a border of $P[1,i]$ (or zero if undefined)

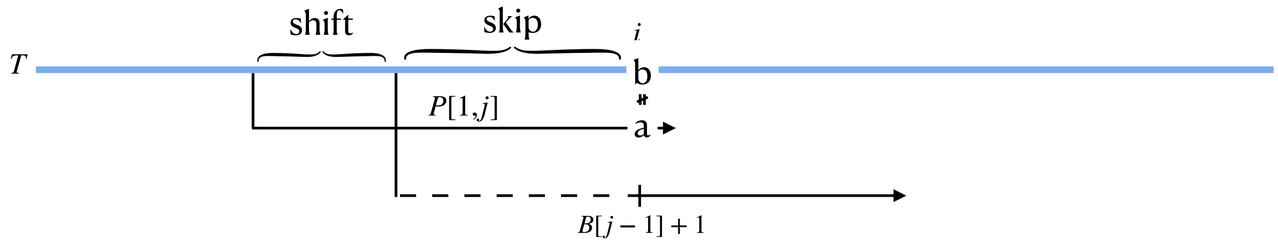
Example: the border array of $P = abcabcabc$ is

0	0	0	1	2	3	4	5	6

Lemma 1: B can be computed in $O(|P|)$ time

Knuth-Morris-Pratt algorithm

shift and skip rule

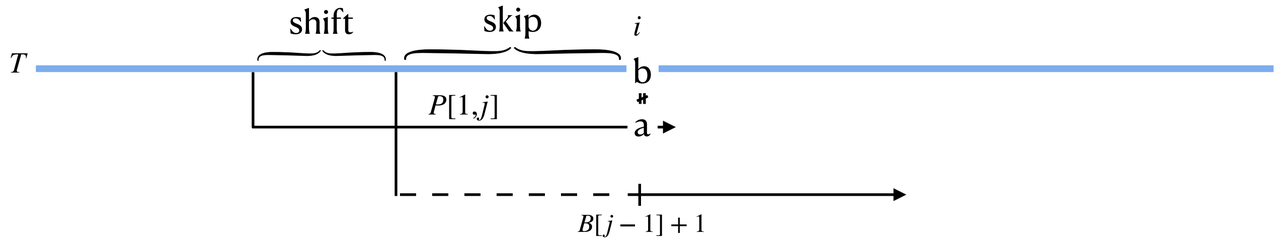


$i \leftarrow 1, j \leftarrow 1$
while $i \leq n - m + j$ **do**
  $(i, j) \leftarrow \text{match}(i, j, m)$
  if $j = m + 1$ **then**
    $(i - m)$ - occurrence //reached the last letter of P
  else
    if $j = 1$ **then**
      $i \leftarrow i + 1$ //mismatch at $P[1]$
    else
      $j \leftarrow B[j - 1] + 1$

$\text{match}(i, j, m)$
  mismatch ← False
  while $j \leq m$ **and not** mismatch **do**
    if $T[i] = P[j]$ **do**
      $j \leftarrow j + 1, i \leftarrow i + 1$
    else
      mismatch ← True

Correctness

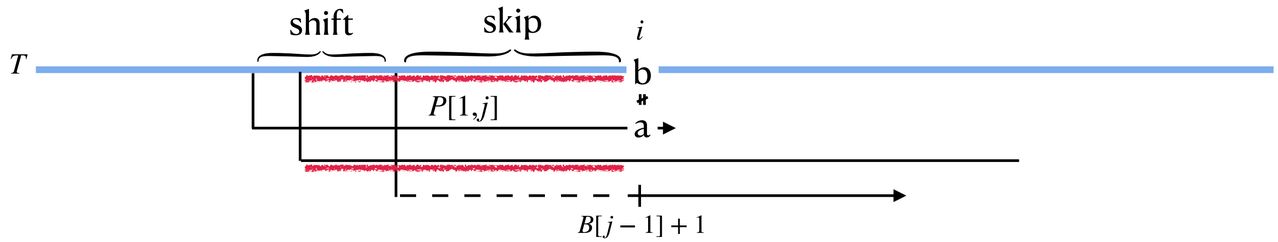
shift and skip rule



Lemma 2: no occurrence of P can start in the shift interval

Correctness

shift and skip rule

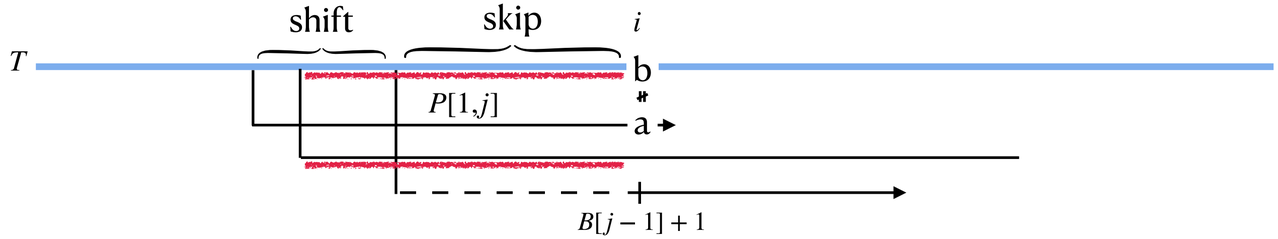


Lemma 2: no occurrence of P can start in the shift interval

Proof: If $T[x, x+m-1] = P$, then $P[1,j-1]$ has a border of length > B[j-1]. Contradiction.

Correctness

shift and skip rule



Lemma 2: no occurrence of P can start in the shift interval

Proof: If $T[x, x+m-1] = P$, then $P[1,j-1]$ has a border of length > B[j-1]. Contradiction.

Observation 1: we have

$P[1,B[j-1]] = T[i+(j-1-B[j-1]), i+j-2]$

Hence, we do not need to compare the first $B[j-1]$ letters of the pattern with the letters of the text after the shift.

Knuth-Morris-Pratt algorithm

Theorem: algorithm finds all occurrences of P in T in $O(m)$ extra space and $O(m+n)$ time.

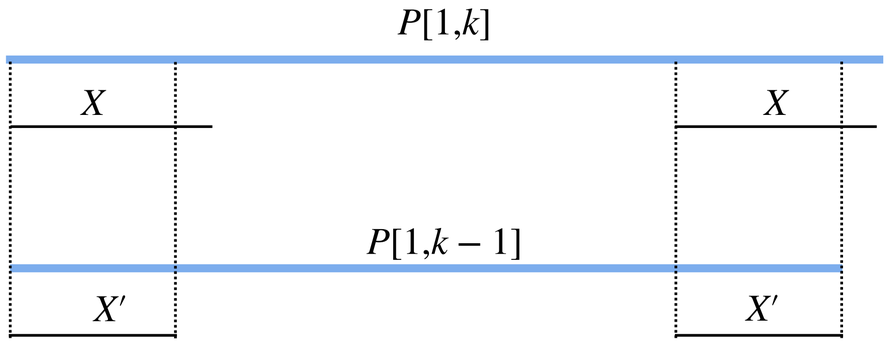
Correctness: Lemma 2 and Observation 1.

Extra space: $O(m)$ to store B + constant

Time: $O(m)$ to build B. To process T, we need $O(n+m)$ time: note that $1 \leq i \leq n$, $1 \leq j \leq n$, and their difference increases after any step (amortised analysis in action once again!)

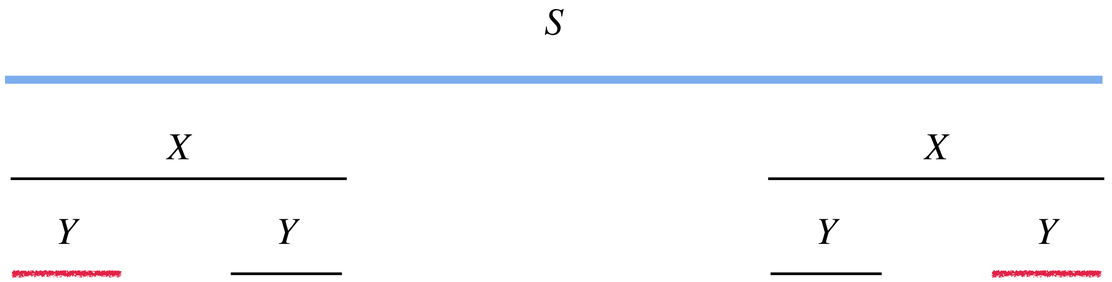
Border array construction (proof of Lm 1)

- $B[1] = 0$
- If X is a border of $P[1,k]$ of length x, then $X' = X[1,x-1]$ is a border of $P[1,k-1]$



Border array construction (proof of Lm 1)

- $B[k] - 1 = B[k-1] \Leftrightarrow P[k] = P[B[k]+1]$
- X is a border of S, and Y is a border of X, then Y is a border of S



Border array construction (proof of Lm 1)

Suppose we have computed $B[1], B[2], \ldots, B[k-1]$. We will now compute $B[k]$.

If $P[k] = P[B[k-1]+1]$, we have $B[k] = B[k-1]+1$ (Property 3)

Let now $P[k] \neq P[B[k-1]+1]$. Consider $B^2[k-1] = B[B[k-1]]$.
If $P[k] = P[B^2[k-1]+1]$, set $B[k] = B^2[k-1]+1$.

Else, consider $B^3[k-1]$, and so on.

Border array construction (proof of Lm 1)

Claim 1: algorithm computes $B[k]$ correctly.

In fact, $B[1,k] \in \{B[k-1]+1, B^2[k-1]+1, \ldots, 0\}$.

From Property 2, any border of $P[1,k]$ can be obtained by appending $P[k]$ to some border of $P[1,k-1]$.

$B^j[k-1]$ is the j-th longest border of $P[1,k-1]$ by Property 4.

Border array construction (proof of Lm 1)

Claim 2: in total, the algorithm uses $O(m)$ time.

Time for computing $B[k] \leq |B[k] - B[k-1]|$ (if $B[k] = B^j[k-1] + 1$, we use j steps and $|B[k] - B[k-1]| \geq j$)

$$-1 \leq \sum_k (B[k] - B[k-1]) \leq m$$

$$-1 \leq \sum^{+} (B[k] - B[k-1]) + \sum^{-} (B[k] - B[k-1]) \leq m$$

amortised analysis

By Property 2, $\sum^{+} (B[k] - B[k-1]) = O(m)$. Hence, $\sum |B[k] - B[k-1]| = O(m)$.

Bonus: real-time version

An algorithm is called **online** if can process its input item-by-item in a serial fashion, without having the entire input available from the start.

An online algorithm is called **real-time** if it processes each data item in constant time.

KMP is an online algorithm, but it is not real-time.

Exercise: Can you give a tight bound on the time KMP uses for processing one letter of the text?

Bonus: real-time version

$$B'[j,a] = \max\{k : k < j, P[1,k] = P[j-k+1,j], P[k+1] = a\}$$

(the maximal length of a border of $P[1,j]$ followed by “a”)

Exercise: B' occupies $O(m)$ space and can be computed in $O(m \cdot |\Sigma|)$ time.

Real-time version of KMP: let i be the current position of the text, and j be the current position of the pattern. If $T[i] \neq P[j]$, use $B'[j, T[i]]$ to decide how to shift the pattern.

Exercise: The algorithm is real-time and correct.

Trie. Dictionary look-up. Aho-Corasick algorithm.

Trie

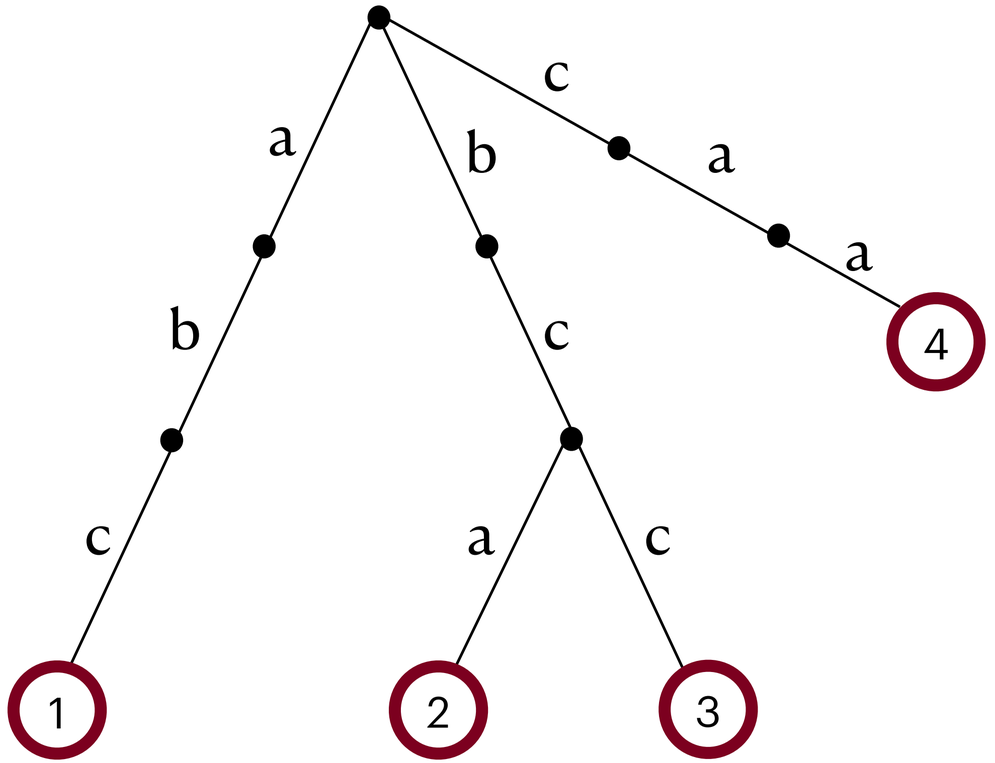
Dictionary = set of strings

Example: $D = \{\overset{1}{abc}, \overset{2}{bca}, \overset{3}{bcc}, \overset{4}{caa}\}$

Trie for D is a tree. Every edge of the trie is labeled with a letter so that:

- For every node, outgoing edges are labeled with different letters.
- For every string $S \in D$, there is a root-to-node path that spells out S. The end of the path is labeled with the id of S.
- Every root-to-leaf path spells out a string from D.

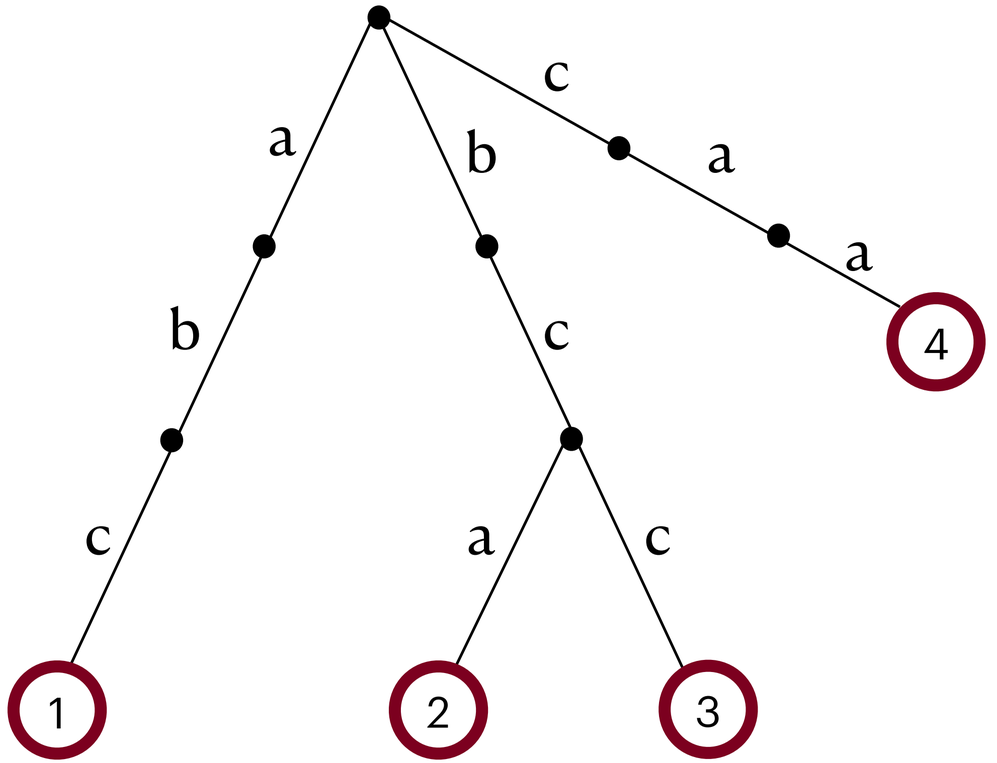
Space = O(total length of strings in D)



Trie

We consider two applications:

1. **Dictionary look-up:** Given a string P, decide if it belongs to the dictionary.
2. **Multiple pattern matching:** Given a dictionary of patterns D, find all their occurrences in a text T.

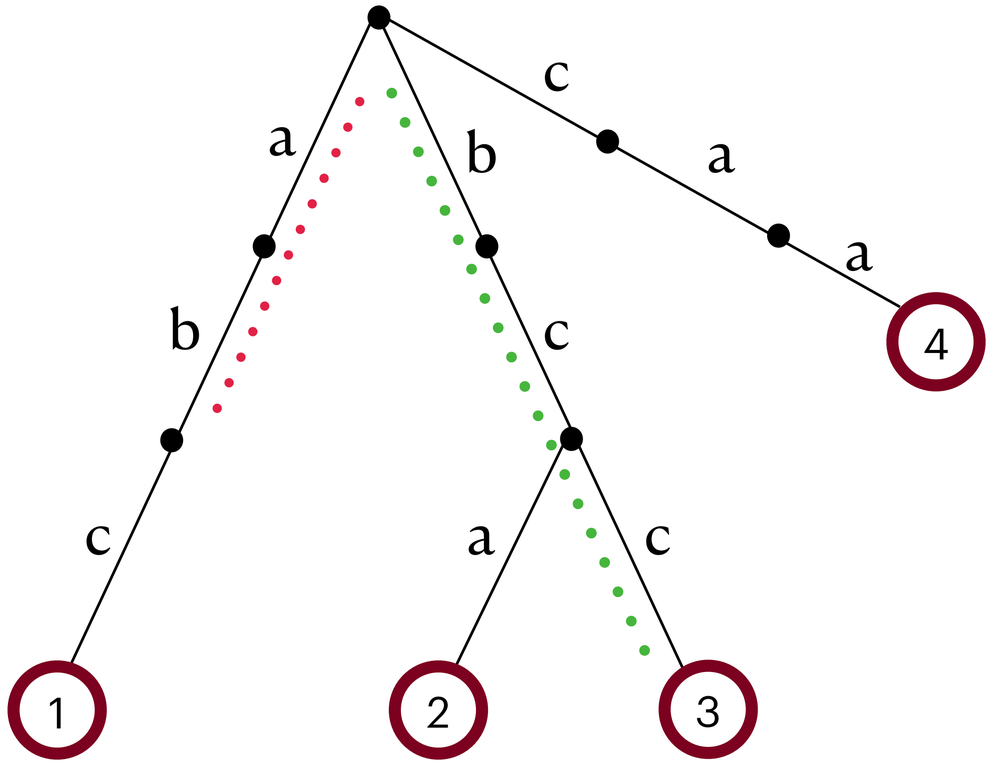


Dictionary look-up

Pattern $P \in D \Leftrightarrow$ there is a root-to-node path that spells out P and its end is labeled by an id of a string from D.

Algorithm starts at the root. If there is an edge from the root to its child u labeled by $P[1]$, the algorithm moves to u and continues recursively.

Time: $O(|P|)$



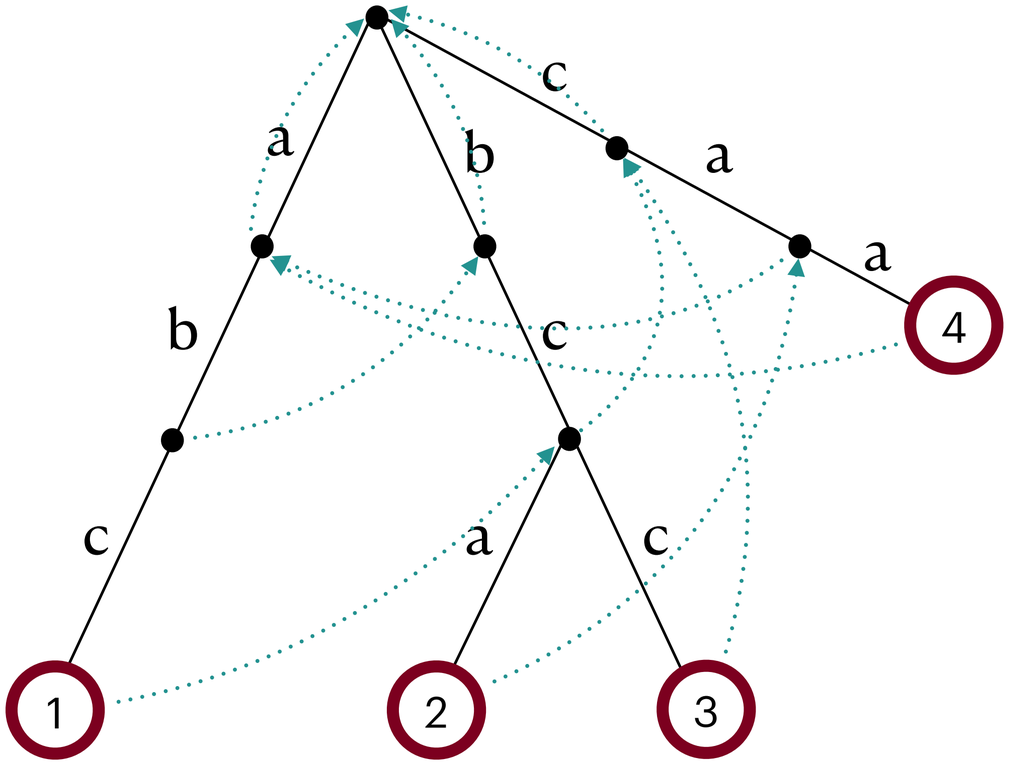
$P_1 = bcc$ $P_2 = abb$

Multiple pattern matching (Aho-Corasick)

First assume that no pattern in D is a substring of another.

The **failure link** from a node u labeled by a string S points to the deepest node v labeled by a proper suffix of S.

Lemma 1: trie occupies $O(m)$ space and can be constructed in $O(m)$ time.

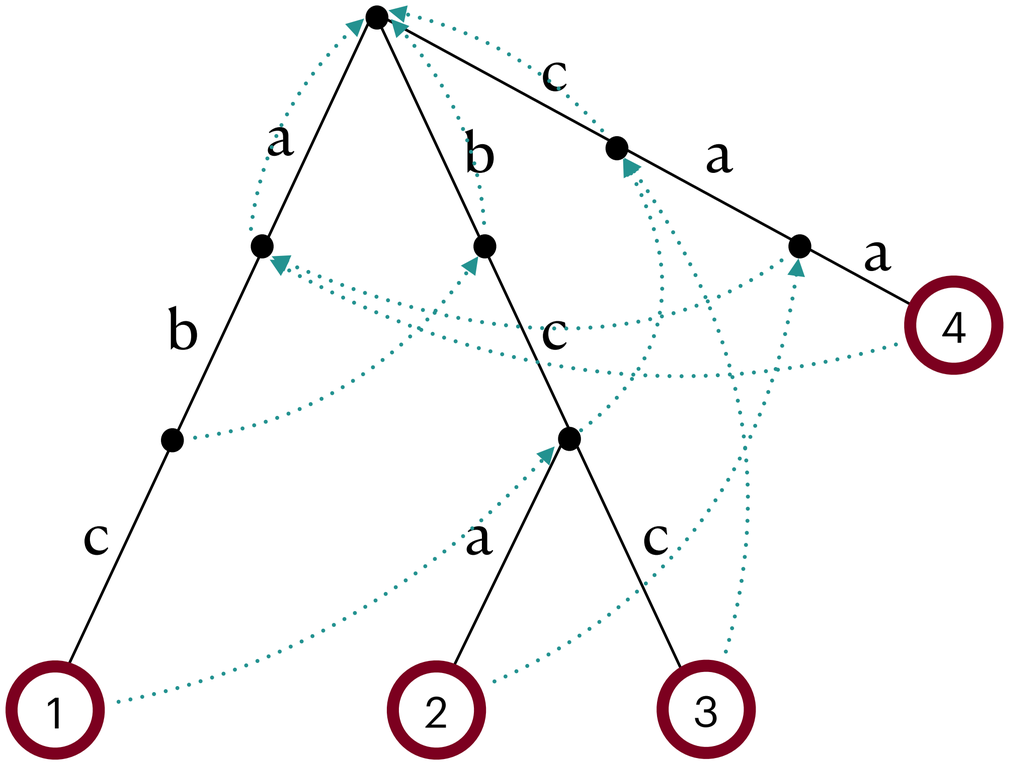


Multiple pattern matching (Aho-Corasick)

First assume that no pattern in D is a prefix of another.

Initialisation: curr_node = root, $i \leftarrow 1$.

```
while i <= n:
    if e = (curr_node, u) labeled with T[i]:
        curr_node = u
        if curr_node corresponds to a pattern:
            report an occurrence
        i = i + 1
    else:
        if curr_node == root:
            i = i + 1
        else:
            curr_node = failure_link(curr_node)
```

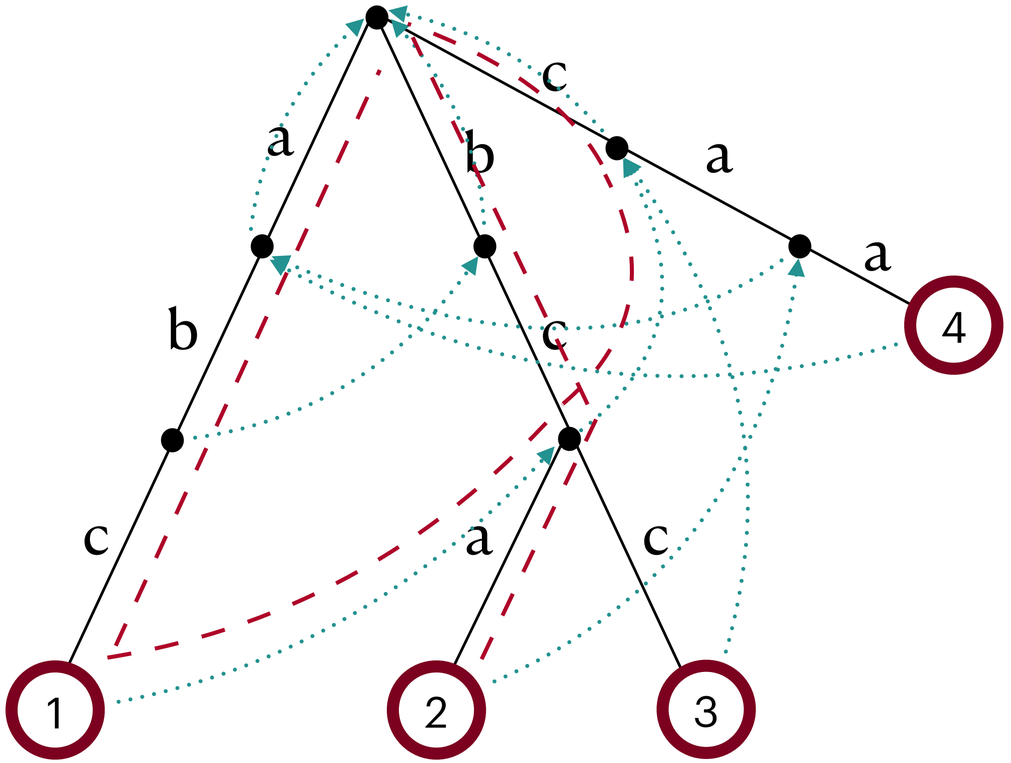


Multiple pattern matching (Aho-Corasick)

First assume that no pattern in D is a prefix of another.

Initialisation: curr_node = root, $i \leftarrow 1$.

```
while i <= n:
    if e = (curr_node, u) labeled with T[i]:
        curr_node = u
        if curr_node corresponds to a pattern:
            report an occurrence
        i = i + 1
    else:
        if curr_node == root:
            i = i + 1
        else:
            curr_node = failure_link(curr_node)
```



T = abcbca

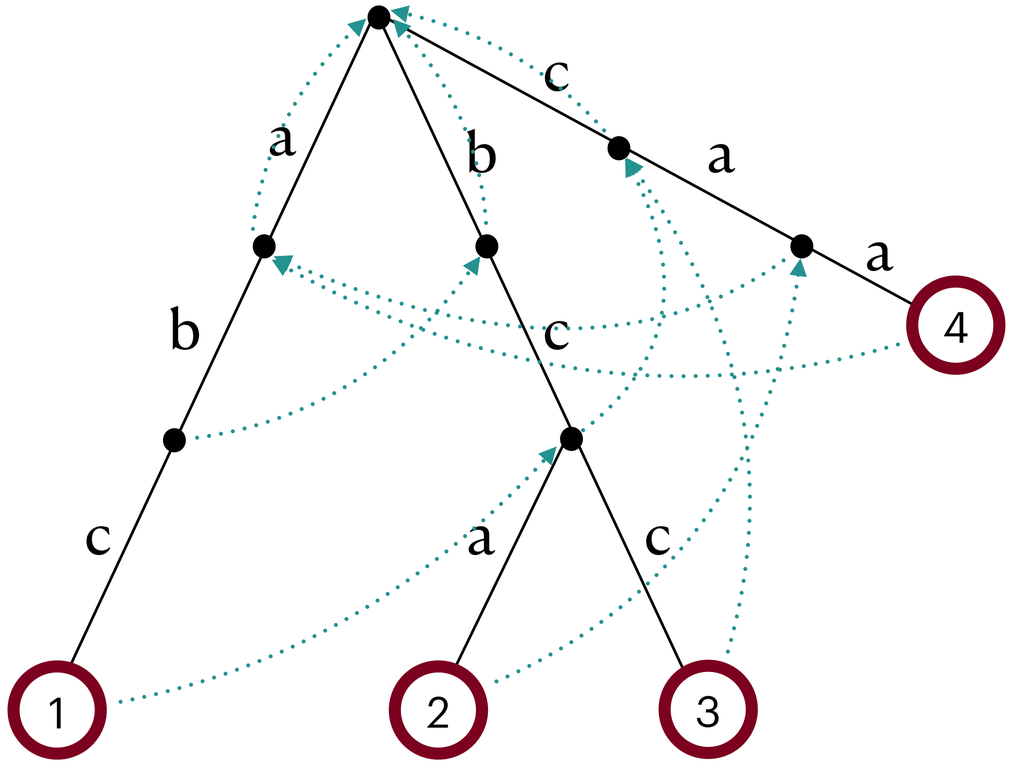
Multiple pattern matching (Aho-Corasick)

Complexity

Space = $O(m)$, where m is the total length of the patterns.

Time = $O(m+n)$: Consider how the depth of curr_node changes during the algorithm.

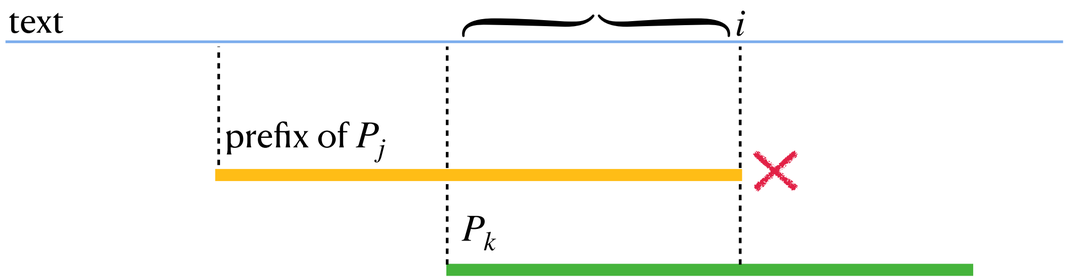
- Every time we go down an edge (happens $\leq n$ times), it increases by 1. Every time we follow a failure link, it decreases by ≥ 1.
- Therefore, as the depth is always positive, we follow a failure link at most n times.

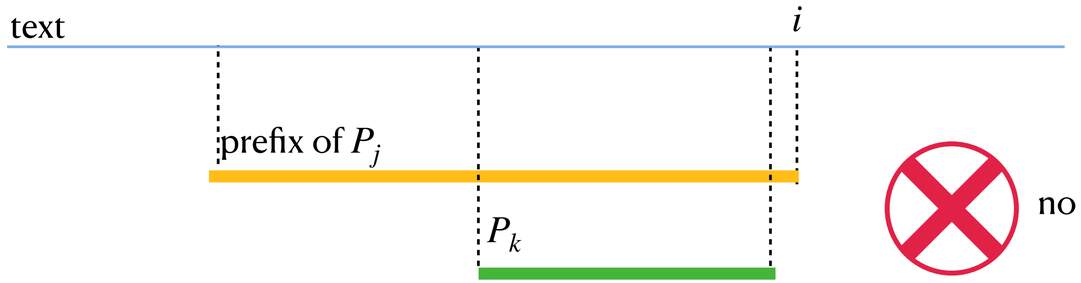


Multiple pattern matching (Aho-Corasick)

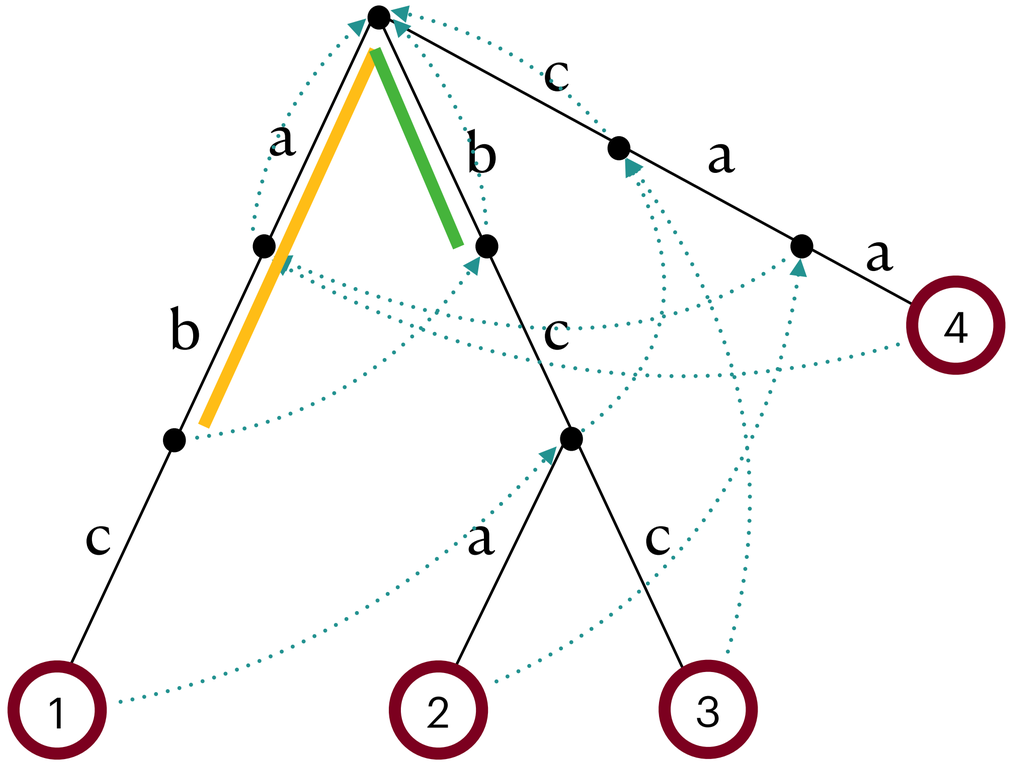
Correctness:

longest suffix of the label of the curr_node that is a prefix of some pattern





no pattern is a substring of another!

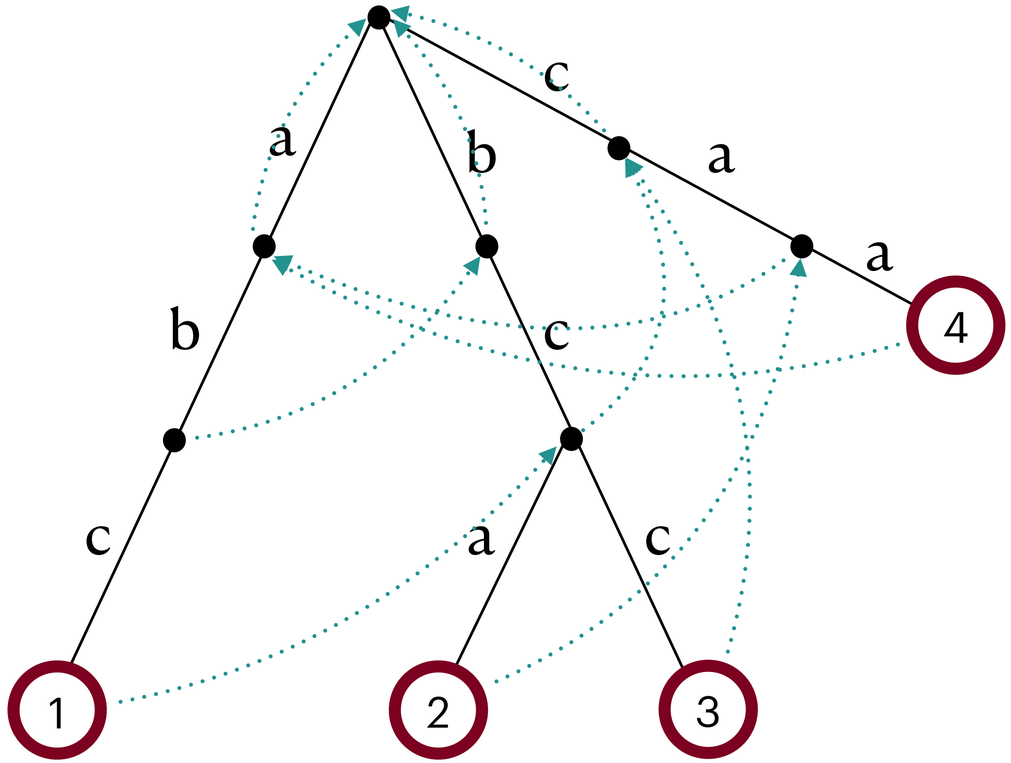


Multiple pattern matching (Aho-Corasick)

Proof of Lemma 1

There are $O(m)$ nodes, each node has exactly one failure link $\Rightarrow$ $O(m)$ space

Failure links are built top-to-down. Links from nodes of depth 1 point to the root.

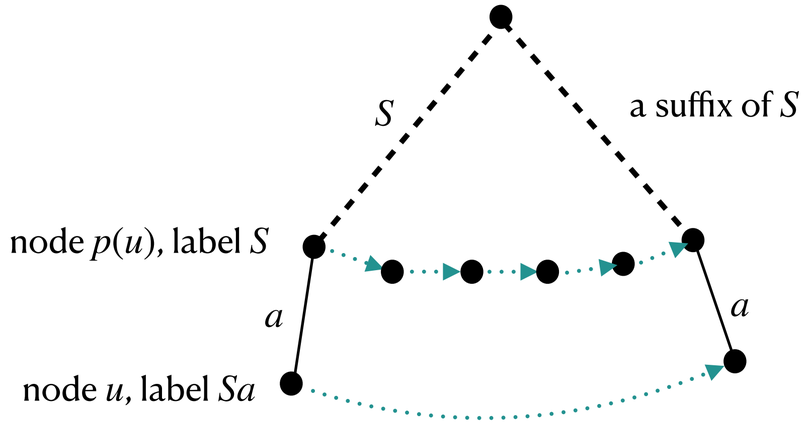


Multiple pattern matching (Aho-Corasick)

Suppose that we have built links for all nodes of depth $\leq d-1$. We now build links for nodes of depth d.

Exercise: the failure link from a node labeled with Sa must point to a node labeled with $S'a$, where S' is a suffix of S.

In other words, the parent of the end of the failure link from the node u must belong to the failure link path from the node $p(u)$, and it must be the deepest node that has an outgoing edge labeled with a.



Algorithm

Follow the failure link path from $p(u)$. The first node that has an outgoing edge labeled with a is the end of the failure link for u.

Multiple pattern matching (Aho-Corasick)

Time complexity

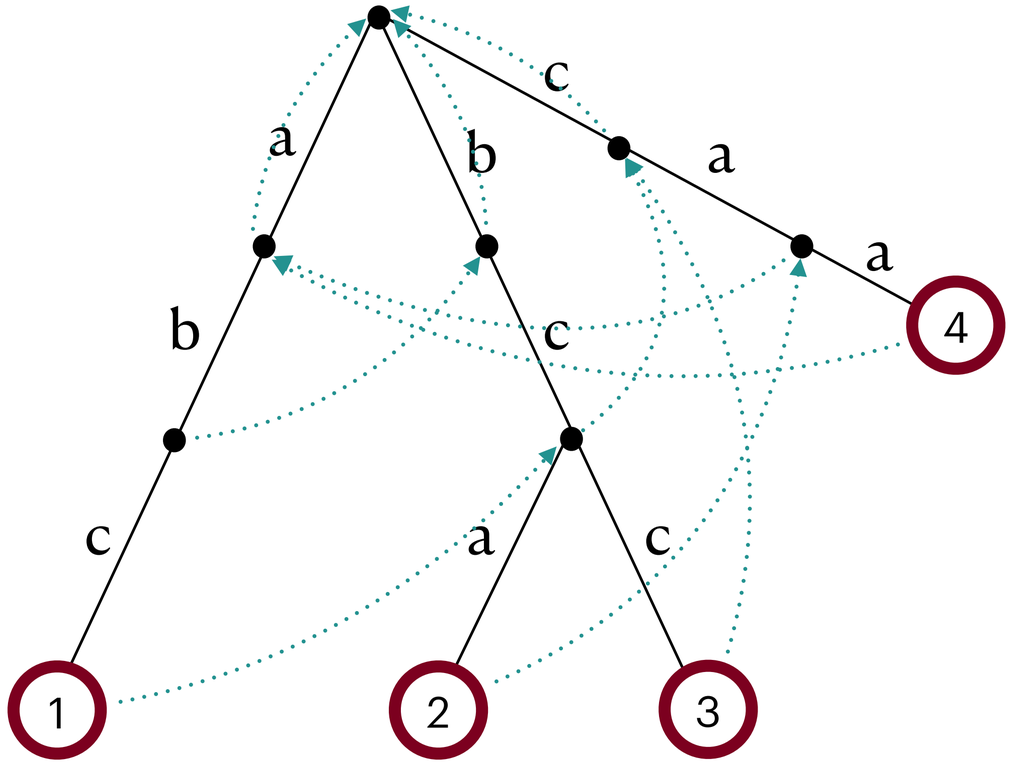
Consider the time needed to build the failure links for one root-to-leaf path.

Let root, $u_1, u_2, \ldots, u_k$ be the nodes in this path, and denote by $f(u_i)$ the depth of the end of the failure link for u_i.

Time to find the link for $u_i \leq c \times (2 + f(u_{i-1}) - f(u_i))$

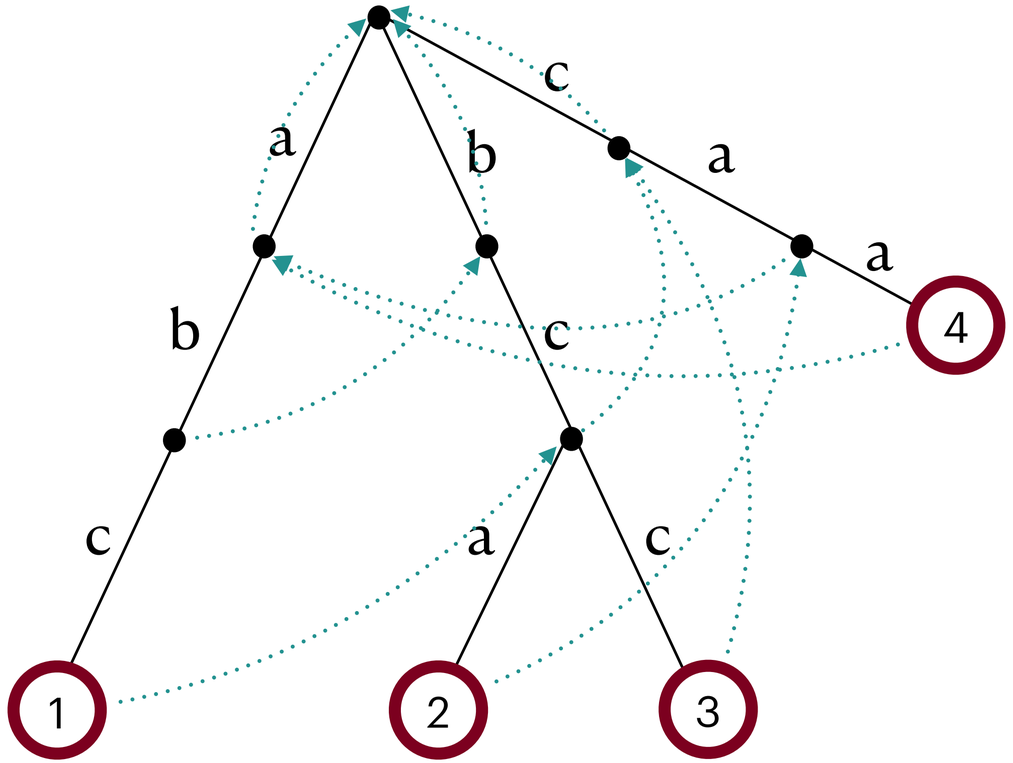
Total time for the path = $O(k)$

Total time for the trie = $O(m)$



Multiple pattern matching (Aho-Corasick)

Theorem: if no pattern is a substring of another, the multiple pattern matching problem can be solved in $O(m)$ space and $O(n + m)$ time.



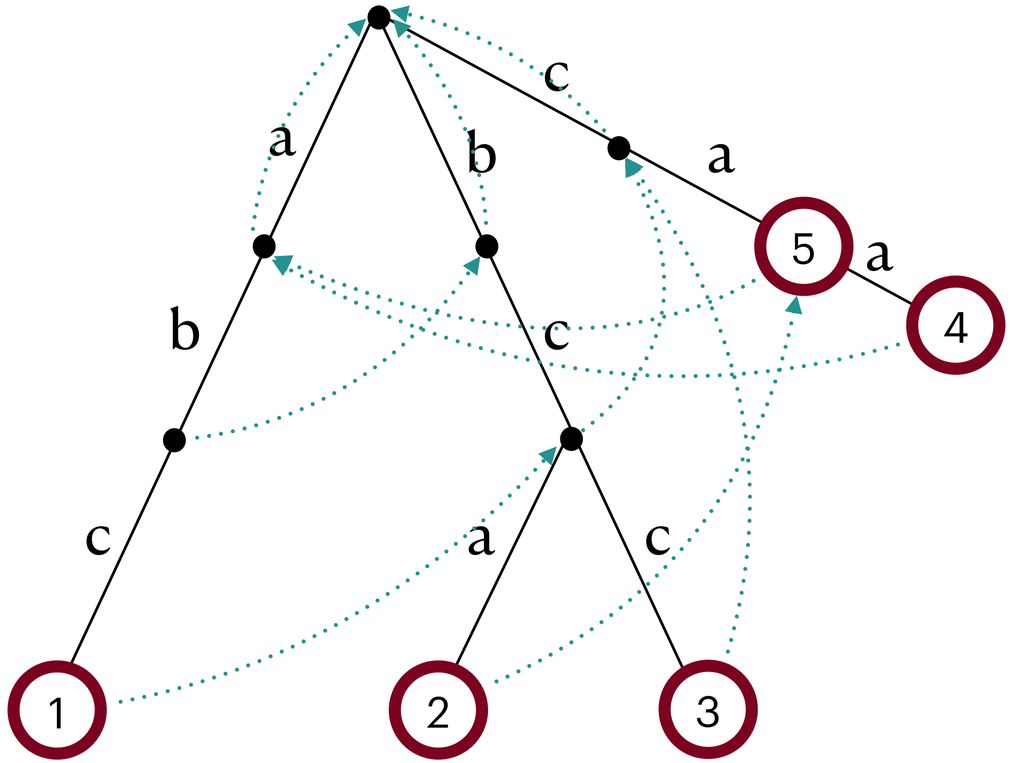
Multiple pattern matching (Aho-Corasick)

What breaks when there is a pattern that is a substring of another pattern?

Example:
$D = \{abc, bca, bcc, caa, ca\}$

T = abcbca

We will miss an occurrence of ca.

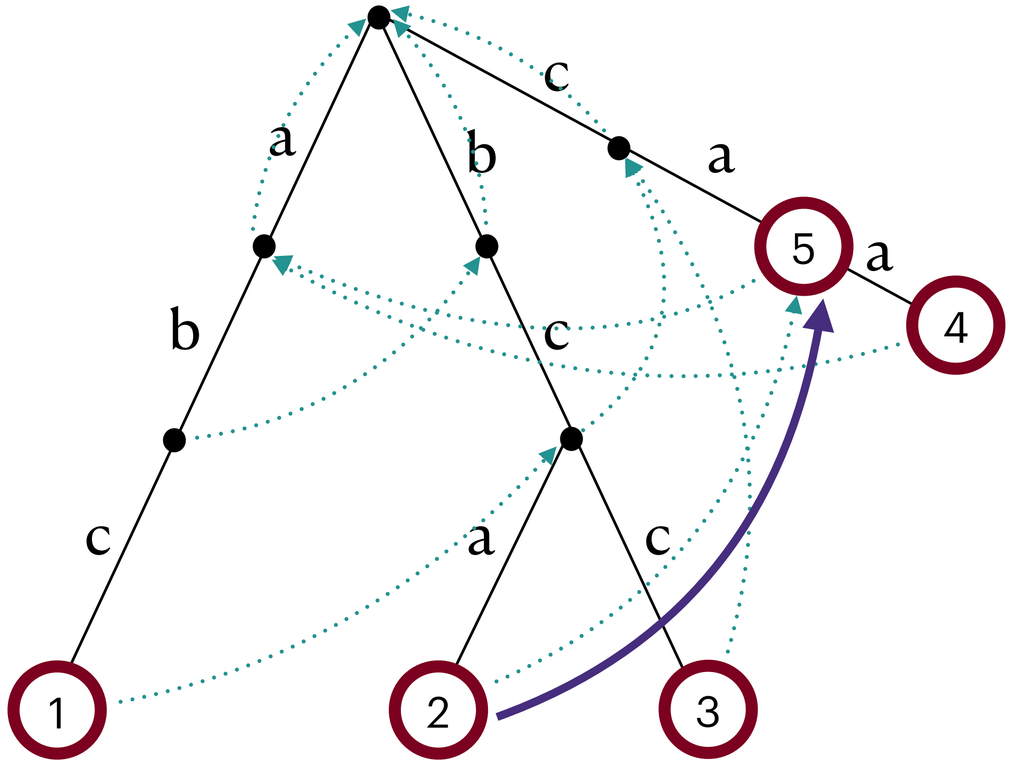


Multiple pattern matching (Aho-Corasick)

To fix that, we add so-called **output links**.

An output link from a node u goes to the nearest node on the failure link path outgoing from u that corresponds to a pattern of the dictionary.

In our example, there is a single output link from node 2 to node 5.

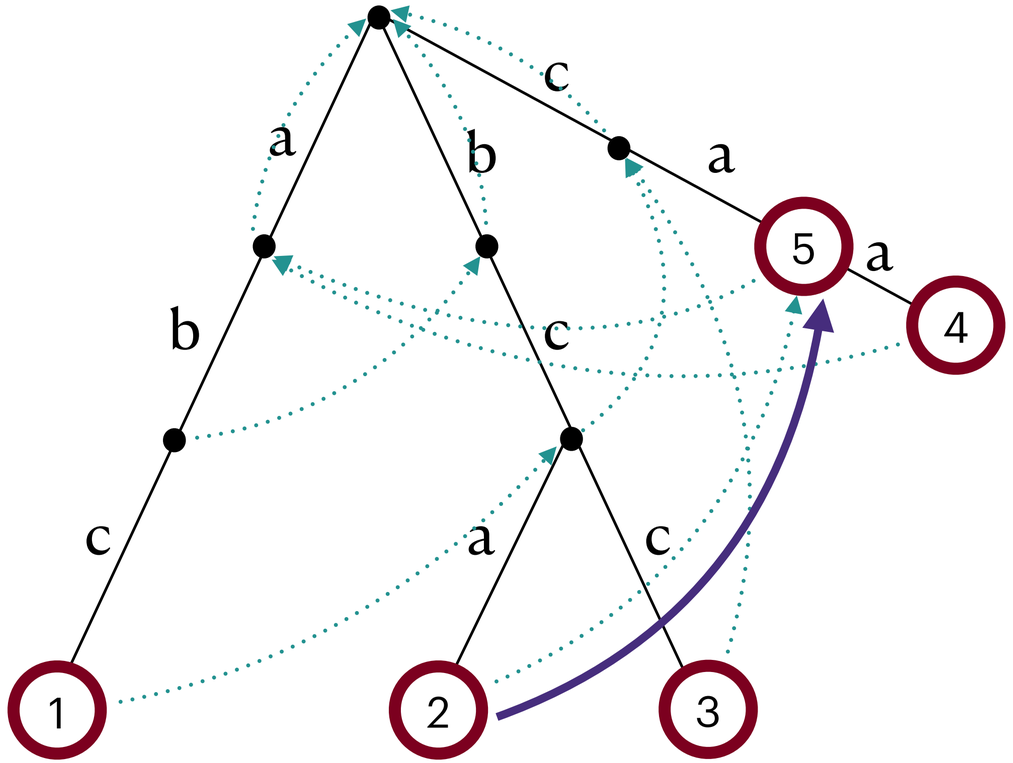


Multiple pattern matching (Aho-Corasick)

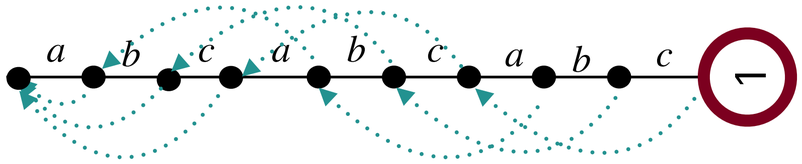
Output links can be built in $O(m)$ time by one top-down traversal of the failure link tree (why is it a tree?)

We modify the algorithm: at each step it follows the path from curr_node and outputs the patterns in the output link path.

Theorem: the multiple pattern matching problem can be solved in $O(m)$ space and $O(n + m + occ)$ time, where occ is the total number of occurrences of the patterns in the text.

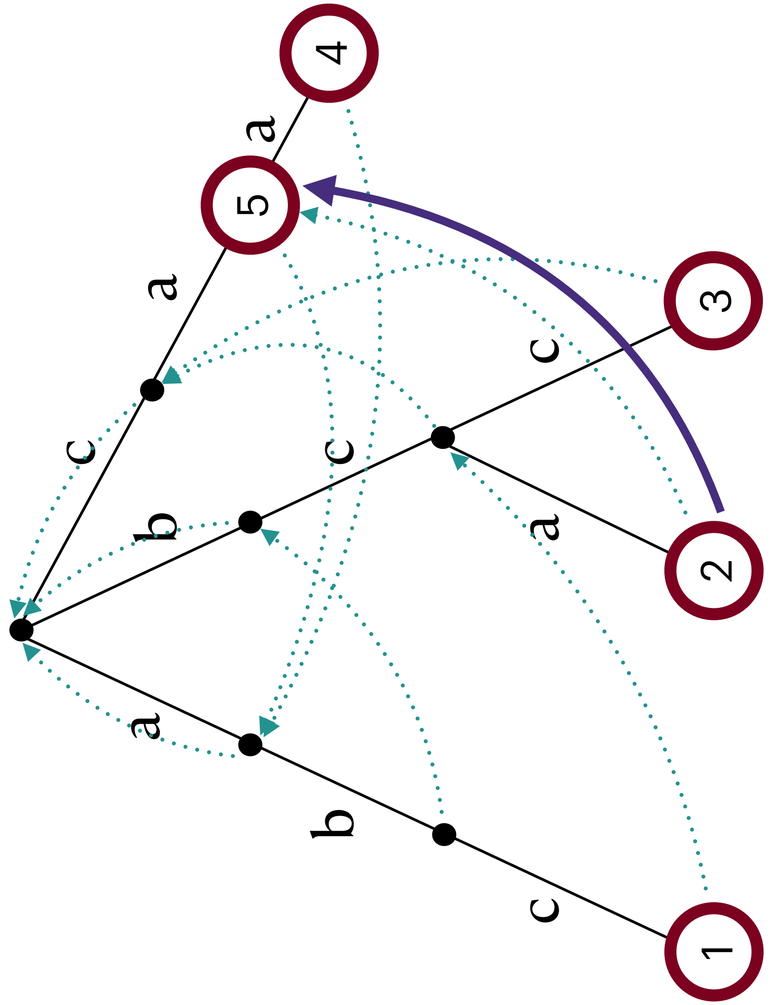


KMP and Aho-Corasick: what's in common?



0	0	0	1	2	3	4	5	6

$$P = abcabcabc$$



In fact, one is the direct generalisation of the other. Both algorithms can be viewed as a traversal of an automaton.

Suffix tree

Text indexes

data structure that represents a text and supports pattern matching queries

- **suffix tree**
- suffix array
- compressed suffix tree
- compressed suffix array
- FM-index
- grammar-compressed indexes
- ...

Suffix tree

LINEAR PATTERN MATCHING ALGORITHMS

Peter Weiner

The Rand Corporation, Santa Monica, California*

Abstract

In 1970, Knuth, Pratt, and Morris [1] showed how to do basic pattern matching in linear time. Related problems, such as those discussed in [4], have previously been solved by efficient but sub-optimal algorithms. In this paper, we introduce an interesting data structure called a bi-tree. A linear time algorithm for obtaining a compacted version of a bi-tree associated with a given string is presented. With this construction as the basic tool, we indicate how to solve several pattern matching problems, including some from [4], in linear time.

"Algorithm of the year 1973"
-Donald Knuth

I. Introduction

In 1970, Knuth, Morris, and Pratt [1-2] showed how to match a given pattern into another given string in time proportional to the sum of the lengths of the pattern and string. Their algorithm was derived from a result of Cook [3] that the 2-way deterministic pushdown languages are recognizable on a random access machine in time O(n). Since 1970, attention has been given to several related problems in pattern matching [4-6], but the algorithms developed in these investigations usually run in time which is slightly worse than linear, for example O(n log n). It is of considerable interest to either establish that there exists a non-linear lower bound on the run time of all algorithms which solve a given pattern matching problem, or to exhibit an algorithm whose run time is of O(n).

In the following sections, we introduce an interesting data structure, called a bi-tree, and show how an efficient calculation of a bi-tree can be applied to

...efore giving a formal definition of a bi-tree, we review ...sic definitions and terminology concerning t-ary trees (see Knuth [7] for further details.)

A *bi-tree* T over $\Sigma = \{\sigma_1,\ldots,\sigma_t\}$ is a set of nodes N which is either empty or consists of a *root*, $n_0 \in N$, and t ordered, disjoint t-ary trees.

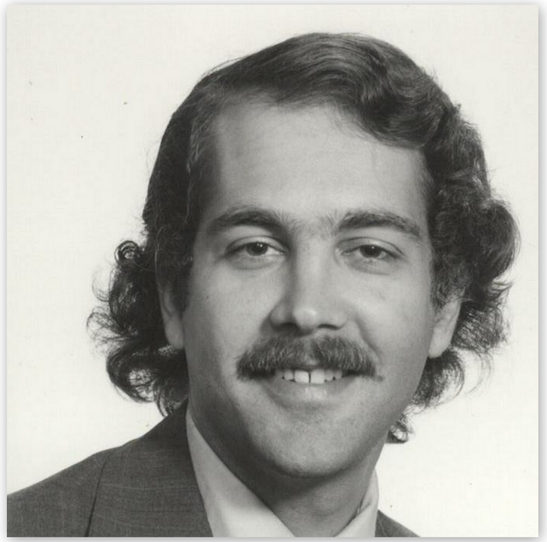
Clearly, every node $n_i \in N$ is the root of some t-ary tree T^i which itself consists of n_i and t ordered, disjoint t-ary trees, $T^i_1, T^i_2, \ldots, T^i_t$. We call the tree T^i_j a *sub-tree* of T^i, and all sub-trees of T^i_j are considered to be sub-trees of T^i. It is natural to associate with a tree T a *successor function*

$$S: N\times\Sigma \to (N-\{n_0\}) \cup \{NIL\}$$

defined for all $n_i \in N$ and $\sigma_j \in \Sigma$ by

n^i_j the root of T^i_j if T^i_j is non-empty

Suffix tree

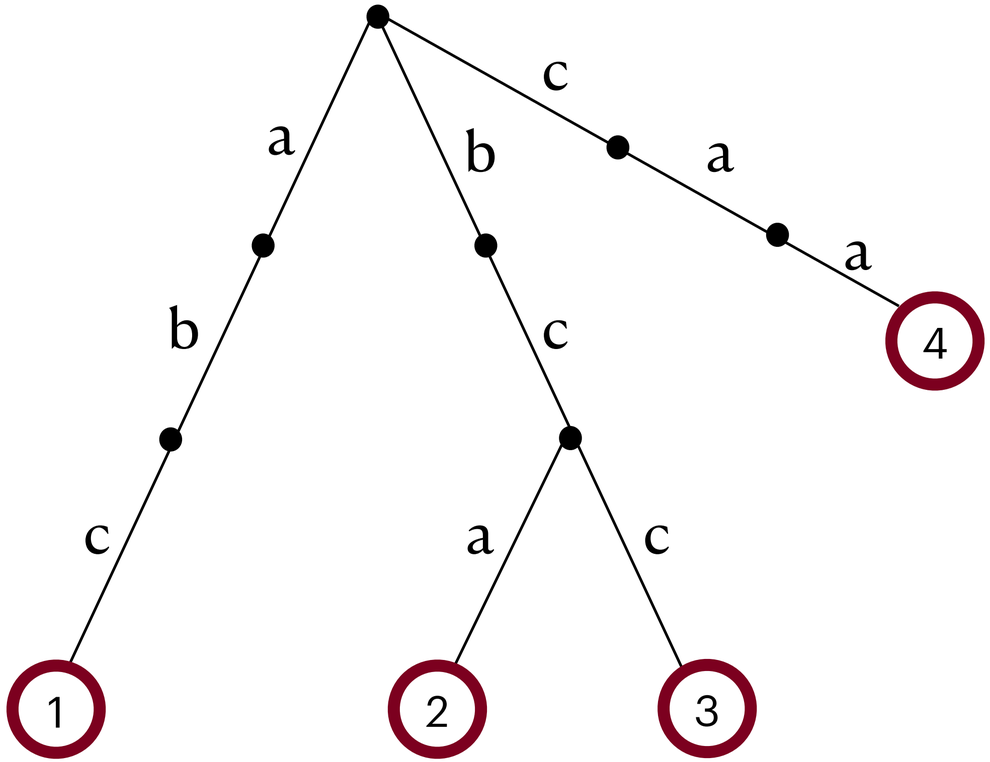


?

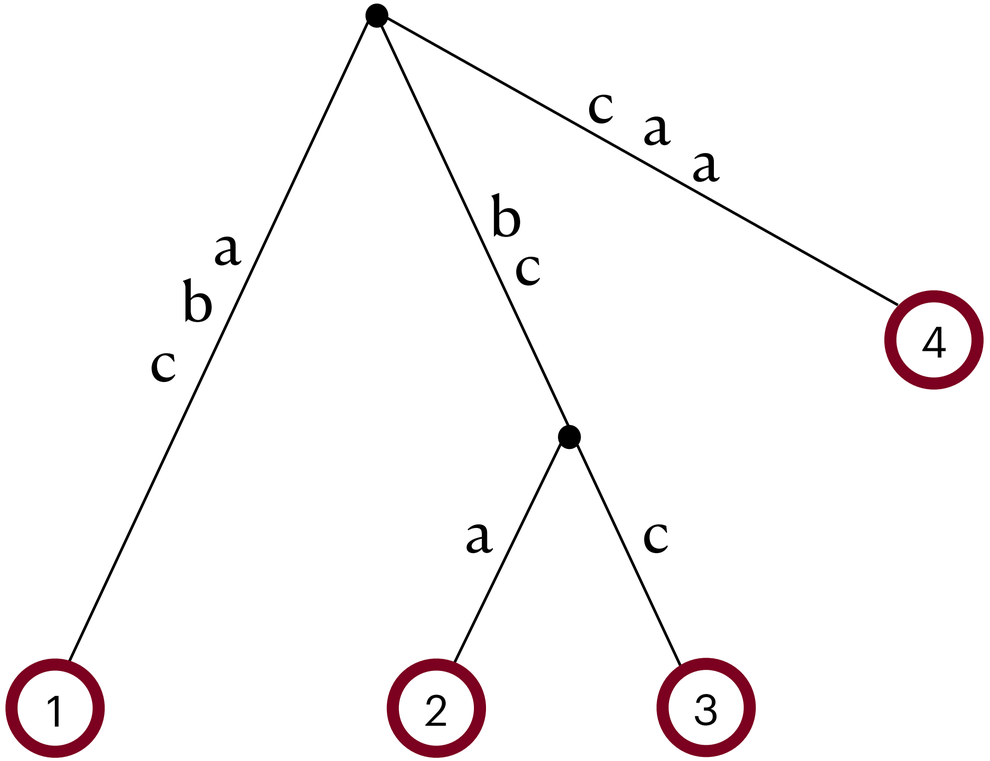
Series of talks dedicated to the 40 years' anniversary of the suffix trees

https://vimeo.com/channels/574784/

Compact trie



trie



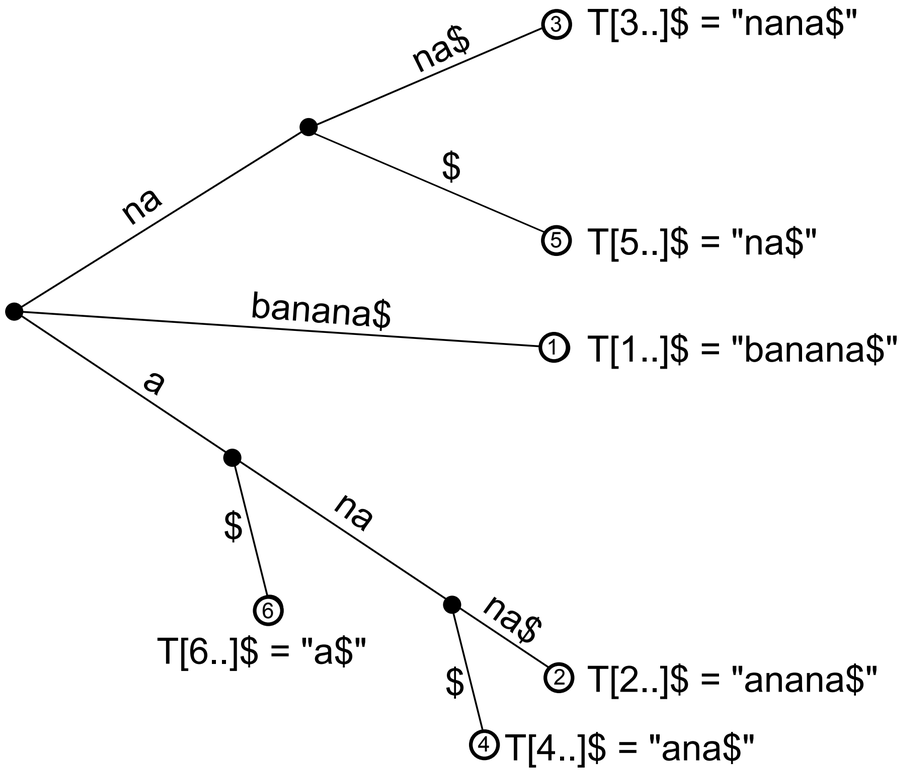
compact trie

Suffix tree

Suffixes of a string T = banana:

T[1,6] = banana
T[2,6] = anana
T[3,6] = nana
T[4,6] = ana
T[5,6] = na
T[6,6] = a

We append $ to each of the suffixes and build the compact trie for them.

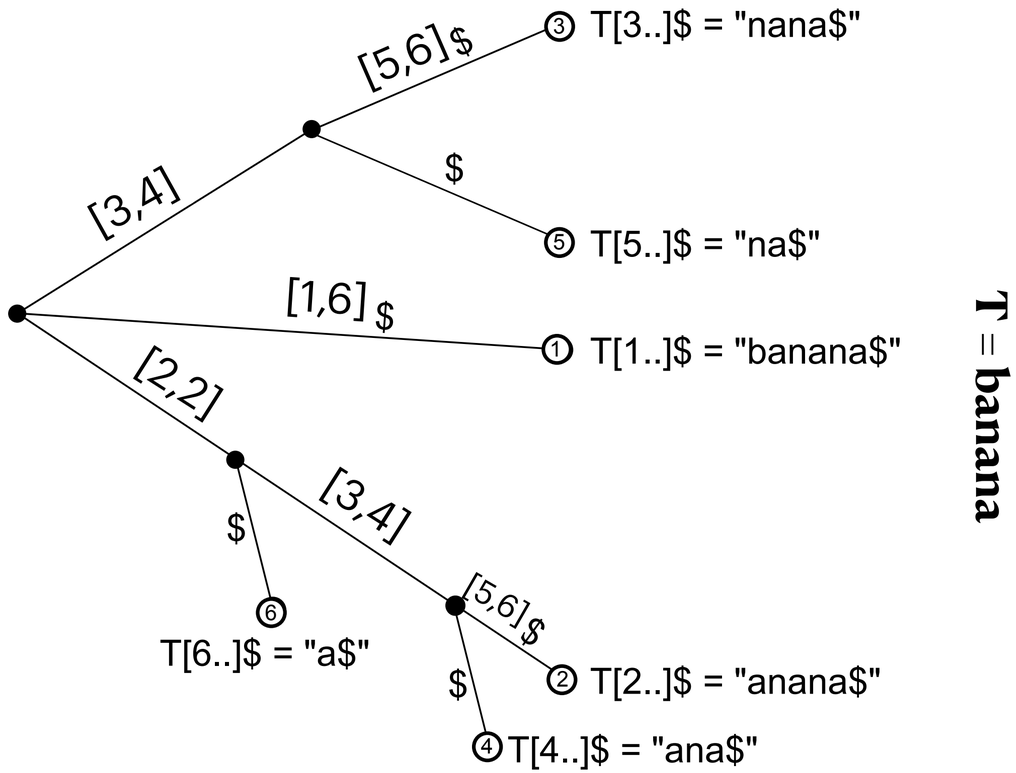


Suffix tree

Storing the labels on the edges can take $\Theta(|T|^2)$ space.

To save the space, we represent each label as two numbers: the left and the right endpoints of the label in T.

Number of leaves: $|T|$
Number of nodes: $\leq 2|T| - 1$
Number of edges: $\leq 2|T| - 2$



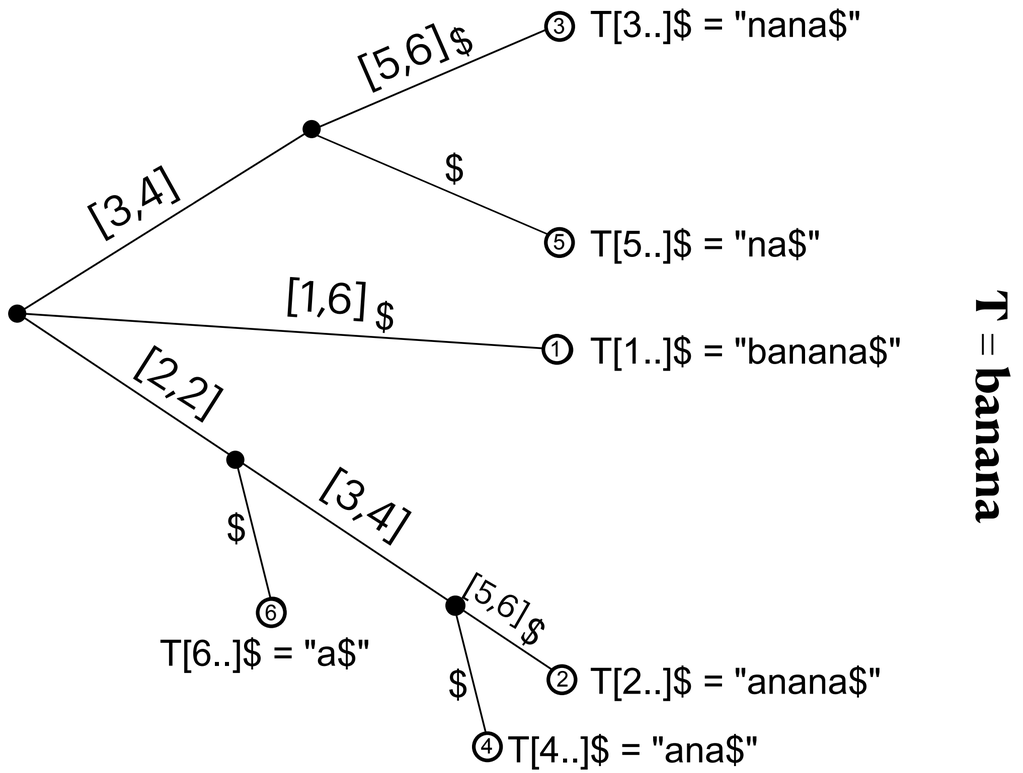
this is the final suffix tree!

Suffix tree

To implement algorithms on the suffix tree efficiently, we need to be able to identify, given a node u and a letter a, an edge (u, v) such that its label starts with a.

For each node u, we store an array A_u of size $|\Sigma|$. In $A_u[a]$, store a pointer to the child v of u such that the label of (u, v) starts with a.

Space: $O(|\Sigma| \cdot |T|)$

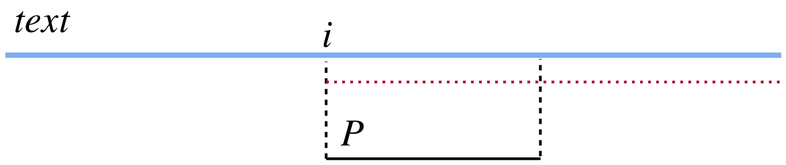


this is the final suffix tree!

Pattern matching

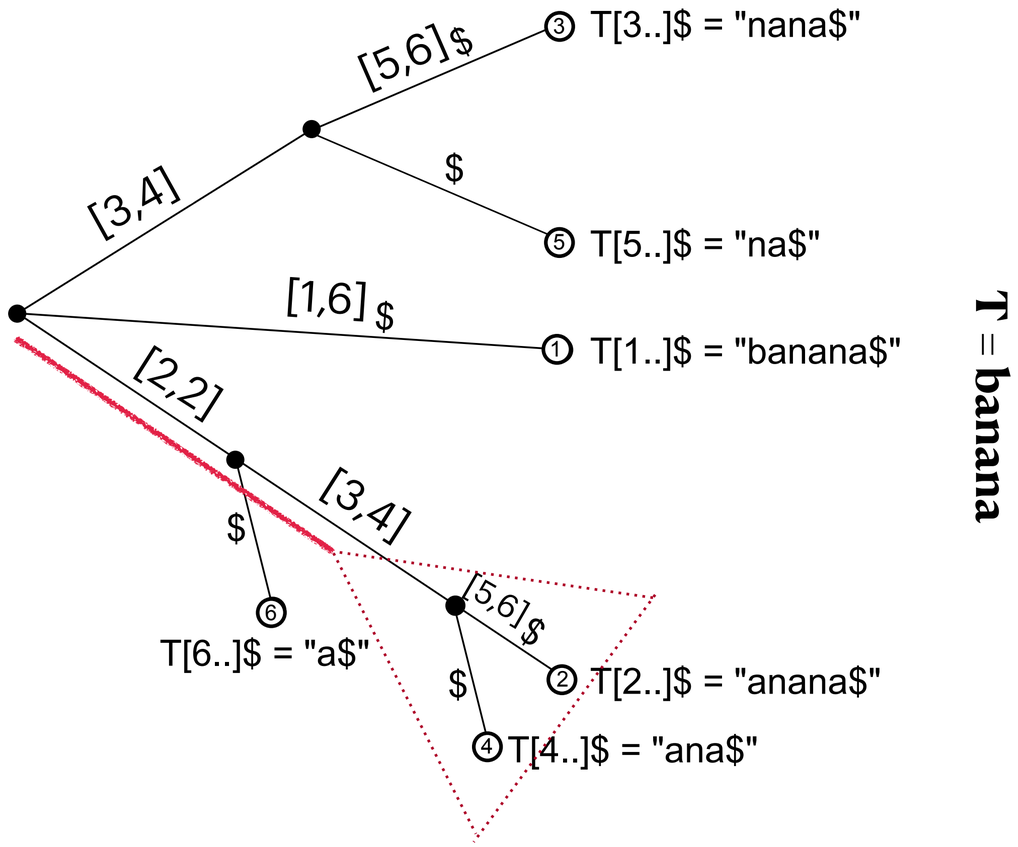
Given a pattern P of length m, find all its occurrences in T.

Observation: if there is an occurrence of the pattern starting at a position i, the suffix $T[i\,..\,]$ starts with P .



Observation: a suffix of T starts with $P \Leftrightarrow$ it corresponds to a leaf of the suffix tree that belongs to a subtree rooted at the end of the path labeled with P.

Example: P = ana



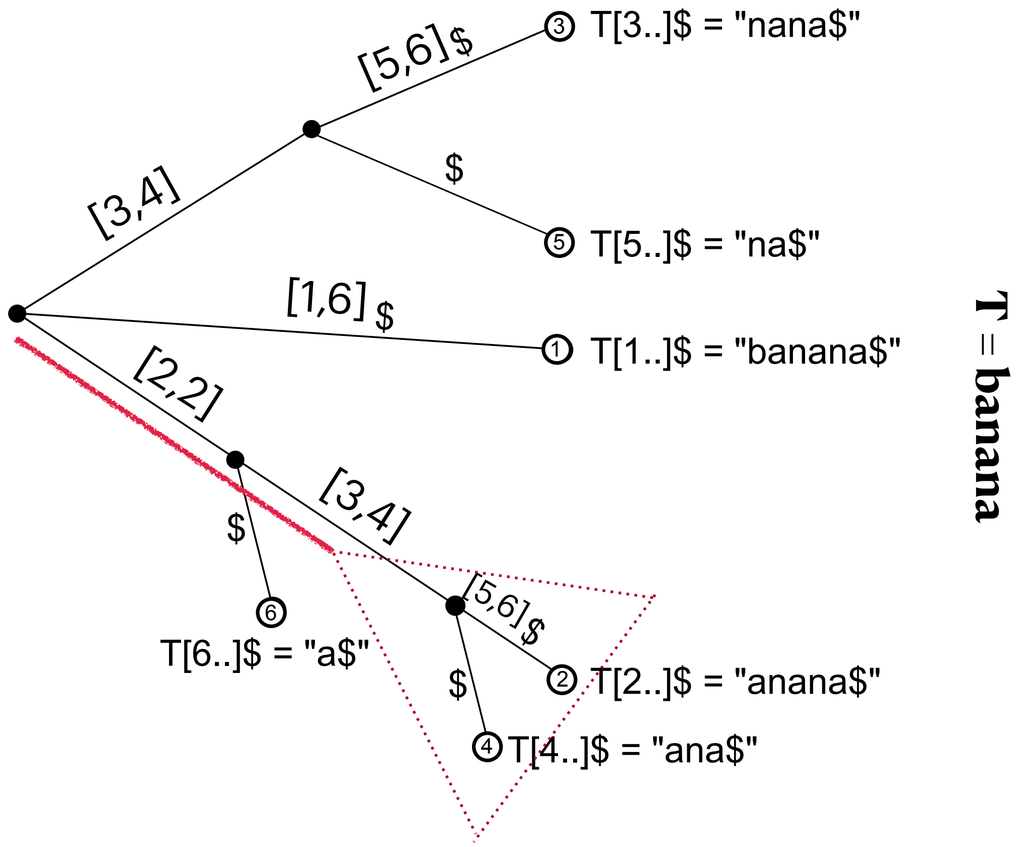
Pattern matching

Algorithm

Start at the root. Follow the path labeled by the letters of P. Use arrays A_u to find the next edge to follow.

If there is no path labeled by P, there are no occurrences of P in T. Otherwise, the starting positions of the occurrences of P in T are the starting positions of the suffixes that are in the subtree rooted at the end of the path labeled by P.

We retrieve the occurrences by traversing the subtree depth-first.



Pattern matching

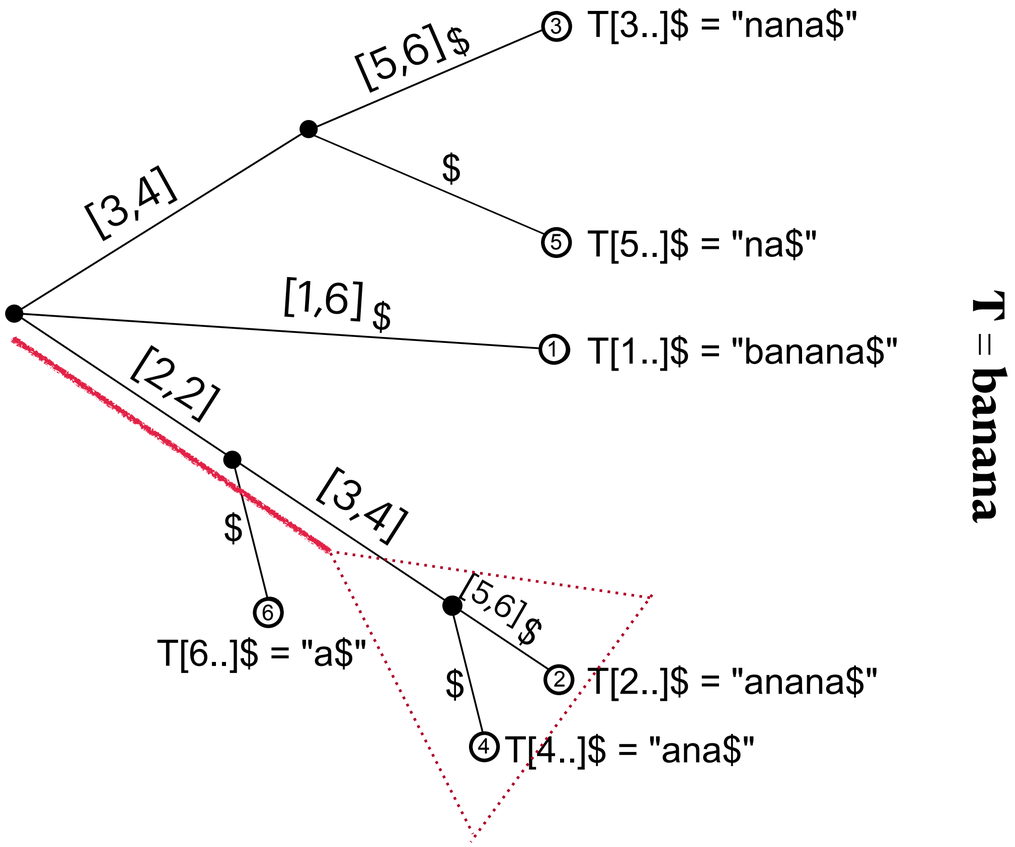
Time complexity

The arrays allow to decide which edge to follow next in $O(1)$ time. In total, finding the path requires $O(m)$ time.

If there is a path labeled by P, the subtree rooted at its end has $O(occ)$ leaves, where occ is the number of occurrences of P in T.

As the subtree does not have nodes of degree one (except, possibly, for the root), its size is $O(occ)$ and its traversal takes $O(occ)$ time.

Total time: $O(m + occ)$



Longest Common Substring

Given two strings S, T, find the longest substring that occurs both in T_1 and T_2.

Example: $T_1 = abcaaabca$, $T_2 = abaaaba$, longest common substring is $aaab$

Knuth believed that this problem requires $\Omega(n \log n)$ time, where $n = |T_1| + |T_2|$, but Weiner proved him wrong!

Applications:

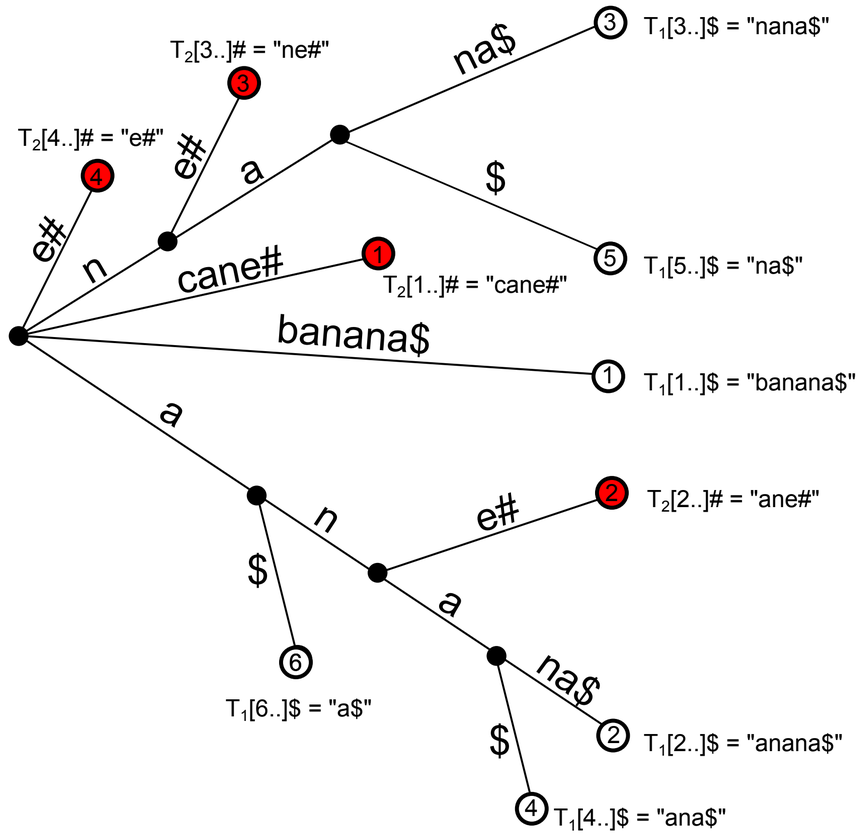
- find substrings common to biological sequences that have important functionalities
- avoid showing similar results in a search engine
- detect fake news

Longest Common Substring

Weiner showed that the suffix tree of one string T can be built in $O(|T|)$ time.

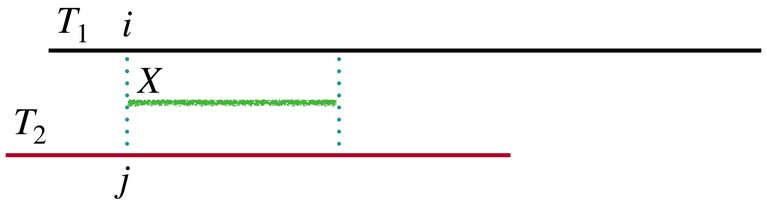
To find the longest common substring of T_1 and T_2, we need the suffix tree containing the suffixes of both strings (sometimes called **generalised suffix tree**).

In TD, we will show that this suffix tree can be built in $O(|T_1| + |T_2|)$ time.



Longest Common Substring

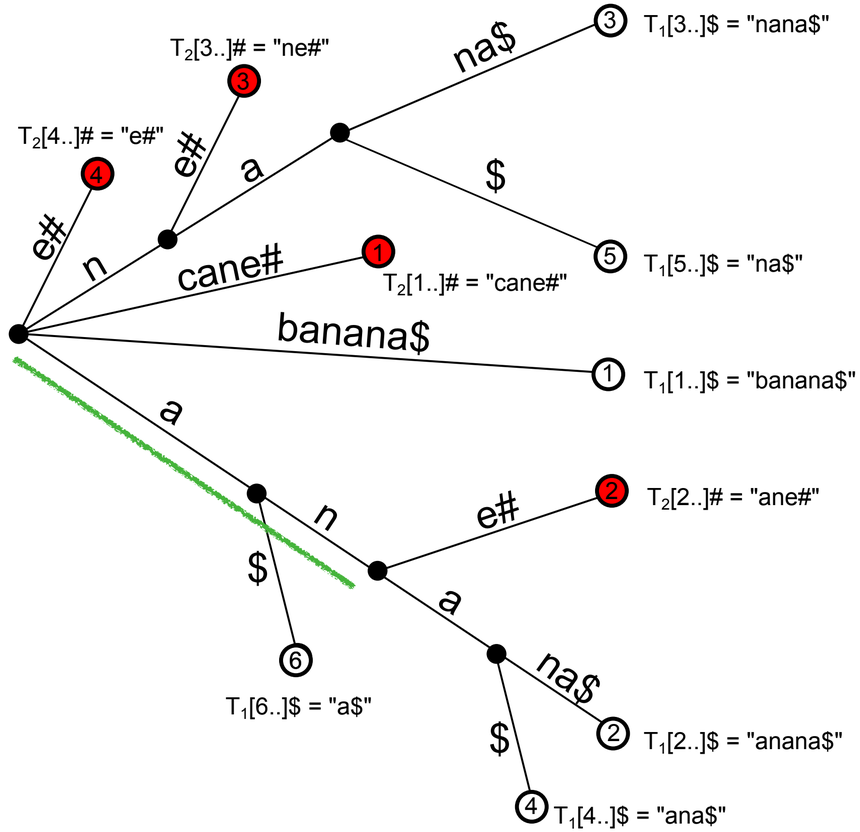
Observation: if a string X occurs at position i in T_1 and at position j in T_2, then the subtree rooted at the end of the path labeled by X contains the leaf corresponding to $T_1[i\,..\,]$ and the leaf corresponding to $T_2[j\,..\,]$.



Algorithm:

By bottom-up traversal, find the deepest node in the tree such that its subtree contains both leaves corresponding to the suffixes of T_1 and leaves corresponding to the suffixes of T_2. It is labeled by the longest common substring of T_1 and T_2.

Time = $O(|T_1| + |T_2|)$



Longest Common Substring

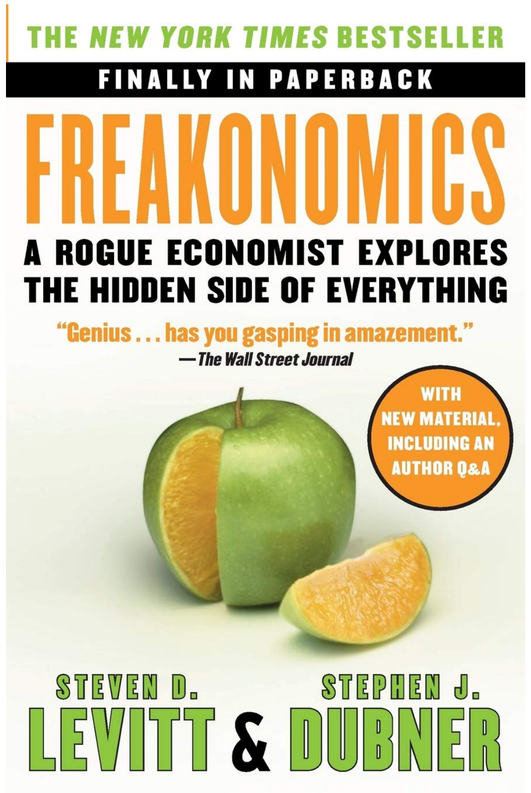
Generalisation: Given an integer d, and strings $T_1, T_2, \ldots, T_m$ of total length n, find the longest substring that occurs in at least d of the strings.

a342cb214d0001acd24a3a12dadbcb4a0000000
41cacbddad3142a2344a2ac23421c00adb4b3cb
4d4ac42d23b141acd24a3a12dadbcb4a2134141
3dcacb1dadbc42ac2cc31012dadbcb4adb40000
3d43232d32323c213c22d2c23234c332db4b300
ad1acbdda212b1acd24a3a12dadbcb400000000
2124cbddadbcb1a42cca3412dadbcb423134bc1
4d4a2b1dadbc3ca22c00000000000000000000
a24acb1d32b412acd24a3a12dadbcb422143bc0
ad1ac3d2a23431223c00012dadbcb400000000
3dcacbd32d313c21142323cc300000000000000
4d1ac3dd43421240d24a3a12dadbcb400000000
a24acb11a3b24cacd12a241cdadbcb4adb4b300
adcacb1dad3141ac212a3a1c3a144ba2db41b43
40c2cbddadb4b1acd24a3a12dadbcb43d133bc4
4dc4cbdd31b1b2213c4ad412dadbcb4adb00000
4d4a23d24131413234123a243a2413a21441343
4d14c3d2ad4cbcac1c003a12dadbcb4adb40000
a21ac3d2ad3c4c4cd40a3a12dadbcb400000000
a2cacbd1a13211a2d02a2412d0dbcb4adb4b3c0
adc4cbddadbcbc2c2cc43a12dadbcb4211ab343
a3cacbddadbcbca42c2a3212dadbcb42344b3cb

Teacher A

31422bd131b4413cd422a1acda332342d3ab4c4
a11acb2d3dbc1ca22c23242c3a142b3adb243c1
2d2a4b1d32b21ca2312a3411d00000000000000
4344c32d21b1123cdc0000000000000000000000
ad12cbdd3d4c1ca112cad2ccd00000000000000
431a2b2d2d44b2acd2cad2c2223b40000000000
2d2a1bd2431141342c13d212d233c34a3b3b000
4d4a1bdd23b242a22c2a1a1cda2b1baa33a0000
23c4cbddadb23c322c2a222223232b443b24bc3
4313c31d42b14c421c42332cd2242b3433a3343
ad122b1da2b11242dc1a3a12100000000000000
ad1a13d23d3cb2a21ccada24d2131b440000000
33c4cbd142141ca424cad34c122413223ba4b40
adcacbddadbc42ac2c2ada2cda341baa3b24321
4dc2cb2dadb24c412c1ada2c3a341ba20000000
431acbddad3c4c213412da22d3d1132a1344b1b
a21a1b2dadb24ca22c1ada2cd32413200000000
3d2a2bddadbcbca11c2a2accda1b2ba20000000

Teacher B

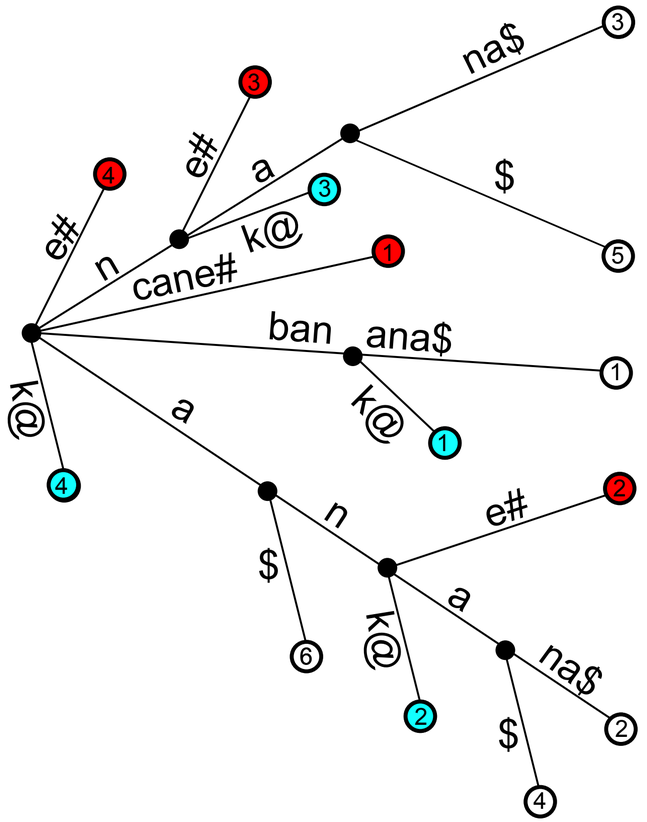


Longest Common Substring

Start by building the generalised suffix tree of $T_1, T_2, \ldots, T_m$ in time $O(n)$.

Analogously to the case $m = 2$, we must find the deepest node such that below it there are leaves corresponding to at least d distinct strings.

Exercise: How would you do that?

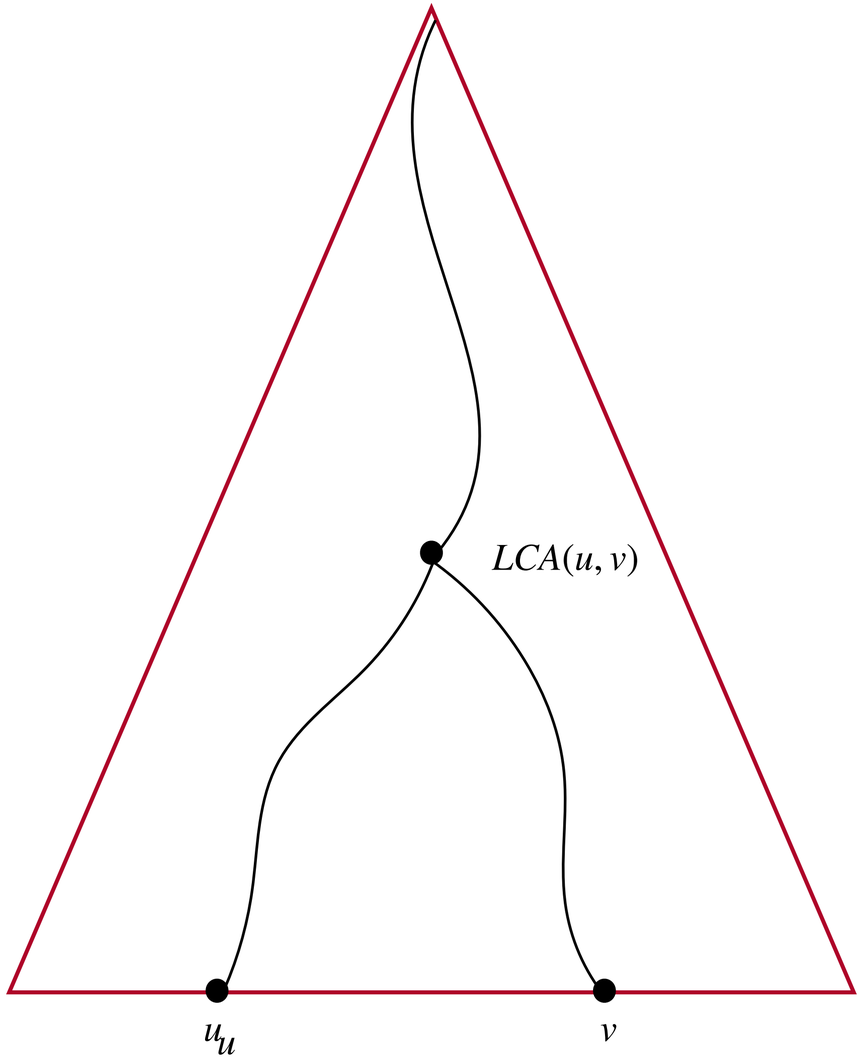


Longest Common Substring

You probably came up with an algorithm that uses $O(nd)$ time, but we can do better.

We will use the following fact: We can preprocess a tree of size $O(n)$ in time $O(n)$ to support lowest common ancestor (LCA) queries in constant time. [See this paper]

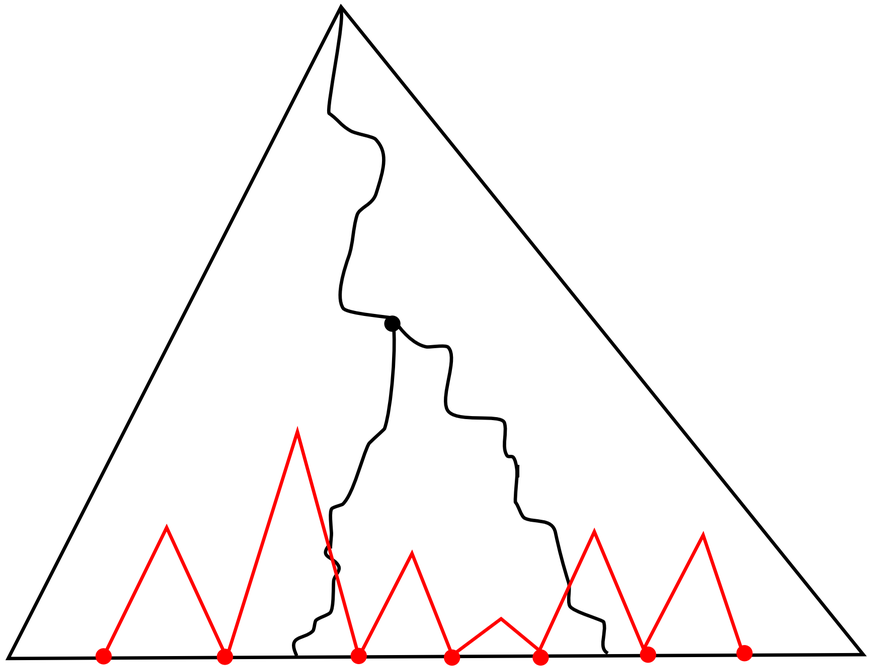
$LCA(u, v)$ must return the lowest node that is an ancestor of both u and v.



Longest Common Substring

- $LCA_c(x)$ = # of nodes in the subtree of a node x that is an LCA of two consecutive leaves corresponding to suffixes of T_c
- $A_c(x)$ = (# of leaves corresponding to suffixes of T_c in the subtree of x) - $LCA_c(x)$ = 1
- $\sum_c A_c(x)$ = number of distinct strings such that their suffixes are in the subtree
- $\sum_c$ (# of leaves corresponding to suffixes of T_c in the subtree of x) = # of leaves in the subtree of x
- $\sum_c LCA_c(x)$ can be found in $O(n)$ time for all nodes of the suffix tree via one bottom-up traversal (using LCA queries)

Longest common substring problem can be solved in $O(n)$ time and space [Hui, 1992].



Open problem

We showed that the longest common substring problem for two strings can be solved in $O(n)$ space and $O(n)$ time.

It is also known that the problem can be solved in $O(s)$ *extra* space and $O(\frac{n^2 \log n \log^* n}{L \cdot s} + n \log n)$ time, where L is answer.

What is the optimal trade-off?

Summary

Today we saw:

- Knuth-Morris-Pratt algorithm
- Trie
- Aho-Corasick algorithm
- Suffix tree
- Applications of suffix trees