Suffix trees and suffix arrays

Introduction

Exact string matching is used in many algorithms in computational biology as a first step:

Given a pattern $p = p[1 \ldots m]$, find all j occurrences of p in a text $T = T[1 \ldots n]$.

This can readily be done with string matching algorithms in time $O(m + n)$. If however, the text is very long, we would prefer not to scan it completely for every query, but rather spend time $O(m + j)$ per query.

To do that we have to preprocess the *text*. The preprocessing step is especially useful in scenarios where the text is relatively constant over time (e. g., a genome), and we will search for many different patterns.

Introduction (2)

In this lecture we first introduce such a preprocessing, namely the construction of a *suffix tree*. Later we get to know the related, more space-efficient, *suffix array*.

Both are central data structures in computational molecular biology.

This is just a brief introduction, and we will skip some important topics. These will, however, be covered in detail in future classes.

Introduction (3)

History.

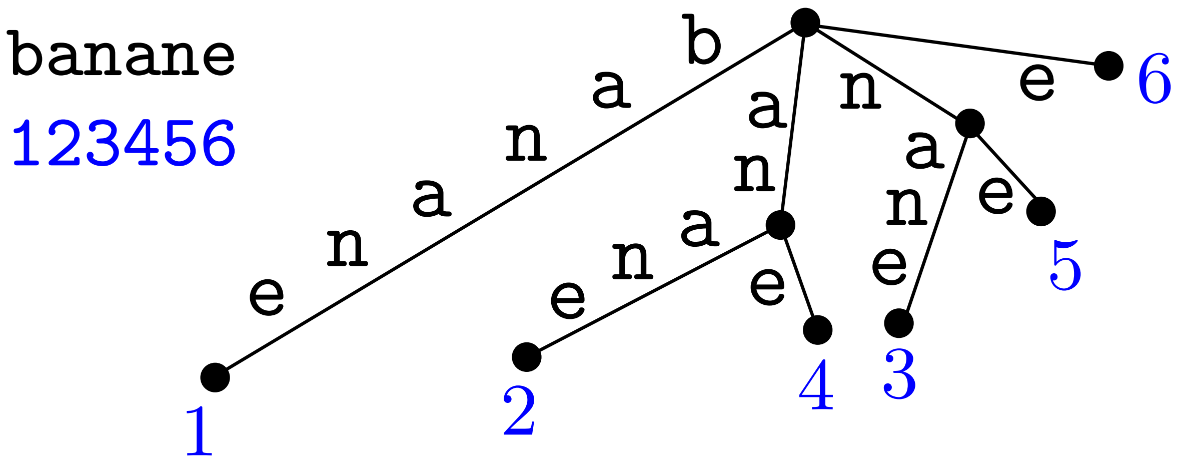
- Weiner, 1973: suffix trees introduced, linear-time construction algorithm
- McCreight, 1976: reduced space-complexity
- Ukkonen, 1995: new algorithm, easier to describe

In this lecture, we will only cover a naïve (quadratic-time) construction.

Suffix trees

Definition. Let $T = T[1 .. n]$ be a text of length n over a fixed alphabet Σ. A *suffix tree* for T is a tree with n leaves and the following properties:

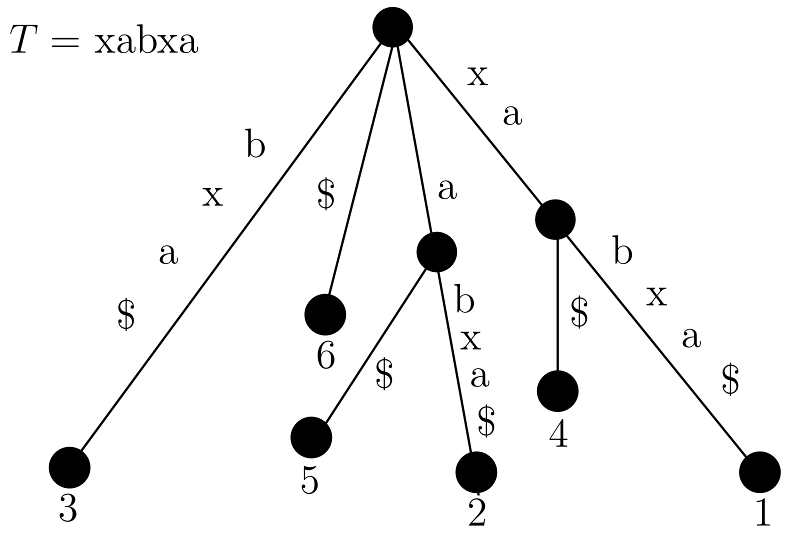
1. Every internal node other than the root has at least two children.
2. Every edge is labeled with a nonempty substring of T.
3. The edges leaving a given node have labels starting with different letters.
4. The concatenation of the labels of the path from the root to leaf i spells out the i-th suffix $T[i \ldots n]$ of T. We denote $T[i \ldots n]$ by T_i.



Marking the end of T

Note that according to the above definition there is no suffix tree if a suffix of T is a prefix of another suffix of T. For example for $T =$ `xabxa` there is no leaf for the fourth suffix T_4.

This problem is easily overcome by adding a special letter `$` to the alphabet which does not occur in T, and putting it to the end of T. This way no suffix can be a prefix of another suffix.



The above figure shows a suffix tree for the string `xabxa$`.

Storing the edge labels efficiently

What about the space consumption? The total length of all edge labels in a suffix tree can easily be $\Omega(n^2)$, e.g., for $T = abc \cdots xyz$.

Therefore, we do not store the substrings $T[i \dots j]$ of T in the edges, but only their start and end indices (i, j). Nevertheless we keep thinking of the edge labels as substrings of T.

A naïve algorithm for suffix tree construction

The *naïve algorithm* for constructing a suffix tree is as follows:

We insert the suffixes $T_1, T_2, \ldots, T_n$ (in this order) and modify the tree according to the definition.

1. Say we want to insert T_i into the current tree. We read the letters of T_i and walk down the path from the root accordingly, until a mismatch occurs. At this point we have to branch off a new edge.
2. If the mismatch occurs in the midst of an edge, then we split it into two edges and branch off from the inserted vertex.
3. Otherwise we can branch off from a vertex which is already present.

We need time $O(n - i + 1)$ for the i-th suffix and hence the total running time is

$$\sum_{1}^{n} O(n - i + 1) = \sum_{1}^{n} O(i) = O(n^2) \quad .$$

Searching with suffix trees

Given T and p. How do we find all occurrences of p in T?

Observation. Each occurrence has to be a prefix of some suffix. Each such prefix corresponds to a path starting at the root.

1. Of course, as a first step, we construct the suffix tree for T. Using the naïve method this takes quadratic time, but *linear-time* algorithms (e.g., Ukkonen's algorithm) exist.

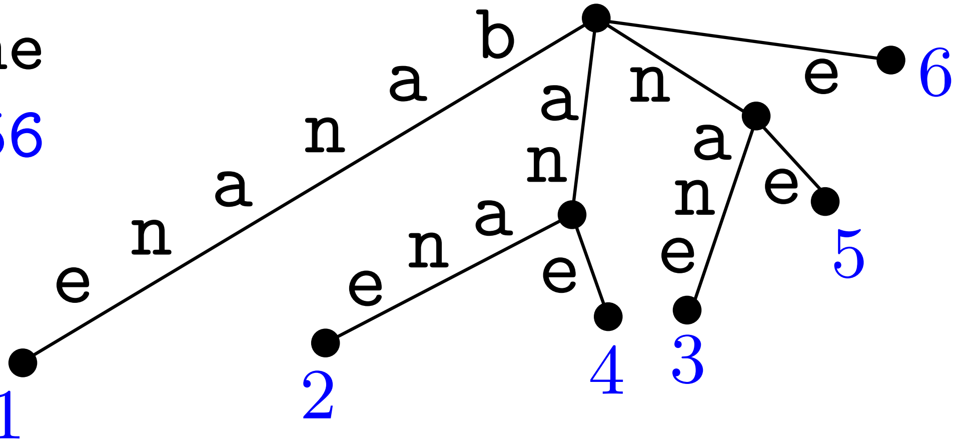
2. Try to match p on a path, starting from the root. Three cases:
   - (a) The pattern does not match $\rightarrow$ p does not occur in T
   - (b) The match ends in a node u of the tree. Set $x = u$.
   - (c) The match ends inside an edge (v, w) of the tree. Set $x = w$.

3. All leaves below x represent occurrences of p.

Searching with suffix trees (2)

Example. (completion by blackboard)

```
banane
123456
```



Searching with suffix trees (3)

Theorem. We can find all j occurrences of a pattern $p = p[1 \dots m]$ in a suffix tree in time $O(m + j)$.

Proof.

- Finding x takes time $O(m)$.
- Collecting the j leaves (e.g., using depth-first search) takes time $O(j)$.*
- Together, this yields $O(m + j)$.

*Why? How many nodes does the subtree have?

Suffix arrays

Suffix arrays were introduced by Manber and Myers in 1989 (and published in 1993).

While both suffix trees and suffix arrays require $O(n)$ space, suffix arrays are more space efficient. A recent suffix tree implementation requires 15-20 Bytes per character. For suffix arrays, as few as 5 bytes are sufficient (with some tricks).

This is offset by a moderate increase in search time from $O(m + j)$ to $O(m + j + \log n)$. In practice this increase is counterbalanced by better cache behavior.

Suffix arrays (2)

Definition. Given a text T of length n, the *suffix array* for T, called *suftab*, is an array of integers of range 1 to $n+1$ specifying the lexicographic ordering of the $n+1$ suffixes of the string T\$.

```
mississippi$
123456789012
```

i	T_i	$T_{\text{suftab}[i]}$	$\text{suftab}[i]$
1	`mississippi$`	`$`	12
2	`ississippi$`	`i$`	11
3	`ssissippi$`	`ippi$`	8
4	`sissippi$`	`issippi$`	5
5	`issippi$`	`ississippi$`	2
6	`ssippi$`	`mississippi$`	1
7	`sippi$`	`pi$`	10
8	`ippi$`	`ppi$`	9
9	`ppi$`	`sippi$`	7
10	`pi$`	`sissippi$`	4 $\longleftarrow$
11	`i$`	`ssippi$`	6
12	`$`	`ssissippi$`	3

$\text{suftab}(10) = 4$ means: T_4 is number 10 in the sorted list.

We will assume that n fits into 4 bytes of memory. Then the basic form of a suffix array needs only $4n$ bytes. The suffix array can be computed by sorting the suffixes, as illustrated in the above example.

Why another algorithm?

The suffix array can be constructed in $O(n^2 \log n)$ time by sorting the suffix indices using a sorting algorithm*. But such an approach fails to take advantage of the fact that we are sorting a collection of related suffixes. We cannot get an $O(n)$ time algorithm in this way.

Alternatively, we could first build a suffix tree in linear time, then transform the suffix tree into a suffix array *in linear time**, and finally discard the suffix tree. Of course, sufficient memory has to be available to construct the suffix tree. Thus this approach fails for large texts.

In this lecture, we will *not* discuss the original construction proposed by Manber and Myers which needs (essentially) $8n$ bytes and runs in $O(n \log n)$ time.

*How?

Searching

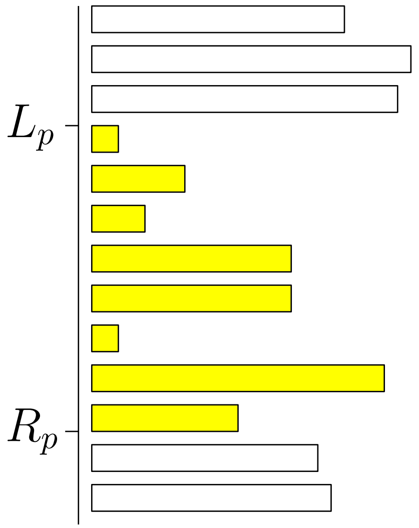
After constructing our suffix array we have the table *suftab*, which gives us the suffixes of T in sorted order. Suppose now we want to find all j instances of a pattern $p = p_1, \ldots, p_m$ of length $m < n$ in T.

Then let

$$L_p = \min\{k \mid p \leq T_{\textit{suftab}\,[k]} \text{ or } k = n\}$$

and

$$R_p = \max\{k \mid p \geq T_{\textit{suftab}\,[k]}[1 \ldots m] \text{ or } k = -1\} \ .$$



Since *suftab* is ordered, it follows that p matches a suffix T_i if and only if $i =$ *suftab* $[k]$ for some $k \in [L_p, R_p]$. Hence a simple binary search can find L_p and R_p. Each comparison in the search needs $O(m)$ character comparisons, and we can find all j instances in the string in time $O(m \log n + j)$.

Searching (2)

This is the simple code piece to search for L_p (the search for R_p is similar):

```
if $p \leq T_{\textsf{suftab}\,[1]}$ then $L_p = 1$;
else if $p > T_{\textsf{suftab}\,[n]}$ then $L_p = n + 1$;
else
    $(L, R) = (1, n)$;
    while $R - L > 1$ do
        $M = \lceil (L + R)/2 \rceil$;
        if $p \leq T_{\textsf{suftab}\,[M]}$ then $R = M$; else $L = M$;
    $L_p = R$;
```

Searching (3)

For example if we search for $p =$ `aca` in the text $T =$ `acaaacatat$` $L_p = 4$ and $R_p = 5$. We find the value L_p and R_p respectively, by setting (L, R) to $(1, n)$ and changing the borders of this interval based on the comparison with the suffix at position $\lceil (L + R)/2 \rceil$, e.g., we find L_p with the sequence: $(1, 10) \Rightarrow (1, 6) \Rightarrow (1, 4) \Rightarrow (3, 4)$. Hence $L_p = 4$.

1	`$`
2	`aaacatat$`
3	`aacatat$`
4	`acaaacatat$`
5	`acatat$`
6	`at$`
7	`atat$`
8	`caaacatat$`
9	`catat$`
10	`t$`
11	`tat$`

Searching (4)

The binary searches each need $O(\log n)$ steps. In each step we need to compare m characters of the text and the pattern in the $\leq$ operations. Finally we have to report the j matches. This leads to a running time of $O(m \log n + j)$.

Can we do better?

While the binary search continues, L and R are the left and right boundaries of the current search interval. At the start, L equals 1 and R equals n. Then in each iteration of the binary search a query is made at location $M = \lceil (R + L)/2 \rceil$ of *suftab*.

We keep track of the longest prefixes of *suftab* (L) and *suftab* (R) that match a prefix of p. Let l and r denote the prefix lengths respectively and let

$$\mathit{mlr} = \min(l, r) \quad .$$

Searching (5)

Then we can use the value *mlr* to accelerate the lexicographical comparison of p and the suffix *suftab*$[M]$. Since *suftab* is ordered, it is clear that all suffixes between L and R share the same prefix. Hence we can start the first comparison at position *mlr* $+ 1$.

In practice this trick already brings the running time to $O(m + \log n + j)$, however one can construct an example that still needs time $O(m \log n + j)$.

Final remarks

- This was just an introduction.
- We did not cover:
  - linear-time construction of suffix trees (e. g., Ukkonen's algorithm)
  - direct $O(n \log n)$ time construction of suffix arrays (Manber & Myers)
  - faster (theoretical) search in suffix arrays (longest common prefix values)
  - $\ldots$