582670 Algorithms for Bioinformatics

Lecture 6: Combinatorial Pattern Matching

8.10.2015

Background: Whole Genome Sequencing

- Sequencing the complete genome of an individual is now relatively cheap
- Next generation sequencing methods produce a large number of short reads
- The reads are aligned against a reference genome
- This identifies differences between the individual and the reference
- Many applications, for example, finding potential disease causing mutations

Aligning reads

- Aligning the reads against the reference genome could be done using the aligning methods in Lecture 4
- The running time is $O(|Genome| \cdot |Reads|)$
  - $|Genome|$ = length of reference genome
  - $|Reads|$ = sum of the lengths of the reads
- For example
  - $|Genome| \approx 3 \cdot 10^9$ (Human genome)
  - $|Reads| \approx 10^9 \times 100$ (one billion reads of length one hundred)
  - $|Genome| \cdot |Reads| \approx 3 \cdot 10^{21}$
- Far too slow

Multiple Pattern Matching Problem

- Goal: Find all occurrences of a collection of patterns in a text
- Input: A string *Text* and a collection *Patterns* containing (shorter) strings
- Output: All positions in *Text* where a string from *Patterns* appears as a substring

- Models read aligning
    - For the moment we ignore sequencing errors and mutations

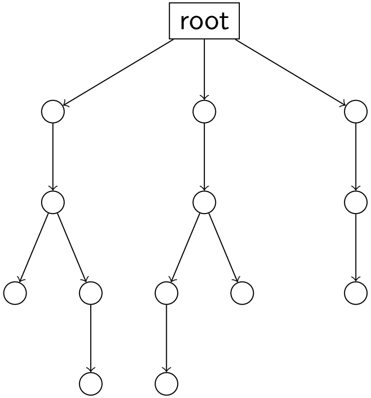
Single pattern matching

- Many algorithms for searching a single pattern *Pattern* in a text
    - *String Processing Algorithms* course, period II
- Brute force algorithm runs in $O(|Text| \cdot |Pattern|)$ time in the worst case
- Best average case runtime: $O(|Text| \cdot \log(|Pattern|) \ / \ |Pattern|)$
- Still too slow, about 10^{18} steps for the read aligning example

- Instead of searching for each pattern separately, can we search for all of them simultaneously?

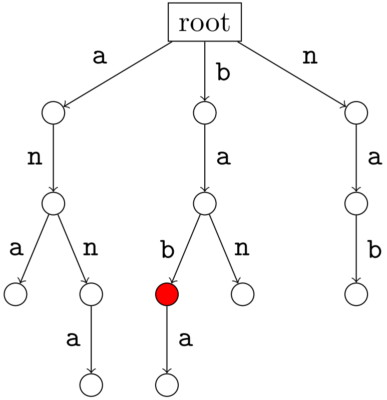
Rooted Tree

- Directed graph
- A single node with indegree 0, called the *root*
- Other nodes have an indegree 1
- Every node is reachable from the root
  - For each node there must be a unique path from the root to the node
- A node with outdegree 0 is called a *leaf*



Trie

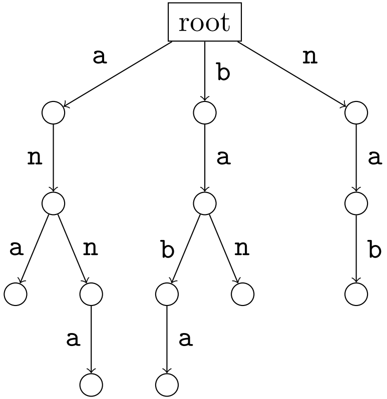
- Rooted tree
- Each edge is labeled with a letter
- Edges leading out of a given node have distinct labels
- Each node represents the string obtained by concatenating the letters on the path from the root to the node
- For example, the red node represents the string bab



Trie for a set of strings

Trie(*Patterns*)

- Every leaf represents some string in *Patterns*
- Every string in *Patterns* is represented by some node
  - We assume that each node representing a pattern string is a leaf
- Example: Trie(ana, anna, baba, ban, nab)
- Can be constructed in $O(|Patterns|)$ time by inserting strings one at a time (exercise)



Prefix Trie Matching

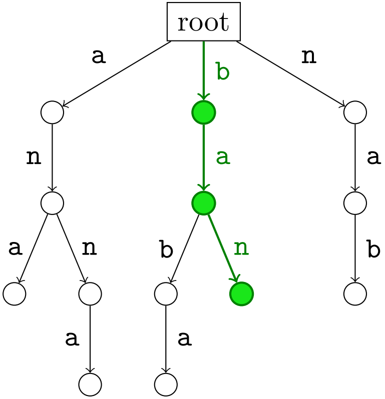
- Goal: Find if a prefix of *String* matches a pattern in a trie
- Basic idea: Starting from the root walk along the path labeled by the letters of *String*

PrefixTrieMatching(*String*, Trie(*Patterns*))

```
 1: symbol ← first letter of String
 2: v ← root of Trie(Patterns)
 3: while forever do
 4:    if there is an edge (v, w) labeled by symbol then
 5:       v ← w
 6:       symbol ← next letter of String
 7:    else if v is a leaf then
 8:       return pattern represented by v
 9:    else
10:       return empty string (signifying no match)
```

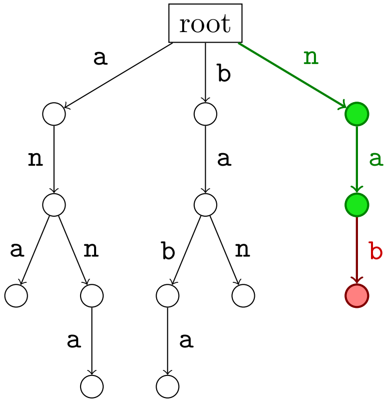
Example: Successful prefix trie matching

PrefixTrieMatching(`banana`, Trie(`ana`, `anna`, `baba`, `ban`, `nab`))



Example: Unsuccessfull prefix trie matching

PrefixTrieMatching(nana, Trie(ana, anna, baba, ban, nab))



Trie Matching

- Goal: Find all occurrences of patterns in *Text*
- Basic idea: Perform prefix trie matching for all suffixes of *Text*

```
TrieMatching(Text, Trie(Patterns))
1: for each Suffix of Text do
2:     Pattern ← PrefixTrieMatching(Suffix, Trie(Patterns))
3:     if Pattern is not empty then
4:         output Pattern occurs in Text at the starting position of Suffix
```

Example: Trie Matching

Patterns	*Text* = banana
ana	banana
anna	anana
baba	nana
ban	ana
nab	na
	a

Example: Trie Matching

Patterns	*Text* = <u>ban</u>ana
ana	<u>ban</u>ana
anna	anana
baba	nana
<u>ban</u>	ana
nab	na
	a

Example: Trie Matching

Patterns	*Text* = b<u>ana</u>na
<u>ana</u>	banana
anna	<u>ana</u>na
baba	nana
ban	ana
nab	na
	a

Example: Trie Matching

Patterns	*Text* = banana
ana	banana
anna	anana
baba	$\underline{\texttt{na}}\texttt{na}$
ban	ana
$\underline{\texttt{na}}\texttt{b}$	na
	a

Example: Trie Matching

Patterns	*Text* = ban<u>ana</u>
<u>ana</u>	banana
anna	anana
baba	nana
ban	<u>ana</u>
nab	na
	a

Example: Trie Matching

Patterns	*Text* = banana
ana	banana
anna	anana
baba	nana
ban	ana
nab	na
	a

Example: Trie Matching

Patterns	*Text* = banana
ana	banana
anna	anana
baba	nana
ban	ana
nab	na
	a

Trie Matching: Analysis

- Trie construction: $O(|Patterns|)$ time
- Trie matching: $O(|Text| \cdot |LongestPattern|)$ time
- Trie matching time can be improved to $O(|Text|)$
  - Aho–Corasick algorithm which uses an augmented trie
- Fast enough
- The problem is the memory needed for the trie
  - $O(|Patterns|)$ nodes and edges
  - The trie for the reads may need terabytes of memory

Suffix Trie

- We are comparing one set of strings (*Patterns*) to another set of strings (suffixes of *Text*)

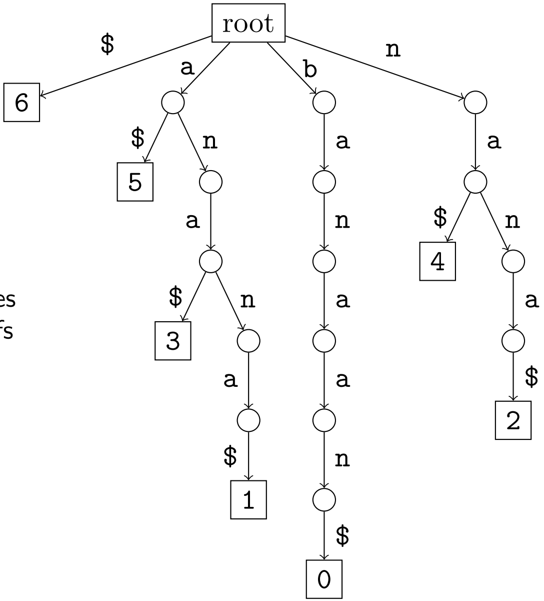
Patterns	*Text* = banana
ana	banana
anna	anana
baba	nana
ban	ana
nab	na
	a

- What if we swap the roles: use a trie of the suffixes
  - In read aligning, $|Text| < |Patterns|$
  - The reference genome changes only rarely.
    No need to rebuild the trie every time.

Suffix Trie

SuffixTrie(*Text*)

- Trie of suffixes of *Text*
- Append special symbol \$ to *Text*
  - Ensures that all suffixes are represented by leafs
- Leafs are labeled by the starting positions of the suffixes
- Example: SuffixTrie(`banana$`)



Suffix Trie Matching

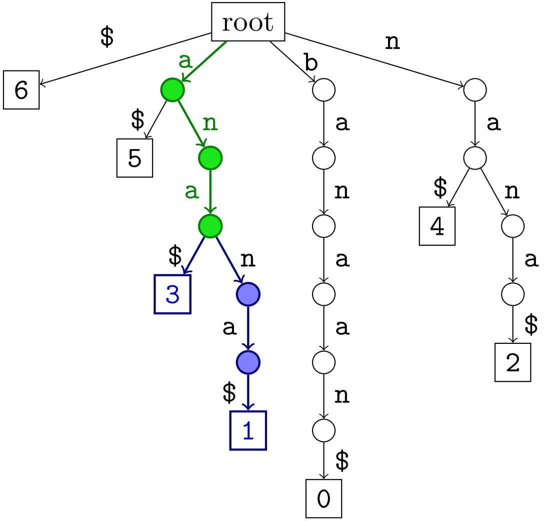
- Goal: Find all occurrences of *Pattern* in *Text* using SuffixTrie(*Text*)
- Basic idea: Starting from the root walk along the path labeled by the letters of *Pattern*

SuffixTrieMatching(*Pattern*, SuffixTrie(*Text*))

```
1:  symbol ← first letter of Pattern
2:  v ← root of SuffixTrie(Text)
3:  while forever do
4:      if there is an edge (v, w) labeled by symbol then
5:          v ← w
6:          if there are Pattern symbols left then
7:              symbol ← next letter of Pattern
8:          else
9:              return all positions stored in the leafs below v
10:     else
11:         return empty set of occurrences
```

Example: Suffix Trie Matching

$$\textbf{SuffixTrieMatching}(\texttt{ana}, \text{SuffixTrie}(\texttt{banana\$})) = \{1, 3\}$$



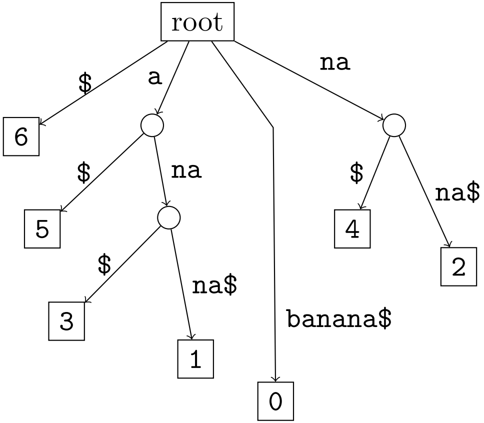
Suffix Trie is too big

- The total length of all suffixes is $O(|Text|^2)$
- This is also the size of $\mathrm{SuffixTrie}(Text)$
- This is far too much
- Fortunately, there is a much smaller alternative: *suffix tree*

Suffix Tree

SuffixTree(*Text*)

- SuffixTrie(*Text*) where each non-branching path segments has been concatenated into a single edge
- The edge labels are substrings of *Text* and are represented by pointers to *Text*
- Memory: $O(|Text|)$
- Can be used the same way as SuffixTrie(*Text*)



Suffix Tree Construction

- A brute force constrution would require $O(|Text|^2)$ time
- There are more sophisticated algorithms that can construct the suffix tree in linear time

Multiple Pattern Matching with Suffix Tree

Algorithm

- Construct $\mathrm{SuffixTree}(\mathit{Text})$
- For each pattern, find the occurrences using **SuffixTreeMatching**
  - **SuffixTreeMatching** is a version of **SuffixTrieMatching** adapted to use suffix tree instead of suffix trie

Runtime analysis

- Suffix tree construction: $O(|\mathit{Text}|)$
- Matching: $O(|\mathit{Patterns}| + |\mathit{Occurrences}|)$
  - $|\mathit{Occurrences}| =$ total number of occurrences
  - We can assume $|\mathit{Occurrences}| \leq |\mathit{Text}|$ (Why?)
- Total: $O(|\mathit{Text}| + |\mathit{Patterns}|)$

Is suffix tree too big?

- The size of suffix tree is $O(|Text|)$ which is asymptotically optimal but this ignores constant factors
- Even a careful implementation of a suffix tree needs about 20 times the size of the text
- For large genomes, this is a lot
- There are more compact alternatives: the suffix array and text indexes based on the Burrows–Wheeler transform

Suffix Array

SuffixArray(*Text*)

- Array of all suffixes of *Text* in lexicographical order
  - Suffixes are represented by their starting positions
- Sophisticated algorithms for linear time construction
- Pattern matching by binary searching
- Memory: $\sim 4 \cdot |\mathit{Text}|$

SuffixArray(`banana$`)

6	`$`
5	`a$`
3	`ana$`
1	`anana$`
0	`banana$`
4	`na$`
2	`nana$`

Burrows–Wheeler Transform

$\mathrm{BWT}(\mathit{Text})$

- Sort the set of all rotations of *Text* lexicographically
- $\mathrm{BWT}(\mathit{Text})$ is the string formed by the last letters of the rotations
- Example: $\mathrm{BWT}(\texttt{banana\$}) = \texttt{annb\$aa}$

Sorted rotations
`$ b a n a n a`
`a $ b a n a n`
`a n a $ b a n`
`a n a n a $ b`
`b a n a n a $`
`n a $ b a n a`
`n a n a $ b a`

- Pattern matching using an algorithm called *backward search*
  - Requires additional data structures
- Memory: less than $2 \cdot |\mathit{Text}|$
  - Sometimes even less than $|\mathit{Text}|$ (compression)

Full-Text Indexes

- Data structures such as suffix trees, suffix arrays and BWT-based data structures supporting fast pattern matching are often called full-text indexes
- They have numerous other applications
    - Example: Find the longest repeat in a text (exercise)
- Full-text indexes (as well as single and multiple pattern matching without an index) are covered in more detail on the course *String Processing Algorithms* (period II)

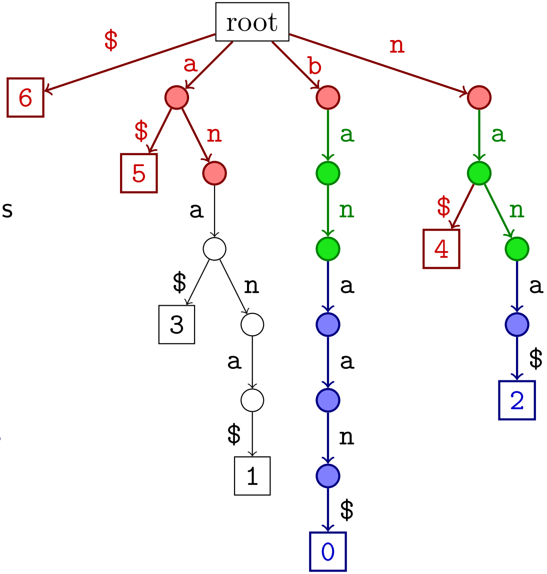
- When trying to solve a problem using one of these data structures, it is probably easiest to first design an algorithm for the suffix trie and then translate the solution to the more compact alternatives

Pattern Matching with Mismatches

- We want to align reads against the reference genome even if the match is not perfect
  - In fact, we are most interested in the differences
  - Reads may also contain sequencing errors
- How can we use the suffix trie (or its compact alternatives) to find occurrences with some mismatches?

Pattern Matching with Mismatches on Suffix Trie

- Follow all paths from root as long as the number of mismatches is not too large
- Example: Searching for pattern `man` in text `banana$` with at most one mismatch finds two occurrences at positions 0 and 2.



Pattern Partitioning

- Another approach is to reduce approximate matching to exact matching
- When allowing at most k mismatches, split the pattern into $k + 1$ pieces, called *seeds*
- Find the exact occurrences of all the seeds
- For each seed occurrence, try expanding it to an approximate occurrence of the whole pattern
- Example: At least one of four seeds matches exactly if there are no more than three mismatching characters

```
    X X X X X X|X X X X X X|X X X X X X|X X X X X X
... X X Y X X X X X X X X Y X X X X X X X Y X X X X ...
```

END OF COURSE

Your feedback on improving the course is greatly appreciated.
Please use the anonymous feedback form at
`https://ilmo.cs.helsinki.fi/kurssit/servlet/Valinta?kieli=en`