BMI/CS 776
Lecture #17: Pattern Matching - Suffix trees

Colin Dewey
2007.03.20
Alignment vs pattern matching

• Global sequence alignment
  • Input: \( n \geq 2 \) relatively short sequences
  • Homology assumptions: homologous along entire length, colinear
  • Goal: determine homologous positions

• Pattern matching
  • Input: \( n \geq 1 \) sequences (short or long)
  • Homology assumptions: none
  • Goal: find short exact/inexact substring (local) matches between or within input sequences
Why pattern matching?

• Applications:
  • Database search - short query, large DB
  • Alignment of long sequences - exact global alignment no longer feasible
  • Alignment of rearranged/duplicated sequences - no longer global alignment

• Key idea:
  • Short local matches can seed longer global alignments
Database search

highly-similar short match to query

DB

query

..............

pattern matching

global alignment of each candidate to query
Global alignment of long sequences

Find seed matches using pattern matching

Sparse dynamic programming with seed matches
Alignment of rearranged sequences

Each dot represents a seed match between genome sequences

whole-genome dot plot
Suffix trees

- Very important for pattern matching
- Substring problem:
  - Given text $T$ of length $m$
  - Preprocess $T$ in $O(m)$ time
  - Such that, given query string $S$ of length $n$, find occurrence (if any) of $S$ in $T$ in $O(n)$ time
- Suffix trees solve this problem, and many more
History of suffix trees

• Weiner (1973) - First linear time algorithm for suffix tree construction
• “the algorithm of 1973” (Knuth)
• McCreight (1976) - Space efficient version of algorithm
• Ukkonen (1995) - Simple, elegant algorithm, with “online” property
Suffix tree definition

- A suffix tree $T$ for a string $S$ of length $m$ is a tree with the following properties:
  - *rooted* and *directed*
  - $m$ leaves, labeled 1 to $m$
  - Each edge labeled by a substring of $S$
  - Concatenation of edge labels on path from root to leaf $i$ is suffix $i$ of $S$ (we will denote this by $S_{i...m}$)
  - Each internal non-root node has at least two children
  - Edges out of a node must begin with different characters
Suffix tree example

- $S = \text{"banana$"}$
- Add ‘$’ to end so that suffix tree exists
Solving the substring problem

- Assume we have suffix tree $T$
- $\text{FindMatch}(Q, T)$:
  - Follow (unique) path down from root of $T$ according to characters in $Q$
  - If all of $Q$ is found to be a prefix of such a path
    - return label of some leaf below this path
  - else, return no match found
Solving the substring problem

\[ Q = \text{nan} \]

\[ Q = \text{anab} \]

return 3

return no match found
Runtime of substring problem with suffix tree

- Finite alphabet: $O(1)$ work at each node
- Edges out of each node start with unique characters: unique path from root
- Size of tree below end of path: $O(k)$, $k =$ number of suffixes starting with $Q$
- $O(n + k)$ time to report all $k$ matching substrings
- $O(n)$ to report just one with an additional trick
Naive suffix tree building

- Now we need a $O(m)$ time algorithm for building suffix trees
- Naive algorithm is $O(m^2)$:
  - $T \leftarrow$ empty tree
  - For $i$ from 1 to $m$:
    - Add suffix $S_{i\ldots m}$ to $T$ by finding longest matching prefix of $S_{i\ldots m}$ in $T$ and branching from there

$O(m)$
O(m^2) suffix tree building

i=1

i=2

i=3

i=4 ...

banana

$\rightarrow$

l

2

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

2

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

2

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

2

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a

n

banana

$\rightarrow$

a
Ukkonen’s $O(m)$ algorithm

- On-line algorithm
- Builds implicit suffix tree for each prefix of string $S$
- Implicit suffix tree of $S_{1...i}$ denoted $I_i$
- Builds $I_1$, then $I_2$ from $I_1$, ..., then $I_m$ from $I_{m-1}$
- Basic algorithm is $O(m^3)$, but with a series of tricks, it is $O(m)$
Implicit suffix tree

- Suffix tree → implicit suffix tree
- remove $ characters from labels
- remove edges with empty labels
- remove internal nodes without two children
Ukkonen’s algorithm overview

• Construct \( I_1 \)

• For \( i \) from 1 to \( m - 1 \):
  • For \( j \) from 1 to \( i + 1 \):
    • Find end of path from root labeled \( S_{j...i} \)
    • Add character \( S_{i+1} \) to the end of this path in the tree, if necessary (suffix extension)
Conversion to suffix tree

• Convert implicit suffix tree at end of algorithm into true suffix tree

• Simply run algorithm for one more iteration with $ final character

• Traverse tree to label leaf edges with positions
Suffix extension rule I

1. If path $S_{j...i}$ in tree ends at leaf, add character $S_{i+1}$ to end of label of edge into leaf

$$S_{j...i} = ...an \quad S_{j...i+1} = ...ana$$
Suffix extension rule II

II. If there are paths continuing from path $S_{i...j}$ in the tree, but none starting with $S_{i+1}$, then create a new leaf edge with label $S_{i+1}$ at the end of path $S_{i...j}$ (creating a new internal node if $S_{i...j}$ ends in the middle of an edge)

$$S_{j...i} = \ldots na$$
$$S_{j...i+1} = \ldots nay$$
Suffix extension rule III

III. If there are paths continuing from path $S_{i...j}$ in the tree, and one starts with $S_{i+1}$, then do nothing

$S_{j...i} = \ldots na$  \hspace{1cm} $S_{j...i+1} = \ldots nan$
Suffix links

• How to find end of each suffix $S_{j...i}$?

• Could search down tree in $O(i-j+1)$ time $\rightarrow O(m^3)$ time total

• Better: create links between nodes corresponding to ends of similar suffixes

• With some additional tricks, get runtime to $O(m^2)$ time total
A suffix link is a pointer from an internal node $v$ to another node $s(v)$ where

- $x$ is a character, $\alpha$ is a substring (possibly empty)
- $v$ has path-label $x\alpha$
- $s(v)$ has path-label $\alpha$
Edge-label compression

- Label edges with pair of indices into string rather than with explicit substring
- Makes space requirement only $O(m)$

$$S = \text{“banana$”}$$
Final runtime

• With a few more tricks and implementation details, Ukkonen’s algorithm runs in time $O(m)$

• More details found in (Ukkonen, 1995) or book by Dan Gusfield (Gusfield, 1997)