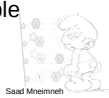
Computational Biology

Lecture 18

Saad Mneimneh

Genome Rearrangements Finding preserved genes

- We have seen before how to rearrange a genome to obtain another one based on:
  - Reversals
  - Knowledge of preserved blocks (or genes)
- Now we are concerned in determining the preserved genes, or more generally, given two strings *x* and *y*, determine all their possible maximal matches



Saad Mneimneh

Global alignment

- One possibility is to perform a global alignment of *x* and *y* with a special scoring scheme (for instance +1, 0, 0) and identify the maximal positively scoring chunks.
  - Takes O(mn) time
  - Does not give all candidate matches
  - Might not give maximal matches
- Example:

```
x= bbbaxaaabbb
y =------aabbb
```

missed

```
bbbaxaaabbb
---a---abbb
```

non-maximal, aabbb is better

Saad Mneimneh

A simple way: *k*-mers

Another possibility is to find common *k*-mers. Here's one way:

Algorithm:

- Denote a k-mer of x by (w, 0, i) where $w = x_i \ldots x_{i+k-1}$
- Denote a *k*-mer of *y* by (z, 1, j) where $z = y_j \ldots y_{j+k-1}$
- Sort them lexicographically
- Deduce all *k* long matches between *x* and *y*

Saad Mneimneh

k-mers

Example:

x = cabbc, *y* = bbcab

x's 3-mers: (cab, 0, 1), (abb, 0, 2), (bbc, 0, 3)
y's 3-mers: (bbc, 1, 1), (bca, 1, 2), (cab, 1, 3)

sort them: (abb, 0, 2), (bbc, 0, 3), (bbc, 1, 1), (bca, 1, 2), (cab, 0, 1), (cab, 1, 3)

Identify matches: (bbc, 0, 3), (bbc, 1, 1) and (cab, 0, 1), (cab, 1, 3)

Saad Mneimneh

Disadvantages

- Worst case running time still $O(mn)$
  - e.g. $O(m)$ *k*-mers in *x* and $O(n)$ *k*-mers in *y* are identical.
  - *Thought*: aren't we supposed to find these anyway and, therefore, the $O(mn)$ bound is not necessarily bad?
  - No, they might be small insignificant matches, we are interested in maximal matches
- Making *k* larger reduces the running time because it results in a shortest list of matches, but we might miss significant matches

Saad Mneimneh

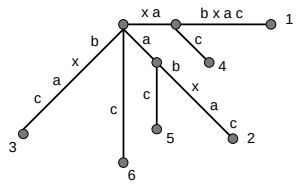
A better solution… Suffix tree

- We will use an efficient data structure called suffix tree that stores all suffixes of a string and supports fast lookup
- Definition: A suffix tree *T* of a string *s* of length *m* is a rooted tree such that:
  - It has exactly *m* leaves numbered 1 to *m*
  - Each internal node other than the root has at least two children
  - Each edge is labeled by a substring of *s*
  - No two edge labels out of a node start with the same character
  - For any leaf *i*, the concatenation of the edge labels on the path from the root to *i* spells out the suffix $x_i \ldots x_m$

Saad Mneimneh

Example of suffix tree

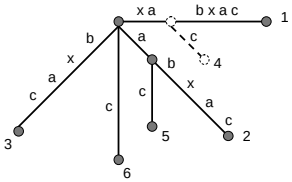
s = xabxac



Saad Mneimneh

Existence

- Does a suffix tree of a string *s* always exist?
- Consider *s* = xabxa



- If a suffix is a prefix of another suffix, then the path for the first suffix would not end at a leaf!
- Solution: always terminate a string with a special character $ that does not occur anywhere.

Saad Mneimneh

Properties of a suffix tree

A suffix tree satisfies the following:

- |E| = |V| – 1 (tree)
- Number of leaves = m + 1 (now |s| = m + 1)
- Since each internal node has at least two children, the number of edges |E| = O(# leaves) = O(m)
- Any sub-tree with k leaves satisfies |E| = O(k)

Saad Mneimneh

Building a suffix tree

Here's a simple algorithm:

given $x = x_1 \ldots x_m$

insert a special character $ at the end of s
initialize the tree T to one root

for j = 1 to m + 1

find the longest match of $x_j \ldots x_m$ in T starting from the root and following a unique path

split the edge where the match stops, add a new node w

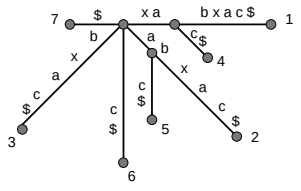
add an edge (w, j) (j is the new leaf) and label it with the remaining unmatched characters of $x_j \ldots x_m$

Saad Mneimneh

Example

s = xabxac

xabxac$
abxac$
bxac$
xac$
ac$
c$
$



Saad Mneimneh

Analysis

- Running time: $O(m^2)$
  - Each suffix requires $O(m)$ time to update the tree
- But, there exists an $O(m)$ time suffix tree algorithm
- Space: $O(m)$
  - How? Each label has $O(m)$ characters and we have $|E| = O(m)$ labels!
  - Solution: do not explicitly store labels, but store the indices $[i,j]$ of a label

Saad Mneimneh

Now what?

- How can we use the suffix tree data structure to identify all maximal matches between two strings x and y?
- Consider first the following problem: given a string x, determine all locations where another string y occurs.
- This can be solved efficiently as described next.

Saad Mneimneh

string matching

Find all occurrences of y in x

Algorithm

build a suffix tree T for x	$O(m)$
Match the characters of y along the unique path in T until (case 1) either y is exhausted or (case 2) no more matches are possible	$O(n)$
if (case 2) y does not occur in x	$O(1)$
else the k leaves in the sub-tree below the point of the last match give the k location of y in x (traverse the tree in linear time)	$O(k)$

Saad Mneimneh

Correctness

- Why is the string matching algorithm correct?
- If *y* occurs in *x* at position *i*, then the i^{th} suffix of *x* must start with *y*
- Therefore, leaf *i* must be reached by the path determined by *y*

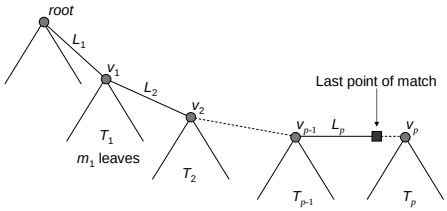
Saad Mneimneh

Finding maximal matches

- Given *x* and *y*, we would like to find all maximal matches between *x* and *y*
  - $x_i \ldots x_{i+l} = y_j \ldots y_{j+l}$
  - Cannot extend $x_i \ldots x_{i+l}$ and $y_j \ldots y_{j+l}$ and obtain a match
- We will find all matches starting at y_j that cannot be extended to the right
  - Build a suffix tree for *x* (do this only once)
  - Find the path in *T* determined by the longest possible prefix of the suffix $y_j \ldots y_n$ (it could stop in the middle of an edge *e* in that case *e* is part of the path)
  - Let v_k, $k = 1 \ldots p$ be an internal node on this path and T_k be the sub-tree rooted at v_k that excludes v_{k+1}
  - Identify the leaves in each sub-tree

Saad Mneimneh

Illustration



A leaf *i* in T_k gives the location in *x* of a match between $x_i \ldots x_{i+|L1|+\ldots+|Lk|-1}$ and $y_j \ldots y_{j+|L1|+\ldots+|Lk|-1}$ that cannot be extended to the right

Running time: $O(m)$ {building *T*} + $\underbrace{O(\Sigma|L_k| + \Sigma m_k)}_{O(n+m)}$

Saad Mneimneh

What about left?

- Given a leaf i, let $left(i)$ be the character x_{i-1}
- If $left(i) \neq y_{j-1}$, then i represents a maximal match
- Therefore, we obtain all maximal matches between x and y in $O(mn)$ time by repeating the previous algorithm for every suffix of y

Saad Mneimneh

Algorithm

Build a suffix tree T for x — O(m)

for j = 1 to n

find the path in T determined by the longest possible prefix of the suffix $y_j \ldots y_n$ (it could stop in the middle of an edge e in that case e is part of the path) — O(n)

let v_k, $k = 1 \ldots p$ be an internal node on this path and T_k be the sub-tree rooted at v_k that excludes v_{k+1}

Let $l(v_k)$ = length of match up to node v_k

identify all leaves i in each sub-tree such that $left(i) \neq y_{j-1}$ — O(m)

Such a leaf i in sub-tree T_k represents a maximal match of length $l(v_k)$ starting at position i in x and position j in y

Saad Mneimneh

Generalized suffix tree for a set of strings

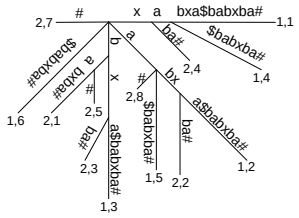
- We can build a suffix tree for a set of strings $s_1, s_2, \ldots, s_n$
  - Append a different end of string marker to each string in the set
  - concatenate all the strings together
  - build a suffix tree for the concatenated string
- The resulting suffix tree will have a leaf for each suffix of the concatenated string and is build in time proportional to the sum of all lengths
- The leaf numbers can be easily converted to two numbers, one identifying a string s_i and the other a starting position in s_i

Saad Mneimneh

Example

s_1 = xabxa, s_2 = babxba

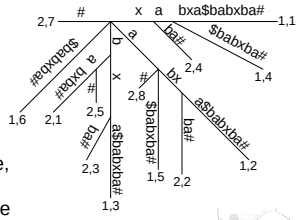
s = xabxa$babxba#



Saad Mneimneh

Fix labels of leaf edges

- One defect is that the tree now represents suffixes that span more than one original string
- Because each string marker occurs only once, the unwanted suffixes are removed by fixing the label on leaf edges



Saad Mneimneh

Suffix tree for *x* and *y*

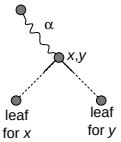
- Therefore, given two strings *x* and *y*, we can build a suffix tree for both in O(*m* + *n*) (i.e. linear) time.
- Each leaf in the tree represents
  - Either a suffix from *x*
  - Or a suffix from *y*
- Mark each internal node *v* with *x* (*y*) if there is a leaf in the sub-tree of *v* representing a suffix from *x* (*y*). This can be done in linear time by a bottom up traversal of the tree from leaves to the root.
- Note that if *v* is marked *x* (*y*), all ancestors of *v* are marked *x* (*y*).

Saad Mneimneh

Common substrings

If αp is a substring of x and αq is a substring of y for $p \neq q$, then α corresponds to an internal node v marked with both x and y *and vice-versa.*

Proof: α occurs in both x and y such that the character to the right of α in x differs from the character to the right of α in y.



conversely, every internal node marked with both x and y has to satisfy the situation depicted above, then αp is a substring of x and αq is a substring of y for $p \neq q$.

Saad Mneimneh

Left diverse node

An internal node v is *left diverse* iff it has two children v_1 and v_2 with a leaf i for x in v_1's sub-tree and a leaf j for y in v_2's sub-tree, such that $left(i) \neq left(j)$

(assume x_0 and y_0 are different and distinct from any other character)

If $u\alpha p$ is a substring of x and $w\alpha q$ is a substring of y for $u \neq w$ and $p \neq q$, then α corresponds to a left diverse node v and *vice-versa.*

Proof: similar to previous proof

Call such an α a maximal common substring

Saad Mneimneh

Compact representation

- Therefore, we have only $O(m + n)$ maximal common substrings for x and y (but each maximal common substring might appear in multiple locations)
- If we identify left diverse nodes in linear time, we need only $O(m + n)$ time and space to come up with this compact representation of all maximal common substrings
- A maximal match can be represented as (p_1, p_2, l) where p_1 and p_2 are the positions of a maximal common substring of length l in x and y respectively
- We can obtain all maximal matches in $O(m + n + k)$ where k is their number (we will not present the algorithm)

Saad Mneimneh

Identifying left diverse nodes

- For each node the algorithm records:
  - the character ***a*(*v*)**:
    - the left character of every leaf for *x* in *v*'s sub-tree, or
    - a special character ε if no leaf for *x* exists in *v*'s sub-tree, or
    - a special character &
  - the character ***b*(*v*)**:
    - the left character of every leaf for *y* in *v*'s sub-tree, or
    - a special character ε if no leaf for *y* exists in *v*'s sub-tree, or
    - a special character @
- Computing $a(v)$ and $b(v)$ can be done in a bottom up approach in linear time
- Note that v is left diverse iff it has two children v_1 and v_2 with:
  - $a(v_1) \neq b(v_2)$, $a(v_1) \neq \varepsilon$, $b(v_2) \neq \varepsilon$ or
  - $b(v_1) \neq a(v_2)$, $b(v_1) \neq \varepsilon$, $a(v_2) \neq \varepsilon$
  - It takes $O(|\Sigma|^2)$ time (constant) to find two such children or none, where Σ is the alphabet (each node has at most $|\Sigma|$ children)

Saad Mneimneh