Computational Biology
Lecture 18: Genome rearrangements, finding maximal matches
Saad Mneimneh

We have seen how to rearrange a genome to obtain another one based on reversals and the knowledge of the preserved blocks or genes. Now we are concerned with determining those preserved genes. More generally, given two strings x and y we would like to find all the maximal matches between them.

Global alignments

One possibility is to perform a global alignment of the two strings x and y with a special scoring sheme; for instance, $+1$ for a match, 0 for a mismatch, and 0 for a gap. Then we could identify all the maximal positively scoring chunks of the alignment. The disadvantages of this approach is that it requires $O(mn)$ running time, might not obtain all candidate matches, and obtains matches that are not necessarily maximal.

Here's an example:

```
x = bbbaaabbb
y = abbb
```

Then an optimal alignment might be:

```
bbbaaabbb
---a--bbb
```

The above alignment find the matches $(x[4..4], y[1..1]) = a$ and $(x[7..9], y[2..4]) = bbb$. While the first match is maximal (it cannot be extended to the left or to the right), the second match is not since it can be extended to $abbb$. Moreover, the alignment missed many maximal matches among which the maximal match $(x[1..3], y[3..5]) = bbb$.

k-mers

Another possibility for identifying possible maximal matches between x and y is to find their common k-mers. Recall that a k-mer is simply a k long substring.

Here's an algorithm:

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

Example ($k = 3$):

```
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)
```

The worst case running time of this algorithm is still $O(mn)$ e.g. we can have $O(m)$ k-mers in x and $O(n)$ k-mers in y that are identical. However, it is better than above because it is actually $O((m+n)\log(m+n)+L)$, where L is the number of matches; whereas the running time of a global alignment is $O(mn)$ regardless of the number of matches. The disadvantage of this approach is that matches are not necessarily maximal and, therefore, we spend $O(L)$ time on finding short matches that we are not really looking for. A possible solution would be to increase k, which reduces L and hence the runnig time, and increases our chances of obtaining maximal matches. However, this would mean that we will miss the short (less than k, but significant) maximal matches! So by setting a value for k, we are making a compromise among the long and short maximal matches and the running time of the algorithm.

An exact solution

Here's a solution that will give all maximal matches between two strings x and y. Construct the matrix P such that $P(i,j)$=1 if $x_i = y_j$, and $P(i,j) = 0$ otherwise. Clearly a match of length k starting at x_i and y_j will correspond to a diagonal in the matrix $P(i,j), P(i+1,j+1), ..., P(i+k-1,j+k-1)$. The match is maximal if $P(i-1,j-1) = 0$ (or $i = 1$ or $j = 1$) and $P(i+k,j+k) = 0$ (or $i+k-1 = m$ or $j+k-1 = n$).

Therefore, all maximal matches can be obtained in $O(mn)$ time in the following way: First construct the matrix P. Then in a rowwise manner, check if $P(i,j)$ is the start of a maximal match, i.e. if $P(i-1,j-1) = 0$ (or does not exist). If $P(i,j)$ is the start of a maximal match, follow the match along the diagonal $P(i,j), P(i+1,j+1), ..., P(i+k-1,j+k-1)$ such that $P(i+k,j+k) = 0$ (or does not exist). Constructing the matrix takes $O(mn)$ time. Checking whether a $P(i,j)$ is the start of a match takes $O(1)$ time for a total of $O(mn)$ time for all $P(i,j)$'s. Each diagonal is investigated once and all diagonals are disjoint (maximal); therefore, finding the matches takes an additional $O(mn)$ time.

Below we describe a data structure that will allow us to obtain all maximal matches in $O(m+n+k)$, where k is their number.

Suffix trees

To find maximal matches between x and y we will use an efficient data structure called *suffix tree*. A suffix tree for string s sores all suffixes of s and supports fast lookup, e.g. it requires $O(l)$ time to find whether a string of length l is a substring of s.

Definition: A suffix tree T for 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 on outgoing edges 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 $x_i...x_m$

Below is an example of a suffix tree for $s = xabxac$:

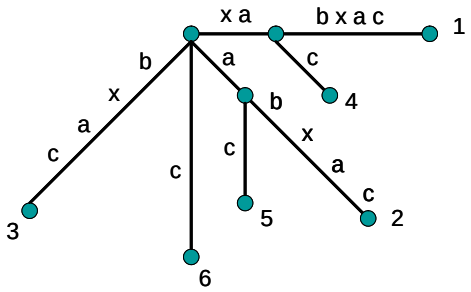


Figure 1: Suffix tree for $s = xabxac$

The definition of a suffix tree above does not guarantee the existence of a suffix tree for every string s. In fact, a suffix tree satisfying all of the above conditions is not guaranteed to exist.

Consider for instance the string $s = xabxa$, i.e. remove that last character c from the above example. There must be a path from the root to leaf number 4 that spells xa. Also, there must be a path from the root to leaf number 1 that spells $xabxac$. Since two edges at a node cannot start with the same character, the portion of the path that spells xa must be shared by the two paths. Therefore, leaf 4 has to be an internal node as well, which cannot happen!

This problem arises when a suffix of s is a prefix of another suffix of s. Therefore, we can eliminate the problem by terminating the string s with a special end marker \$ that is guaranteed not to occur anywhere in s. As a result, a suffix of s (now always ending in \$) can never be a prefix of another suffix, and this guarantees that the suffix tree exsits.

Let us look at the properties of a suffix tree T with a set of nodes V and a set of edges E:

- $|E| = |V| - 1$ (a tree)
- Number of leaves is m (assume $|s| = m$ including the special \$ end marker)
- Since every internal node, other than the root, has at least two children, $|E| = O(\#leaves) = O(m)$
- More generally, any subtree S of T with k leaves satisfies $|E_S| = O(k)$

Here's a simple algorithm for building a suffix tree:

Algorithm

given $s = s_1...s_m$

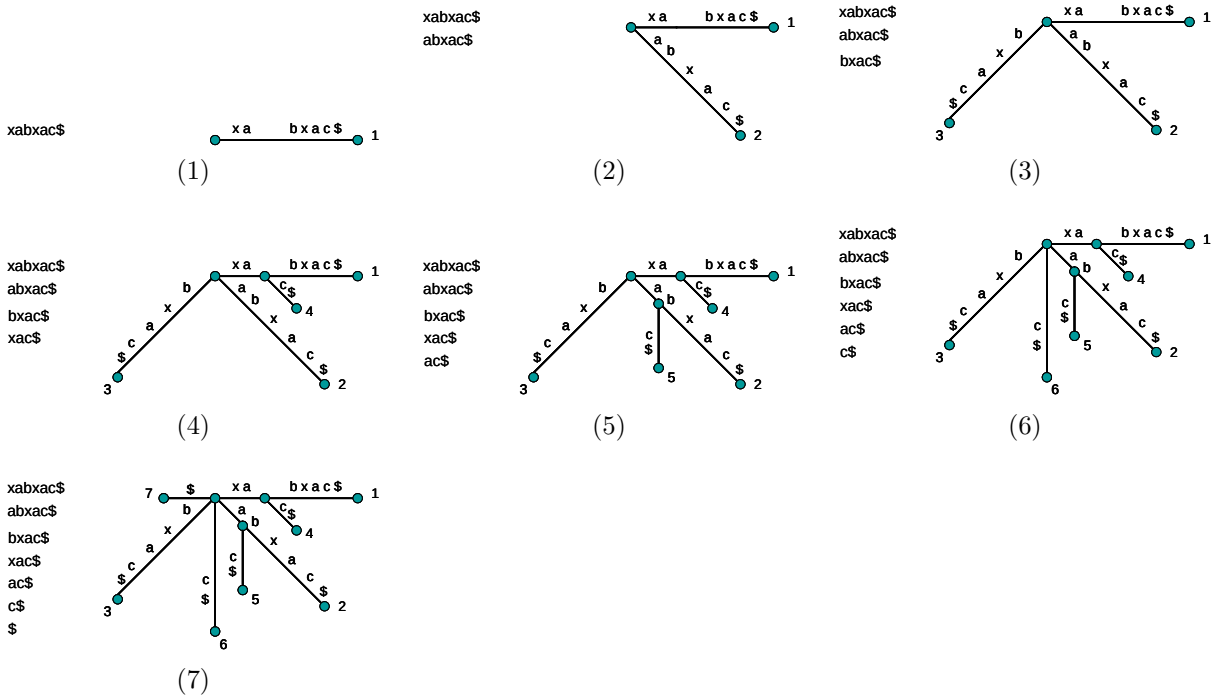
insert a special end marker \$ at the end of s
initialize the suffix tree T to one root

for $j = 1$ to $m + 1$
do
find the longest match of $x_j...x_m$ 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 $x_j...x_m\$$

It is easy to prove that the above algorithm produces a suffix tree for s by checking all the conditions that a suffix tree has to satisfy. We will leave the proof as a simple exercise. Here's the same example of $s = xabxac$ repeated, showing the above algorithm step by step.



Let us now analyze the time and space requirement of the above algorithm. The running time is $O(m^2)$ since each suffix requires an $O(m)$ time (dominated by the matching of successive characters) to update the suffix tree.

The space required by the above algorithm is $O(m^2)$ as well because each edge stores an $O(m)$ size label and we have $O(m)$ edges as described earlier (look at the properties of a suffix tree above). This space requirement can, however, be reduced to only $O(m)$ by not explicitly storing a label $x_i...x_j$, but implicitly by storing $[i, j]$, i.e. the indices marking the start and end of the label in s. Therefore, we have an $O(m^2)$ time and $O(m)$ space algorithm for building a suffix tree T for string s of length m.

There exists an $O(m)$ time algorithm (hence $O(m)$ space as well) for building a suffix tree. We are not going to describe it here because it is too complicated and not so important by itself. So we will assume the existence of a linear time and space algorithm for building a suffix tree.

But now that we have a suffix tree data structure, how can we use it efficiently to find all maximal matched between two strings x and y. Well, let us look at a simpler problem first which is a very famous problem in computer science: the string matching problem.

String Matching: Given a string x and a string y, find all locations in x where y occurs.

Here's an algorithm for string matching:

Algorithm

build a suffix tree T for x $O(m)$

match the characters of y along a unique path in T until
(case 1) either y is exhausted, or $O(n)$
(case 2) no more matches are possible

if (case 2), y does not occur in x

else the k leaves in the sub-tree below the point of the last match give the k location of y in x $O(k)$
(traverse the tree in linear time)

Why is the above algorithm correct? Simply because if y occurs in x at position i, then the i^{th} suffix of x must start with y. As a result, leaf i must be reached in T from the root by a path starting with y; therefore, all such leaves are in the subtree indentified by the algorithm.

The above algorithm runs in $O(m + n + k)$ where m is the length of x, n is the legnth of y, and k is the number of locations where y occurs in x. Since the input has size $O(m + n)$ and the output has size $O(k)$, this is optimal!

For obtaining the maximal matches of x and y, we are going to follow the same spirit of string matching by finding locations of substrings of y in x. But if we do it for every substring of y, then we spend $O(n^2(m + n)) = O(n^2m)$ time, and for each match we find we have to further check whether it is maximal or not. We will be more intelligent and check only for the suffixes of y hence spending $O(n(m + n)) = O(mn)$ time.

Here's what we are going to do: we are interested in matches of the form $x_i...x_{i+l-1} = y_j...y_{j+l-1}$ such that $x_i...x_{i+l} \neq y_j...y_{j+l}$ (cannot be extended to the right) and $x_{i-1}...x_{i+l-1} \neq y_{j-1}...y_{j+l-1}$ (cannot be extended to the left). For a suffix of y, $y_j...y_n$, we will find all matches starting at y_j that cannot be extended to the right. Then we will find a way to drop all those matches among them that can be extended to the left. Finally, doing this for all suffixes of y results in all maximal matches of x and y.

Here's how to find all matches starting at y_j that cannot be extended to the right:

- Build a suffix tree for x (we will do this only once)
- Find the path in T determined by the longest possible prefix of the suffix $y_j...y_n$ (it could stop in the middle of an edge e in that case e is part of the path)
- Let v_k, $k = 1..p$ be an internal node on this path and T_k be the subtree rooted at v_k that excludes v_{k+1}
- Identify the leaves in each subtree, each leaf corresponds to a location of a match that cannot be extended to the right

The following figure illustrates the idea:

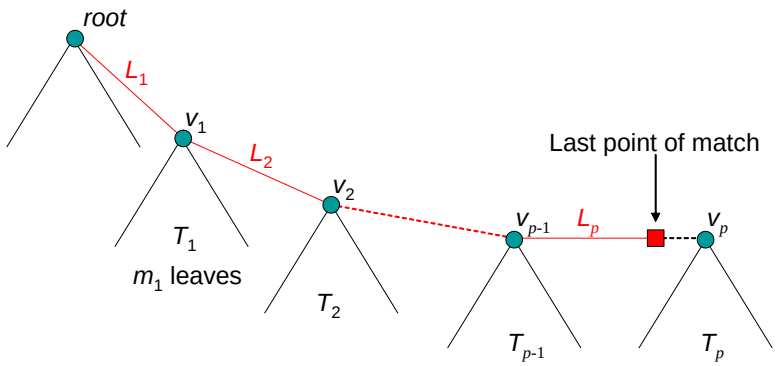


Figure 2: Finding matches starting at y_j that cannot be extended to the right

In the figure above, every leaf i in subtree T_k gives the location i in x of a match between $x_i...x_{i+|L_1|+...+|L_k|-1}$ and $y_j...y_{j+|L_1|+...+|L_k|-1}$ that cannot be extended to the right. Given the suffix tree for x, these can be found in $O(\sum |L_k| + \sum m_k) = O(n+m)$ time.

A match found above is not necessarily maximal since it is possible that we can extend it to the left. Therefore, for every leaf i we find, we have to check whether it corresponds to a maximal match or not.

Given a leaf i, let $left(i)$ be the character x_{i-1} (define x_0 and y_0 to be distinct special characters). When building the suffix tree for x, $left(i)$ for every leaf i can be stored at the leaf.

Now it is obvious that if $left(i) \neq y_{j-1}$, then i represents a maximal match, and this is a simple check.

Therefore, we obtain all maximal matches between x and y by repeating the above process for every suffix of y, resulting in an $O(mn)$ time algorithm.

Algorithm

build a suffix tree T for x $O(m)$

for $j = 1$ to n $O(n)$
do $O(m+n)$

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

let v_k, $k = 1..p$ be an internal node on this path and T_k be the subtree 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 subtree T_k, $k = 1..p$, such that $left(i) \neq y_{j-1}$

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

The running time of the algorithm above is $O(mn)$, hence we find all maximal matches of x and y in $O(mn)$ time. We did not fulfill our promise for a linear time algoritm. As we mentioned before, it is possible to find all maximal matches of x and y in $O(m+n+k)$ time, where k is the number of those matches. The following section explains how.

Generalized sufix tree

The main defficiency causing the $O(mn)$ time bound for the algorithm above summarizes in that we have to repeat the main matching process n times, once for every suffix of y. What if we could incorporate all suffixes of x as well as those of y in the same single suffix tree?, and do it once?, then we could possibly avoid the need to check for every suffix of y; since all the information will be compactly captured in one suffix tree. This is the main idea of this section.

A generalized suffix tree for strings s_1, s_2, ..., s_k is a suffix tree that stores all suffixes of s_1, all sufixes of s_2, ..., all suffixes of s_k.

Here's a simple algorithm to build such a generalized suffix tree for a set of strings based on a suffix tree algorithm for a single string.

- append a different end marker to each string in the set $\{s_1, ..., s_k\}$
- concatenate all the strings together to form a single string $s = s_1...s_k$
- build a suffix tree for the concatenated string s

The resulting suffix tree will have a leaf for each suffix **of the concatenated string** and is built in $O(|s_1| + ... + |s_k|)$ time, i.e. linear time in the size of all k strings. 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. Figure 3 shows the generalized suffix tree for $s_1 = xabxa$ and $s_2 = babxba$.

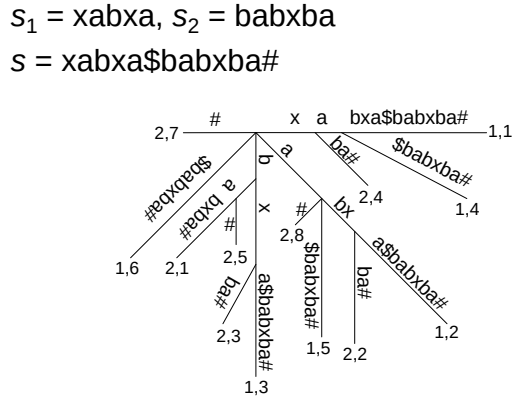


Figure 3: Generalized suffix tree for s_1 and s_2

As it might be obvious, a defect of this algorithm is that the tree now represents suffixes (of the concatenated string) that span more than one original string. These "synthetic" suffixes are not generally of interest. However, because end of string marker is distinct and is not in any of the original strings, the label on any path from the root to an internal node must be a substring of one of the original strings. Hence by reducing the second index of the label on leaf edges only, all unwanted synthetic suffixes are removed. Figure 4 illustrates the idea.

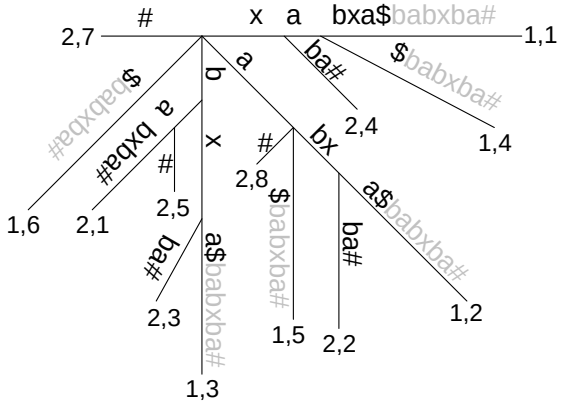


Figure 4: Fixing labels on leaf edges only

Therefore, given two strings x and y of length m and n respectively, we can build a suffix tree for both in $O(m+n)$ time. Each leaf in the tree represents either a suffix for x or a suffix for y. A leaf $(1,i)$ represents the i^{th} suffix of x and a leaf $(2,j)$ represents the j^{th} suffix of y.

Now mark each internal node v with x (y) if there is a leaf in the subtree of v representing a suffix of 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).

Lemma: 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. Therefore, there must be a node in the suffix tree that corresponds to a path labeled α from which there is an edge whose label starts with p leading to a leaf for x, and an edge whose label starts with q leading to a leaf for y. Hence, the node is an internal node marked with both x and y.

Conversely, if we have an internal node v labeled both x and y then let α be the concatenation of labels on the path from the root to v. There is a path from v to a leaf i for x and a path from v to a leaf j for y. If the two paths start with distinct edges e_1 and e_2, then we are done: let p be the first character of e_1's label and q be the first character of e_2 label, $p \neq q$. Then αp is a substring of x and αq is a substring of y.

So assume $e_1 = e_2 = e$ and hence $p = q$. But since v is a brancing node, there must be an edge f from v whose label starts with $w \neq p$. The edge f must lead to either a leaf for x or a leaf for y. If f leads to a leaf for x, then e leads to leaf j for y and we are done. If f leads to a leaf for y, then f leads to a leaf i for x and we are done.

Definition: 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_1s subtree and a leaf j for y in v_2s subtree, such that $left(i) \neq left(j)$.

Lemma: 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 the previous proof.

We call such an α a maximal common substring. Therefore, we have only $O(m+n)$ maximal common substrings for x and y represented by the left diverse nodes of the suffix tree (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 locations 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 describe the algorithm in detail here but you can refer to Gusfield D, Algorithms on Strings, Trees and Sequences pages 143-145, 147, Problem 37 page 173. First we present an algorithm for identifying left diverse nodes in linear time, then describe briefly how to modify this algorithm to obtain all maximal matches in $O(m+n+k)$.

The algorithm for identifying left diverse nodes is as follows. For each node v, the algorithm records two characters $a(v)$ and $b(v)$ such that the character $a(v)$ is:

- the left character of every leaf for x in v's subtree, or
- a special character ϵ if no leaf for x exists in v's subtree, or
- a special dinstinct character

The character $b(v)$ is defined similarly by replacing x above with y. Computing $a(v)$ and $b(v)$ can be done in a buttom up approach in linear time. Then it is obvious that v is left diverse iff it has two children v_1 and v_2 such that:

- $a(v_1) \neq b(v_2)$, $a(v_1) \neq \epsilon$, $b(v_2) \neq \epsilon$, or
- $b(v_1) \neq a(v_2)$, $b(v_1) \neq \epsilon$, $a(v_2) \neq \epsilon$

It takes $O(|\sum|^2)$ time (constant) to find two such children or none, where $\sum$ is the alphabet (each node has at most $|\sum|$ children).

To obtain all maximal matches we can maintain at each node v and for each character a of the alphabet:

- a list $L_{v,x,a}$ of leaves for x in v's subtree such that $left(i) = a$ for all $i \in L_{v,x,a}$
- a list $L_{v,y,a}$ of leaves for y in v's subtree such that $left(j) = a$ for all $i \in L_{v,y,a}$

Clearly a node v is left diverse iff if has two children v_1 and v_2 such that $L_{v_1,x,a} \neq \emptyset$ and $L_{v_2,y,b} \neq \emptyset$ and $a \neq b$.

Then for every (left diverse) node v and all characters $a \neq b$, the cartesian product of $L_{v_1,x,a}$ and $L_{v_2,y,b}$ for all children v_1 and v_2 of v gives the starting positions of all maximal matches corresponding to node v.

Building the lists at every node v can be done by merging all lists of its children (copying is no good since it will take $O(m+n)$ time at each node). This of course will destroy the children's lists, but luckily those lists are not needed once all maximal matches corresponding to v have been identified.

Therefore, building the lists can be done in a bottom up approach and takes linear time (assuming constant alphabet size). Moreover, the time for performing the cartesian product of lists is proportional to the number of maximal matches. Hence we have an $O(m+n+k)$ time algorithm.

References

Gusfield D., Algorithms on Strings, Trees, and Sequences, Chapters 5, 6, 7.