Problem Specifications

Following string matching, two more string analysis problems occur commonly in bioinformatics applications: repeated sequence detection and palindrome detection. Both problems are introduced below.

Maximal repeated pairs. A maximal pair of repeated strings or a maximal repeated pair in a string \( S = s_1 \ldots s_n \), is a pair of identical substrings \( P_1 = s_i \ldots s_{i+m} \) and \( P_2 = s_j \ldots s_{j+m} \), \( P_1 = P_2 \), which start at different positions in \( S \) (i.e., \( i \neq j \)), such that \( s_{i-1} \neq s_{j-1} \) and \( s_{i+m+1} \neq s_{j+m+1} \).

A maximal repeated pair can be represented as a triple \( \langle i, j, m \rangle \) where \( i \) and \( j \) are starting positions of the substrings \( P_1 \) and \( P_2 \) and \( m \) is the length of \( P_1 \) and \( P_2 \). Given a string \( S \), the set of all maximal repeated pairs is denoted \( R(S) \).

Palindromes. A string \( P = p_1 \ldots p_k \) of even length \( k \) is called a palindrome if \( p_1 = p_k, p_2 = p_{k-1}, \ldots, p_{k/2} = p_{1+k/2} \). A string of odd length \( P = p_1 \ldots p_k \) is a palindrome if \( p_1 \ldots p_{(k-1)/2}p_{(k-1)/2+1} \ldots p_k \) (i.e., an even-length string constructed out of \( P \) by taking out the mid-point character) is an even-length palindrome.

Meaningful palindromes exist in all human languages. Examples of palindromes in English are "dud", "madam", "never odd or even", "some men interpret nine memos", "don't nod", "may a moody baby doom a yam" and "no, it never propagates if I set a gap or prevention".

DNA palindromes. In a DNA, a palindrome definition is somewhat different. A complimented DNA palindrome is a string \( S = s_1 \ldots s_m \) in the \{A, T, C, G\} alphabet with the compliment relation defined as \( \text{compliment}(A) = T; \text{compliment}(T) = A; \text{compliment}(C) = G; \text{compliment}(G) = C \), where:

- if \( m \) is even: \( s_1 = s_m, s_2 = s_{m-1}, \ldots, p_{m/2} = p_{1+m/2} \).
- if \( m \) is odd: \( s_1 = s_m, s_2 = s_{m-1}, \ldots, p_{(m-1)/2} = p_{1+(m-1)/2} \).

Complimented palindromes play an important role in DNA: such sequences appear often in various regulatory DNA sequences. Also, complimented palindromes may form hairpin structures on a single DNA strand: nucleotides on
each strand, rather than binding to the complimentary nucleotide on the opposite strand bind to the complimentary nucleotide of the complimentary palindrome.

Often, palindromes in DNA come with gaps. A gapped complimentary DNA palindrome is a string $S = P_1QP_2$, such that $P_1P_2$ is a complimentary palindrome, and $Q = q_1 \ldots q_k$, and $q_1 \neq \text{compliment}(q_k)$.

A maximal palindrome substring in string $S = s_1 \ldots s_n$, is a string $P = s_i \ldots s_j$, such that $P$ is a palindrome, and $s_{i-1} \neq s_{j+1}$.

A maximal (gapped) complimentary DNA palindrome substring in string $S = s_1 \ldots s_n$ from the \{A, T, C, G\} alphabet is a string $P = s_i \ldots s_j$, such that $P$ is a (gapped) complimentary DNA palindrome, and $s_{i-1} \neq s_{j+1}$.

Maximal repeat detection problem. Given a string $S = s_1 \ldots s_n$ find all strings $P$ that are maximal repeated strings in $S$.

Example. Consider a string $S = \text{ATTGATTCATTC}$. This string has two maximal repeated strings: ATT and ATTC. ATT is represented by two triples: $\langle 1, 5, 3 \rangle$ and $\langle 1, 9, 3 \rangle$. ATTC is represented by a single triple $\langle 5, 9, 4 \rangle$. Note, that according to our definition of a maximal repeat, $\langle 5, 9, 3 \rangle$ does not form a maximal repeated sequence.

Note also, that despite the fact that ATT is a substring of ATTC, the output of an algorithm solving the maximal repeat detection problem must contain both.

Maximal palindrome detection problem. Given a string $S = s_1 \ldots s_n$, find all maximal palindromes in it.

Maximal DNA palindrome detection problem. Given a string $S = s_1 \ldots s_n$ in the \{A, T, C, G\} alphabet, find all maximal complimentary DNA palindromes in it.

Efficient Algorithm for Maximal Repeated String Detection

We use suffix trees to construct an efficient algorithm for determining all maximal repeated strings. The algorithm is based on the following observation:

Lemma (Maximal repeats). Let $T(S)$ be a suffix tree of a string $S$. Let $P$ be a maximal repeated string in $S$. Then there exists an internal node $v$ in $T(S)$ whose path label is exactly $P$.

The corollary to this lemma is useful for evaluation of our algorithm:

Theorem. A string $S$ of length $n$ can have no more than $n$ maximal repeats.

This is so, because there are no more than $n$ internal nodes in $T(S)$. 
Definition. Let $S = s_1 \ldots s_n$ be a string. A character $s_{i-1}$ is called the \textit{left character} for position $i$ in $S$. A node $v$ in $T(S)$ is called \textit{left diverse} if at least two leaves in $v$'s subtree have different left characters.

Example. Consider a string $S = \text{ATTCATT}$. The suffix tree $T(S)\$ is show in Figure ?? together with the left character (in a square box) for each leaf node (a.k.a, position in the string). Based on the definition of \textit{left diversity}, of the three internal nodes, $v$, $u$ and $w$, two, $v$ and $u$ are \textit{left diverse}, while $w$ is not.

Example. Consider the same string $S = \text{ATTCATT}$. We can see that there are two distinct maximal substrings in $S$: \text{ATT}, represented by a triple $\langle 1, 5, 3 \rangle$ and $\text{T}$, represented by triples $\langle 2, 3, 1 \rangle$, $\langle 6, 7, 1 \rangle$, $\langle 2, 7, 1 \rangle$ and $\langle 3, 6, 1 \rangle$.

These two substrings, correspond exactly to the path labels of two nodes in $T(S)$: \text{ATT} is a path label of node $v$, while $T$ is a path label of node $u$.

We notice that $v$ and $u$ are the two \textit{left diverse} nodes in $T(S)\$.

This is NOT a coincidence.

Theorem (left diversity). A string $P$ is a \textit{maximal repeated string} in a string $S$, iff the node in $T(S)$ with path label $P$ is \textit{left diverse}.

The set of all left \textit{maximal repeats} in a string $S$ can be represented as a subtree of $T(S)$ (or of $T(S)\$) which contains the paths to all the deepest (otherwise known as \textit{frontier}) left diverse nodes. This is a \textit{compact} representation, as it requires at most $n$ nodes and $n$ edges.

Finding \textit{left diverse nodes} in a suffix tree. We assume that the algorithm is given the string $S$, its size $n$ and the suffix tree $T(S)\$ as inputs. The algorithm operates as follows.

- Perform a depth-first search traversal of $T(S)\$.
- \textbf{Base case.} For each leaf node, record its \textit{left character}. 
• **Inductive step.** For each internal node $v$ examine the labels of its children.
  
  – If at least one child is labeled as *left diverse* then label $v$ as *left diverse*.
  
  – If no child is *left diverse*, then
    * If all child labels coincide (i.e., are the same), set the label of $v$ to be the same.
    * If at least two children have different labels then set the label of $v$ to be *left diverse*.

• **Pruning.** Delete from $T(S\$)$ all nodes that are NOT marked as *left diverse*. Return the remaining tree.

**Analysis.** The suffix tree $T(S\$)$ has the size $O(n)$, where $n$ is the length of $S$. Depth-first search traversal visits each node exactly once. There are $O(n)$ nodes with a parent in $T(S\$)$. The label of each such node is considered exactly once during the induction step. Any non-left diverse node in the tree can be deleted immediately after its label is considered during the induction step for its parent. Therefore, the running time of this algorithm is $O(n)$.

**Palindrome detection: preparation**

Before we introduce the palindrome detection algorithm, we need to discuss one more notion, *generalized suffix trees* and two problems, algorithms for which are an important part of our palindrome detection method: **lowest common ancestor (lca) detection in trees** and **longest common extension** problem.

We introduce and discuss these problems below.

**Generalized Suffix Tree**

**Definition.** A **generalized suffix tree** is a suffix tree representing all suffixes of a set of strings $S_1, \ldots, S_N$.

**Notes.** In practice, all strings represented in the generalized suffix tree are going to be $\$\$-terminated. Leaf nodes will now store multiple labels: one per string being processed. We encode each label as a pair: $(i, j)$, where $i$ is the ordinal representing the string, and $j$ is the ordinal representing the position in that string.

**Construction.** Informally, a generalized suffix tree for a sequence $S_1, \ldots S_N$ of strings can be constructed as follows:

1. Construct $T(S_1\$)$.
2. Starting with $T(S_1\$)$, traverse each suffix of $S_2\$}$ in it, and extend the tree where necessary. Add leaf labels to all leaf nodes you end at.
3. Repeat step 2 for $S_3, S_4, \ldots S_N$. 

4
Figure 2: Transforming a suffix tree into a generalized suffix tree.

Example. Figure ?? shows the construction of the generalized suffix tree for a pair of strings $S_1 = ATAT$ and $S_2 = TATT$. The top left tree is $T(S_1\$)$. On the top right tree, we performed insertion of the first suffix from $S_2$: $TATT$. (Arrow indicates new addition to the tree). The bottom tree is a full generalized suffix tree for the pair of strings $S_1$ and $S_2$.

Lowest Common Ancestor in Trees

Definition. In a tree $T$, the lowest common ancestor (lca) of two nodes $x$ and $y$ is the deepest node $z$ that is an ancestor to both $x$ and $y$.

The lca definition is illustrated in Figure ??.

Naïve algorithm. Start from $x$ and $y$, trace their ancestry in parallel until you hit a node that’s the same.

Efficient algorithm. With some special preparation, the lca between two nodes in a tree can be found in constant time.

For complete binary trees, this is a matter of clever marking of the nodes with binary codes, and a bit-wise XOR operation between the codes of the two nodes.

For arbitrary trees, a mapping $I$ from the nodes of the tree to the nodes of a complete binary tree can be developed, with the property that $I(lca(u), lca(v)) = lca(I(u), I(v))$.

Longest Common Extension

Problem definition. Given two strings $S_1 = s_1 \ldots s_m$, $S_2 = t_1 \ldots t_n$, and two numbers $i, j$, the longest common extension of $S_1$ at $i$ and $S_2$ at $j$ is a
Figure 3: Illustrating the definition of lowest common ancestor: node z is the \( lca \) for nodes x and y.

string \( P = p_1 \ldots p_k \), such that \( s_i \ldots s_{i+k} = t_j \ldots t_{j+k} = P \), but \( s_{i+k+1} \neq t_{j+k+1} \) (or either \( i+k = m \) or \( j+k = n \)).

Efficient algorithm. The longest common extension of two strings at two positions can be computed in constant time with linear pre-processing using the following algorithm:

1. Create a generalized suffix tree \( T(S_1, S_2) \) and process it to allow \( lca \) queries.
2. Find nodes (leaves) \( v_i \) and \( v_j \) in \( T(S_1, S_2) \) representing the suffixes \( s_i \ldots s_m \) and \( t_j \ldots t_n \).
3. Find \( u = lca(v_i, v_j) \). Return the path label for \( u \).

Palindrome detection

Idea. Let \( S = s_1 \ldots s_n \). Consider the string \( S' = s_n \ldots s_1 \), i.e., \( S' = reverse(S) \). The idea behind the linear-time algorithm for even-length palindrome detection is based on the following observation:

Let \( S \) contain an even-length palindrome centered immediately after character \( s_q \). Let the radius of this palindrome be \( k \). Then the \( k \) characters starting at position \( n - q + 1 \) in string \( S' \) are identical to \( s_q \ldots s_{q+k} \).

Example. This is illustrated in figure ???. We consider a string \( S = ATCAACTGAT \). It has a palindrome \( TCAACT \) centered right after position \( q = 4 \). The reverse of \( S \), \( S' = TAGTCAACTA \). Reverses preserve the palindrome. The three-letter extension of \( S \) at position \( q + 1 = 5 \) is \( ACT \). Similarly, the three-letter extension at position \( n - q + 1 = 7 \) of \( S' \) is \( ACT \) - the reverse of the first half of the palindrome in \( S \).
Figure 4: Illustrating the key idea of the palindrome detection algorithm for even-length palindromes. A palindrome can be found as the maximal common extension in $S$ and $S' = reverse(S)$ at positions $q + 1$ and $n - q + 1$.

Figure 5: Illustrating the key idea of the palindrome detection algorithm for odd-length palindromes. A palindrome can be found as the maximal common extension in $S$ and $S' = reverse(S)$ at positions $q$ and $n - q + 1$.

Figure 6: Illustrating the key idea of the palindrome detection algorithm for complimentary DNA palindromes. $S' = reverse(compliment(S))$. 
Algorithm. We propose the following algorithm for even-length palindrome detection.

1. Given a string \( S \), construct \( S' = \text{reverse}(S) \).
2. Construct the generalized suffix tree \( T(S, S') \).
3. For \( q := 1 \) to \( n - 1 \) do:
   (a) Let \( x = \text{LongestCommonExtension}((S : q + 1), (S' : n - q + 1)) \).
   (b) If \( x > 0 \), there is a palindrome of length \( 2x \) centered right after \( q \) in \( S \).

Odd-length palindromes. For odd-length palindromes, we note the if there is a palindrome in \( S \) centered on position \( q \), then the maximal extension of \( (S : q) \) and \( (S' : n - q + 1) \) is going to be equal to the central character of the palindrome followed by the "wing" of the palindrome (Figure ??).

To detect odd-length palindromes, therefore, we check for \( \text{LongestCommonExtension}((S : q), (S' : n - q + 1)) \) and ensure that its length is greater than 1.

Complimentary DNA palindromes. To detect complimentary DNA palindromes instead of taking \( S' = \text{reverse}(S) \), we set \( S = \text{reverse}(\text{compliment}(S)) \). Then continue as before. Figure ?? illustrates this idea.

References