6

Suffix Tree Applications in Computational Biology

Pang Ko
Iowa State University

Srinivas Aluru
Iowa State University

6.1 Introduction

In recent years the volume of biological data has increased exponentially. Concomitantly, the speed with which such data is generated has increased as well. It is now possible to sequence a bacterial genome in a single day. Thus efficient data structures are needed to archive and retrieve biological data. Furthermore, this explosion of data has increased the need to analyze a large amount of data in a reasonable time. With the availability of complete genomes, researchers have begun to compare whole genomes [4, 11, 26, 27]. This further increases the scale of problems addressed, and algorithms that worked well for smaller scale problems are either insufficient or inappropriate. For example, dynamic programming techniques worked well to identify the matching regions between two genes. However, heuristics must be applied when we try to identify highly conserved regions between two genomes in reasonable time and space. Suffix trees can serve as an efficient data structure to analyze DNA and protein sequences. They can also be used to provide exact matches efficiently, which many heuristics depend on.

Computationally, both DNA and protein sequences can be modeled as strings of characters. But unlike natural languages where there are well-defined sentence structures and word boundaries, DNA and protein sequences have no such properties. This makes the traditional approaches of using inverted tables and hash tables less appealing. Suffix trees and generalized suffix trees, the multiple string variant of suffix trees, can be used to solve a number of computational biology related problems in optimal space and time. In this chapter we examine several applications of suffix trees in computational biology. For the most part, our focus will be on solving problems motivated by real applications in molecular biology. In many cases, the algorithms presented here are part of actual bioinformatic software programs developed, illustrating the practical role of suffix trees in computational biology research. We use the same terminology as in the previous chapter where suffix trees and suffix arrays are introduced.

6.2 Basic Applications

In this section, we provide a brief introduction to the pattern matching capabilities of suffix trees. Although pattern matching by itself may not directly correspond to many computational biology applications, it is a basic building block upon which many suffix tree algorithms are founded. Besides, the underlying ideas are frequently used as components within more complicated algorithms, and in some cases they are modified and used in software with vastly different objectives.

6.2.1 Pattern Matching

Given a pattern P and a text T, the pattern matching problem is to find all occurrences of P in T. Let $|P| = m$ and $|T| = n$. Typically, $n >> m$. Moreover, T remains fixed in many applications and the query is repeated for many different patterns. For example, T could be an entire database of DNA sequences and P denotes a substring of a query sequence for homology (similarity) search. Thus, it is beneficial to preprocess the text T so that queries can be answered as efficiently as possible.

The pattern matching problem can be solved in optimal $O(m + k)$ time using $ST(T)$, where k is the number of occurrences of P in T. Suppose P occurs in T starting from position i. Then, P is a prefix of $suff_i$ in T. It follows that P matches the path from root to leaf labeled i in ST. This property results in the following simple algorithm: Start from the root of ST and follow the path matching characters in P, until P is completely matched or a mismatch occurs. If P is not fully matched, it does not occur in T. Otherwise, each leaf in the subtree below the matching position gives an occurrence of P. The positions can be enumerated by traversing the subtree in time proportional to the size of the subtree. As the number of leaves in the subtree is k, this takes $O(k)$ time. If only one occurrence is of interest, the suffix tree can be preprocessed in $O(n)$ time such that each internal node contains the label of one of the leaves in its subtree. Thus, the problem of whether P occurs in T or the problem of finding one occurrence can be answered in $O(m)$ time.

6.2.2 Approximate Pattern Matching

The simpler version of approximate pattern matching problem is as follows: Given a pattern P ($|P| = m$) and a text T ($|T| = n$), find all substrings of length $|P|$ in T that match P with at most k mismatches. To solve this problem, first construct the GST of P and T. Preprocess the GST to record the string-depth of each node, and to answer lca queries in constant time. For each position i in T, we will determine if $T[i..i+m-1]$ matches P with at most k mismatches. First, use an lca query $lca((P,1),(T,i))$ to find the largest substring from position i of T that matches a substring from position 1 of P. Suppose the length of this longest exact match is l. Thus, $P[1..l] = T[i..i+l-1]$, and $P[l+1] \neq T[i+l]$. Count this as a mismatch and continue by finding $lca((P,l+2),(T,i+l+1))$. This procedure is continued until either the end of P is reached or the number of mismatches crosses k. As each lca query takes constant time, the entire procedures takes $O(k)$ time. This is repeated for each position i in T for a total run-time of $O(kn)$.

Now, consider the more general problem of finding the substrings of T that can be derived from P by using at most k character insertions, deletions or substitutions. To solve this problem, we proceed as before by determining the possibility of such a match for every starting position i in T. Let $l = \textit{string-depth}(lca((P,1),(T,i)))$. At this stage, we consider three possibilities:

1. Substitution − $P[l+1]$ and $T[i+l]$ are considered a mismatch. Continue by finding $lca((P,l+2),(T,i+l+1))$.
2. Insertion − $T[i+l]$ is considered an insertion in P after $P[l]$. Continue by finding $lca((P,l+1),(T,i+l+1))$.
3. Deletion − $P[l+1]$ is considered a deletion. Continue by finding $lca((P,l+2),(T,i+l))$.

After each lca computation, we have three possibilities corresponding to substitution, insertion and deletion, respectively. All possibilities are enumerated to find if there is a sequence of k or less operations that will transform P into a substring starting from position i in T. This takes $O(3^k)$ time. Repeating this algorithm for each position i in T takes $O(3^k n)$ time.

The above algorithm always uses the longest exact match possible from a given pair of positions in P and T before considering the possibility of an insertion or deletion. To prove the correctness of this algorithm, we show that if there is an approximate match of P starting from position i in T that does not use such a longest exact match, then there exists another approximate match that uses only longest exact matches. Consider an approximate match that does not use longest exact matches. Consider the leftmost position j in P and the corresponding position $i+l'$ in T where the longest exact match is violated. i.e., $P[j] = T[i+l']$ but this is not used as part of an exact match. Instead, an insertion or deletion is used. Suppose that an exact match of length r is used after the insertion or deletion. We can come up with a corresponding approximate match where the longest match is used and the insertion/deletion is taken after that. This will either keep the number of insertions/deletions the same or reduce the count. Thus, if the value of k is small, the above algorithms provide a quick and easy way to solve the approximate pattern matching problem. For sophisticated algorithms with better run-times, see [9, 30].

6.3 Restriction Enzyme Recognition Sites

Restriction endonucleases are enzymes that recognize a particular pattern in a DNA sequence and cleave the DNA at or near the recognition site. The enzyme typically cuts both strands of double stranded DNA and the recognition sequence is often a short sequence that is identical on both the strands. Recall that due to opposite directionality of the two strands, the sequences are read in opposite directions relative to each other. Thus, the recognition sequence is what is called a *complemented palindrome*; by reversing the sequence and using complementary substitutions A $\leftrightarrow$ T, and C $\leftrightarrow$ G, one would obtain the sequence itself. As an example, the restriction enzyme *SwaI* recognizes the site ATTTAAAT and cleaves it in the center of the pattern. The restriction enzyme *BamHI* detects the sequence GGATCC and cleaves it after the first base (and similarly after the first base in the complementary strand sequence; the first base in the complementary strand is paired with the last base in the original strand). Most restriction enzymes are derived from bacteria and are named after the organism in which they are first discovered. Restriction enzymes play a defense role by cleaving foreign DNA. The DNA of the host organism is protected by mythelation of its own recognition sites, which makes it immune to restriction enzyme activity.

Consider the problem of finding all complemented palindromic sequences in a given long DNA sequence. We focus on the problem of identifying all maximal complemented palindromes, as all other palindromes are contained in them. Formally, a substring $s[i..j]$ of a string s of length n is called a *maximal complemented palindrome* of s, if $s[i..j]$ is a complemented palindrome and $s[i-1]$ and $s[j+1]$ are not complementary bases, or $i = 1$, or $j = n$. Note that a maximal complemented palindrome must necessarily be of even length. For a palindrome of length $2k$, define the center to be the position between characters k and $k+1$ of the palindrome. The palindrome is said to be of radius k. Starting from the center, a complemented palindrome is a string that reads the same in both directions subject to complementarity. Observe that each maximal palindrome in a string must have a distinct center. As the number of possible centers for a string of length n is $n-1$, the total number of maximal palindromes of a string is $n-1$. All such palindromes can be identified in linear time using the following algorithm.

Let s^r denote the reverse complement of string s. Construct a GST of the strings s and s^r and preprocess the GST to record string depths of internal nodes and for answering lca queries. The maximal even length palindrome centered between $s[i]$ and $s[i+1]$ is given by the length of the longest common prefix between $suff_{i+1}$ of s and $suff_{n-i+1}$ of s^r. This is computed as the string-depth of $lca((s, i+1), (s^r, n-i+1))$ in constant time. Thus all maximal complemented palindromes can be recognized in $O(n)$ time. An example to illustrate this algorithm is presented in Figure 6.1. The figure shows the generalized suffix tree of the DNA sequence TAGAGCTCA and its reverse complement TGAGCTCTA.

6.4 Detection of RNAi Elements

RNA interference (RNAi) is a process that utilizes a double stranded RNA (dsRNA) molecule to inhibit the expression of a particular gene by binding to its mRNA. This process was first discovered by Fire *et al.* [14]. Since then, RNAi has been used as an alternative to gene knockout experiments. Unlike traditional experiments where a gene is permanently removed from the genome, researchers can choose when and where to introduce the dsRNA. This gives biologists greater flexibility in experimental design. It has been shown that RNAi is also used in cells as a way to regulate gene expression, and as a defense mechanism against viruses.

Unlike DNA molecules which are double stranded helixes, RNA molecules are usually single stranded and have a secondary structure that sometimes has important functions. Like all RNA molecules, naturally occurring RNAi elements are also produced by transcription from a corresponding genomic sequence. The transcription produces an RNA molecule that contains a sequence and its reverse complement separated by a short sequence. The reverse complementarity causes the sequences to bind to each other with the short sequence in the middle forming a stem-loop-stem structure. Cleaving of this structure results in dsRNA (see Figure 6.2). The resulting dsRNA will then interact with target messenger RNAs (mRNAs) to prevent them from being translated into proteins, thus controlling gene expression.

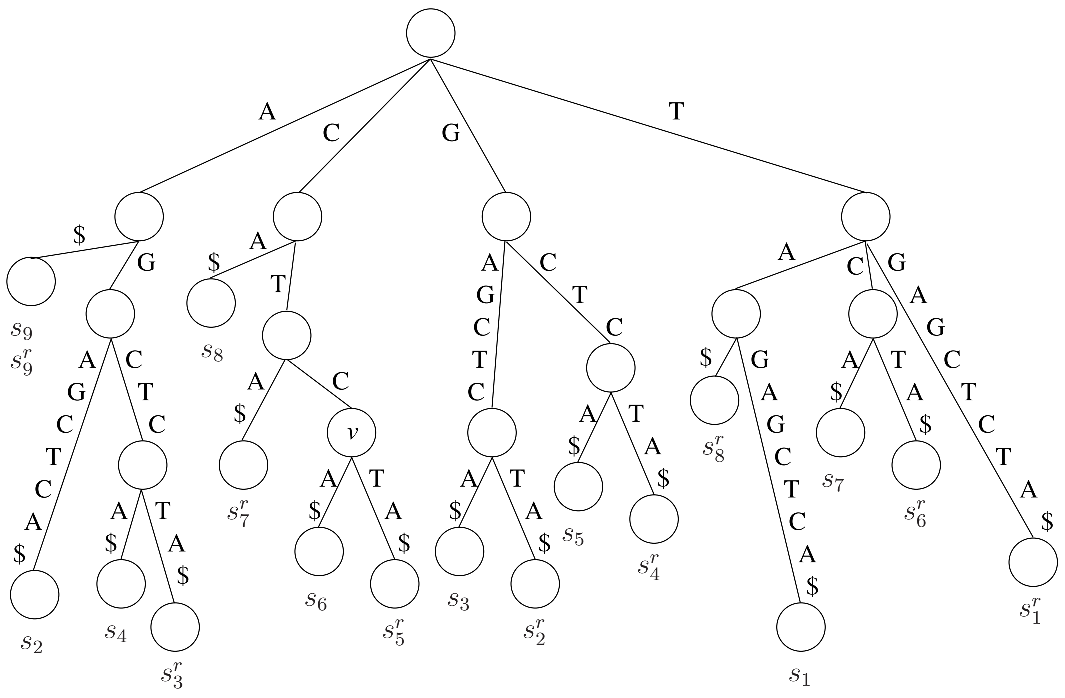


FIGURE 6.1: The generalized suffix tree of the DNA sequence s = TAGAGCTCA and its reverse complement s^r = TGAGCTCTA. For $i = 5$, $v = lca((s, 6), (s^r, 5))$, revealing the maximal complemented palindrome GAGCTC.

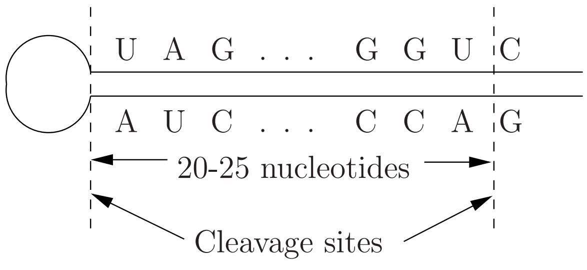


FIGURE 6.2: An example of RNAi element — the stem-loop-stem structure forming a double stranded RNA (dsRNA) that is usually about 20 nucleotides in length.

Horesh *et al.* [19] present a suffix tree based algorithm to detect RNAi elements from a genomic sequence. Here we present how suffix trees can be used to detect such patterns, while avoiding the more complex details necessary for accurate biological modeling.

We can identify RNAi elements in a genome by identifying substrings s_1 and s_2 of the same length (about 20 to 25 nucleotides) that are reverse complements of each other, and separated by a substring s_3 of length $l \leq k$. The parameter k is used to avoid detecting substring and reverse complement pairs separated by great distances. To do this, first a generalized suffix tree is built for the input genomic sequence, and its reverse complement. Let

$s_1s_3s_2$ be a substring in the genomic sequence such that s_1 and s_2 are reverse complements of each other, and s_3 is the loop. Then in the generalized suffix tree we just constructed, there is an internal node v that contains two leaves in its subtree corresponding to suff_i and suff_{n-i-l} of the reverse complement sequence, where $l \leq k$ is the length of s_3 (k is a pre-selected threshold); and v is the lowest common ancestor of the two leaves.

The straightforward solution in this case is to traverse the entire tree; for each internal node at a string depth of about 20 to 25, find pairs of leaves that satisfy the criterion mentioned above. However for each leaf representing some suffix suff_i in the original sequence, checking whether there is a leaf corresponding to suffix suff_{n-i-l} of the reverse complement sequence is not easy. Done naively, this could take $O(n^2)$ time, because for each leaf we need to scan all the leaves in the subtree to check whether a suitable counterpart exists.

Here we make the observation that a post-order traversal of the suffix tree induces a complete ordering of all the suffixes. Each suffix in the original sequence and the reverse complement sequence can be associated with a rank, which can be easily obtained by a traversal of the suffix tree. For each internal node v two values ℓ_v and $\ell\ell_v$ are calculated. The value ℓ_v is the number of leaves in the subtree rooted at v. The value $\ell\ell_v$ is the number of leaves in the entire tree to the left of v, i.e., the number of leaves visited in a post-order traversal before visiting v. The ℓ values for all internal nodes can be calculated using post-order traversal as follows: When internal node v is visited, add up ℓ_u of all the nodes u, such that u is a child of v. If a node w is a leaf node then $\ell_w = 1$. To calculate the $\ell\ell$ values, define $\ell\ell_{root} = 0$. Then for each internal node v, $\ell\ell_v = \ell\ell_u + \sum_w \ell_w$, where u is the parent of v and w ranges over all the siblings to the left of v. In other words, the number of leaves to the left of a node v is the number of leaves to the left of its parent plus the total number of leaves in all the subtrees rooted at its siblings to the left. The $\ell\ell$ values can be calculated using a pre-order traversal of the tree.

Consider an internal node v whose string depth is in the target range, say 20 to 25, and a suffix suff_i in the subtree rooted at v. To check whether there is a leaf corresponding to suffix suff_{n-i-l} of the reverse complement sequence in the same subtree, one can scan the ranks of suffix suff_{n-i-1} to suffix suff_{n-i-k} of the reverse complement sequence. Suppose the rank of suffix suff_{n-i-j} is between $\ell\ell_v$ and $\ell\ell_v + \ell_v$, then we know that this suffix is in the subtree rooted by v. If the ith suffix of the original sequence and the $(n-i-j)$th suffix of the reverse complement sequence appear in the subtrees of two different children of v, then v is the lowest common ancestor of the two leaves. The path label of v is a potential dsRNA sequence.

This algorithm takes $O(nk)$ time, where n is the length of the genomic sequence, and k is the maximum length allowed for the stem-loop-stem structure. A more biologically sensible model can be used to take into account the fact that the two strands need not be identical, either because it is enough to have high sequence similarity, or due to potential sequencing errors. An algorithm allowing mismatches on the two strands can also be found in [19]. However, due to the complexity of the model, the algorithm is close to a brute force algorithm.

6.5 Sequence Clustering and Assembly

DNA sequence clustering and assembling overlapping DNA sequences are vital to knowledge discovery in molecular biology. Part III of this handbook is devoted to assembly and clustering applications, and the reader will once again find that suffix trees are used in some of the algorithms presented in that part. In this section, we discuss two suffix tree related problems that are motivated by applications in clustering and assembly. The problems presented in this section are rather artificial, as they are applicable only in the case input data does not contain any errors or genetic variations. Nevertheless, these problems will serve to develop a basic understanding of some of these applications, and suffix tree based algorithms for real clustering and assembly applications can be found in Part III.

6.5.1 Sequence Containment

One problem that is encountered in sequence clustering and assembly applications is redundancy in the input data. Consider a set $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of DNA sequences. We wish to identify sequences that are completely contained in other sequences and remove them. In the absence of sequencing errors and other types of variations (such as the DNA sequences being derived from different individuals who may have natural genetic variations), this can be abstracted as the *string containment* problem. Given a set $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of strings of total length N, the string containment problem is to identify each string that is a substring of some other string. This problem can be easily solved using suffix trees in $O(N)$ time. First, construct the $GST(\mathcal{S})$ in $O(N)$ time. To find if a string s_i is contained in another, locate the leaf labeled $(s_i, 1)$. If the label of the edge connecting the leaf to its parent is labeled with the string '\$', s_i is contained in another string. Otherwise, it is not. This can be determined in $O(1)$ time per string.

6.5.2 Suffix-Prefix Overlaps

The suffix-prefix overlap problem arises in genome assembly problems. At the risk of oversimplification, the problem of genome assembly is to construct a long, target DNA sequence from a large sampling of much shorter fragments of it. This procedure is carried out to extend the reach of DNA sequencing, which can be directly carried only for DNA sequences hundreds of nucleotides long. The first step in assembling the many fragments is to detect pairs of fragments that show suffix-prefix overlaps; i.e., identify pairs of fragments such that the suffix of one fragment in the pair overlaps the prefix of the other fragment in the pair. The suffix-prefix overlaps are then used to assemble the fragments into longer DNA sequences.

Suppose we are given a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of total length N. In the absence of sequencing errors, the suffix-prefix overlap problem is to identify, for each pair of strings (s_i, s_j), the longest suffix of s_i that is a prefix of s_j. This problem can be solved using $GST(\mathcal{S})$ in optimal $O(N + k^2)$ time. Consider the longest suffix α of s_i that is a prefix of s_j. In $GST(\mathcal{S})$, α is an initial part of the path from the root to leaf labeled $(s_j, 1)$ that culminates in an internal node. A leaf that corresponds to a suffix from s_i should be a child of the internal node, with the edge label '\$'. Moreover, it must be the deepest internal node on the path from root to leaf $(s_j, 1)$ that has a suffix from s_i attached in this way. The length of the corresponding suffix-prefix overlap is given by the string depth of the internal node.

Let M be a $k \times k$ output matrix such that $M[i, j]$ should contain the length of the longest suffix of s_i that overlaps a prefix of s_j. The matrix is computed using a depth first search (DFS) traversal of $GST(\mathcal{S})$. During the DFS traversal, k stacks $A_1, A_2, \ldots, A_k$ are maintained, one for each string. The top of the stack A_i contains the string depth of the deepest node along the current DFS path that is connected with edge label '\$' to a leaf corresponding to a suffix from s_i. If no such node exists, the top of the stack contains zero. Each stack A_i is initialized by pushing zero onto an empty stack, and is maintained during the DFS as follows: When the DFS traversal visits a node v from its parent, check to see if v is attached to a leaf with edge label '\$'. If so, for each i such that string s_i contributes a suffix labeling the leaf, *push string-depth*(v) on to stack A_i. The string depth of the current node can be easily maintained during the DFS traversal. When the DFS traversal leaves the node v to return back to its parent, again identify each i that has the above property and *pop* the top element from the corresponding stack A_i.

The output matrix M is built one column at a time. When the DFS traversal reaches a leaf labeled $(j, 1)$, the top of stack A_i contains the longest suffix of s_i that matches a prefix of s_j. Thus, column j of matrix M is obtained by setting $M[i, j]$ to the top element of stack S_i. To analyze the run-time of the algorithm, note that each *push* (similarly, *pop*) operation on a stack corresponds to a distinct suffix of one of the input strings. Thus, the total number of *push* and *pop* operations is bounded by $O(N)$. The matrix M is filled in $O(1)$ time per element, taking $O(k^2)$ time. Hence, all suffix-prefix overlaps can be identified in optimal $O(N + k^2)$ time.

The above solutions for sequence containment and suffix-prefix overlap problems are not useful in practice because they assume a perfect input free of errors and genetic variations. In practice, one is interested in detecting strong homologies rather than exact matches. The reader interested in how suffix trees can be used for such applications is referred to Chapter 13.

6.6 Whole Genome Alignments

With the availability of multiple genomes, the field of comparative genomics is gaining increasing attention. By comparing the genomic sequences of two closely related species, one can identify potential genes, coding regions, and other genetic information preserved during evolution. On the other hand, by comparing the genomic sequences of distantly related species, one might be able to identify genes that are most likely vital to life. Several programs have been developed to identify such “local” regions of interest [1, 4, 7, 28]. An important problem in comparative genomics is whole genome comparison, i.e., a global or semi-global alignment of two genomes. This allows researchers to understand the genomic differences between the two species. This is particularly useful in comparing two strains of the same virus or bacteria, or even two versions of the assembly of the genome of the same species. We describe a suffix tree based approach for whole genome alignments, as utilized in the popular whole genome alignment tool *MUMmer*, developed by Delcher *et al.* [11, 12]. A suffix array based solution for the same problem is presented in the next chapter.

The *MUMmer* program is based on the identification of maximal unique matches (MUM-s). A maximal match between strings s_1 and s_2 is a pair of matching substrings $s_1[i..i+k] = s_2[i'..i'+k] = \alpha$, that cannot be extended in either direction, i.e. $s_1[i-1] \neq s_2[i'-1]$ and $s_1[i+k+1] \neq s_2[i'+k+1]$. A maximal unique match implies that the pair of matching substrings is not only maximal, but also unique; i.e., the substring α is maximal, and occurs exactly once in each s_1 and s_2. A long MUM is very likely to be in the optimal alignment of two sequences. The program has the following stages:

1. Find all MUMs between the two sequences.
2. Find the longest sequence of MUMs, that occur in the same order in either sequence.
3. Align the regions between the MUMs.

To illustrate the use of suffix trees in whole genome alignment, we focus on identification of MUMs, a step that utilizes suffix trees. Given two strings s_1 and s_2, assume the last characters of s_1 and s_2 are $\$_1$ and $\$_2$, respectively, characters that do not occur anywhere else in both strings. First build the $GST(\{s_1, s_2\})$ of the two strings. Let $suff_i^j$ denote

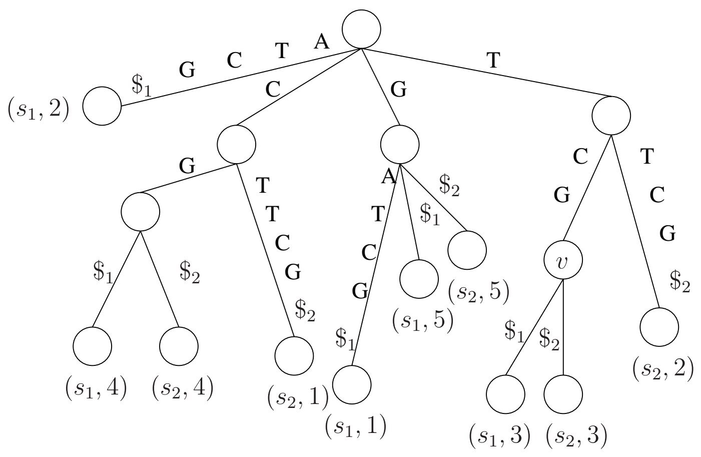


FIGURE 6.3: The generalized suffix tree of sequences GATCG$\$_1$ and CTTCG$\$_2$. Each leaf label is a tuple identifying the string number, followed by the position of the corresponding suffix within the string. In this example, the path-label of node v, is the MUM TCG.

the suffix of string s_j starting at position i. $Lcp(suff_i^1, suff_{i'}^2)$ is a MUM if and only if it is unique in both sequences and $s_1[i-1] \neq s_2[i'-1]$. Let u be the internal node with path label $lcp(suff_i^1, suff_{i'}^2)$. Then the uniqueness part implies u must have exactly two children, the leaves corresponding to suffixes $suff_i^1$ and $suff_{i'}^2$. To ensure left maximality, we need to compare the left characters of the two suffixes under u to make sure that they are unequal. Thus all internal nodes corresponding to MUMs can be identified in a traversal of the $GST(\{s_1, s_2\})$. An example of MUM identification is shown in Figure 6.3.

The space required for the algorithm can be considerably reduced by building the suffix tree of only one string, say s_1, and streaming the other string s_2 to identify MUMs [12]. The algorithm works by considering all suffixes of s_2 starting from $suff_1^2$.

- Find the longest possible match in the suffix tree for $suff_1^2$, the first suffix of string s_2. This is done by traversing from the root of the suffix tree and matching consecutive characters of $suff_1^2$ until no further matches are possible.
- If the match ends inside the edge label between an internal node and a leaf node, then check the left character in s_1 of the suffix corresponding to the leaf and the left character of the suffix from s_2. If they are not the same, then the match is reported.
- After finishing with the first suffix, the same method can be repeated with the second suffix $suff_2^2$. But instead of starting from the root and matching the suffix, suffix links are used to shortcut the process. Let u be the last internal node encountered while matching the previous suffix. Then, take the suffix link from u to, say, u'. Suppose the previous suffix match ended l characters away from node u. It is guaranteed that these l characters will match a path below u'. Therefore, these characters can be matched at the rate of constant time per edge using the same technique as employed in McCreight's suffix tree construction algorithm (Chapter 5). Once the end of these l characters is reached, further matching will continue by examining the subsequent characters of $suff_2^2$ one by one.

- Repeat the process for all suffixes of the second string.

The above algorithm correctly reports all the maximal matches between the two strings. However uniqueness is not preserved for the second string, because we do not actually insert the suffixes of the second string. For example, if we build the suffix tree for the string ATGACGGTCCT$\$_1$, and subsequently stream the second string ATGATGAG$\$_2$, then the substring ATGA will be reported twice. This streaming algorithm also runs in $O(n)$ time, because building the suffix tree for the first string takes $O(|s_1|)$ time, and streaming of the second string is equivalent to inserting all suffixes of it using McCreight's algorithm, which takes $O(|s_2|)$ time.

6.7 Tandem Repeats

Tandem repeats — segments of short DNA repeated multiple times consecutively — are believed to play a role in regulating gene expression. Tandem repeats also have a much higher rate of variation then the rest of the genome (in terms of the number of copies), and this makes them ideal markers to distinguish one individual from another.

A tandem repeat can consist of anywhere from two to hundreds of repetitions. If we have the ability to detect a 2-repeat tandem sequence, it can be used to deduce tandem sequences with more repeats. Therefore, the problem is modeled by defining a tandem repeat to be a string $\beta = \alpha\alpha$, i.e., a consecutive occurrence of two copies of the same string α. Tandem repeats are further divided into two sub-categories, primitive and non-primitive. String β is called a primitive tandem repeat if it does not contain another tandem repeat. For example, strings aa and $abab$ are primitive tandem repeats, while $aaaa$ is not a primitive tandem repeat. A 2-repeat tandem sequence is sometimes referred to as a square. When a substring α repeats more than twice consecutively, it is sometimes referred to as a tandem array. A tandem repeat/array in string s given by $\beta = \alpha^k = s[i..i+k|\alpha|-1]$, where $|\alpha|$ is the length of the substring α, is represented as a triple (i, α, k). We can also represent a tandem repeat $\beta = \alpha\alpha$ as a tuple $(i, 2|\alpha|)$. We use the notation that best suits the situation we are describing.

Detection of tandem repeats is a well-studied problem in computational biology. Crochemore presented an algorithm that computes all occurrences of primitive tandem repeats in $O(n \log n)$ time [3, 10]. On the other hand, all occurrences of tandem repeats (both primitive and non-primitive) can be found in $O(n \log n + occ)$ [24, 29], where occ is the number of occurrences of tandem repeats in the string. We first present a simple $O(n \log n + occ)$ algorithm due to Stoye and Gusfield [29].

6.7.1 Stoye and Gusfield's $O(n \log n)$ Algorithm

Consider a tandem repeat in string s starting at position i of the form $s[i..2|\alpha|+i-1] = \alpha\alpha$, and $\alpha = a\gamma$, where a is the first character of α and γ is the remainder of α. If character $s[2|\alpha|+i] = x \neq a$, then in the suffix tree there is an internal node v at string depth $|\alpha|$, and suffix $suff_i$ and suffix $suff_{|\alpha|+i-1}$ will be in the subtrees of two different children of v. Since the two suffixes branch, the tandem repeat is referred to as a branching tandem repeat. An example is shown in Figure 6.4.

If a tandem repeat $(i, a\gamma, 2)$ is not a branching tandem repeat, then $(i+1, \gamma a, 2)$ is also a tandem repeat. However, $(i+1, \gamma a, 2)$ may not be a branching tandem repeat either. This property of non-branching tandem repeats is easy to see; if $s[i..2|\alpha|+i-1] = a\gamma a\gamma$ is a non-branching tandem repeat, then $s[i+1..2|\alpha|+i] = \gamma a\gamma a$ is a tandem repeat. We say that $(i, a\gamma, 2)$ is on the left of $(i+1, \gamma a, 2)$, while $(i+1, \gamma a, 2)$ is on the right of $(i, a\gamma, 2)$. In

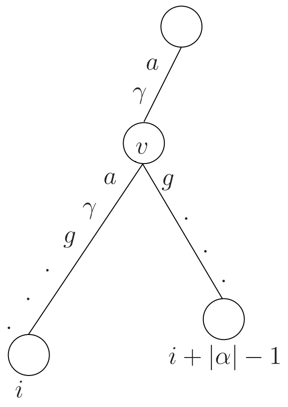


FIGURE 6.4: An example of a branching tandem repeat. $(i, \alpha, 2)$ is a branching tandem repeat, node v is at string depth $|a\gamma|$, and suffixes $suff_i$ and $suff_{|\alpha|+i-1}$ branch from node v.

Figure 6.5, both tandem repeats starting at positions i and $i + 1$ are non-branching, while the tandem repeat starting at position $i + 2$ is a branching tandem repeat. We refer to tandem repeat $(i, a\gamma, 2)$ as a left rotation of tandem repeat $(i + 1, \gamma a, 2)$; right rotation is similarly defined.

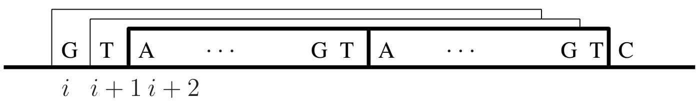


FIGURE 6.5: An example of a non-branching tandem repeat. Tandem repeats starting at positions i and $i + 1$ are both non-branching, while the tandem repeat starting at position $i + 2$ is branching. It is easy to see from this example each non-branching tandem repeat is to the left of another tandem repeat.

The non-branching tandem repeats that are next to each other can be considered a chain, with a branching tandem repeat at the end of the chain. Therefore, by locating all branching tandem repeats, and detecting the non-branching tandem repeats to their left, all tandem repeats can be identified. Hence, we focus on identifying all the branching tandem repeats.

A naive algorithm to identify branching tandem repeats is as follows:

1. For each internal node v, collect all the leaves in the subtree rooted by v in a list $\ell\ell(v)$.
2. Let α be the path label of v. Each leaf represents a suffix $suff_i$, and for each suffix $suff_i$ in $ll(v)$ check if $suff_{|\alpha|+i}$ is in $\ell\ell(v)$.
3. If so check if character $s[i]$ is the same as character $s[2|\alpha| + i]$. If so, $(i, \alpha, 2)$ is a branching tandem repeat.

If we can identify whether a suffix $suff_j$ is in $\ell\ell(v)$ in constant time, then the algorithm runs in $O(n^2)$ time. If we number all leaf nodes according to the order they are encountered in a post-order traversal, then leaves in the subtree under any internal node are consecutive. We can mark this range for each internal node v by storing the number of the first leaf, i.e., the leftmost leaf in the subtree; and the last leaf, i.e., the rightmost leaf. Suppose the leaf that represents $suff_i$ is the jth leaf we encounter in the post-order traversal, then in a separate array R we store j in $R[i]$. Therefore, we can identify whether a suffix $suff_i$ is in $\ell\ell(v)$ by checking if the value stored in $R[i]$ lies in the range of node v.

Since for each suffix $suff_i$ there can be $O(n)$ internal nodes on the path from the root to the leaf, the naive algorithm runs in $O(n^2)$ time. However, we can reduce this runtime to $O(n \log n)$ using the knowledge that the two suffixes in a branching tandem repeat are under different children of the node whose path label is the tandem repeat. So we can check the leaves under all but one child, and all branching tandem repeats can be identified. This is because if a branching tandem repeat has a leaf under the child we did not check, then the other leaf must be in a child we did check.

Let node v' be a child of node v that has the most leaves of all of node v's children. We define $\ell\ell'(v) = \ell\ell(v) - \ell\ell(v')$, and modify the naive algorithm by checking all leaves in $\ell\ell'(v)$ instead of $\ell\ell(v)$. Suppose a suffix $suff_i$ is in both $\ell\ell'(v)$ and $\ell\ell'(u)$, where u is a child of v. Then $|\ell\ell'(u)| \leq \frac{|\Sigma|-1}{|\Sigma|}\ell\ell'(v)$ where $|\Sigma|$ is the size of the alphabet; i.e., the number of leaves in $|\ell\ell'(u)|$ is at most $\frac{|\Sigma|-1}{|\Sigma|}$ times the number of leaves in $\ell\ell'(v)$. So a suffix can be in at most $\log_{|\Sigma|/(|\Sigma|-1)} n$ number of lists, resulting in an $O(n \log n)$ time algorithm.

After locating all branching tandem repeats we can find all the non-branching tandem repeats. Suppose that $(i, \alpha, 2)$ is a branching tandem repeat. If $s[i-1] = s[i+2|\alpha|-1]$ then $(i-1, \delta, 2)$ is a non-branching tandem repeat, where $\delta = s[i-1..i+2|\alpha|-2]$. So for each branching tandem repeat $(i, \alpha, 2)$, we check if its left rotation is a tandem repeat, if so we check the left rotation of this new tandem repeat until it is no longer true. This yields an $O(n \log n + occ)$ runtime algorithm, where occ is the total number of tandem repeats.

6.7.2 Stoye and Gusfield's $O(n)$ Algorithm

In 1998, Fraenkel and Simpson [15] proved that for a string $|s| = n$ there are at most $O(n)$ different types of tandem repeats. Tandem repeats $\beta = \alpha\alpha$ and $\gamma = \delta\delta$ are of different types if and only if $\alpha \neq \delta$. Since all occurrences of tandem repeats can be found from the knowledge of all the types of tandem repeats, it is of interest to find the latter. The set of all types of tandem repeats of a string s is also referred to as the vocabulary of s. Gusfield and Stoye designed a linear time algorithm to identify these [18].

String decomposition

The linear time tandem repeat identification algorithm uses Lempel-Ziv string decomposition, illustrated in Figure 6.6. At some stage during the execution of the algorithm, let i be the first position that is not in any block. Find a position $j < i$ that maximizes $|lcp(suff_i, suff_j)|$. Then mark the next block to be of length $\max\{1, |lcp(suff_i, suff_j)|\}$ starting from the ith position. This procedure is continued until the whole string is decomposed into blocks. An example of the Lempel-Ziv decomposition is shown in Figure 6.7.

This decomposition can be easily obtained using a suffix tree. Given a string s first build its suffix tree $ST(s)$. Then in a postorder traversal of the tree, mark each internal node u with the index of the smallest suffix in its subtree. As the postorder traversal visits all children of an internal node u before visiting u itself, node u is marked with the smallest of the numbers marking its children.

To create the decomposition, start by traversing along the path from root to leaf labeled

FIGURE 6.6: Lempel-Ziv Decomposition

```
Procedure Lemple_Ziv_Decomposition($S$)
        $blocks \leftarrow \emptyset$
        $block\_start \leftarrow 1$
        $block\_end \leftarrow 1$
        While $block\_end < |s|$ do
            Let $block\_len = \max\{1, \max_{k=1}^{block\_start-1} |lcp(suff_k, suff_{block\_start})|\}$
            $block\_end \leftarrow block\_start + block\_len - 1$
            $blocks \leftarrow blocks \cup (block\_start, block\_end)$
            $block\_start \leftarrow block\_end + 1$
        end while
end procedure
```

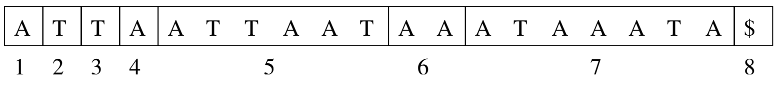


FIGURE 6.7: An example of the Lempel-Ziv decomposition of a string, each number under the block corresponds to the block number.

suff$_1$ in $ST(s)$. The traversal will continue only if the next node along the path is marked with a number smaller than the current position in the string. Continue the traversal until we cannot go any further, and this is the end of the block. Repeat this process by starting at the next position in the string and the root of $ST(s)$.

Using the string given in Figure 6.7 as an example, there is a node u with edge label a from the root of the suffix tree. This node is marked with 1, because *suff*$_1$ is in its subtree. When we start at position 1, we cannot go to node u because while its edge label is a, its marker is not less than 1. So the end of the block starting at position 1 is 1. The procedure is continued starting at position 2. It is easy to see that this algorithm produces the correct result, and its run time is $O(n)$.

Leftmost-covering set

Since we are only interested in discovering the vocabulary of tandem repeats, and not all their occurrences, it suffices to discover the leftmost occurrence of each type of tandem repeat. Recall that a non-branching tandem repeat is on the left of another tandem repeat with equal length, and this series of consecutive equal length tandem repeats forms a chain. Let (i, l) and (j, l) be two tandem repeats in such a chain. We say that (i, l) covers (j, l) if and only if $i < j$. A set of tandem repeats is a leftmost-covering set if and only if the leftmost occurrence of each type of tandem repeat is covered by a tandem repeat in the set.

Figure 6.8 shows an example of the leftmost-covering set. Tandem repeats $(1, 8)$, $(2, 2)$, $(2, 8)$, $(3, 8)$, $(4, 2)$, $(7, 6)$, $(11, 8)$ are the leftmost tandem repeats of their types. But tandem repeats $(2, 8)$ and $(3, 8)$ are covered by $(1, 8)$, so the leftmost-covering set is $\{(1, 8)$, $(2, 2)$, $(4, 2)$, $(7, 6)$, $(11, 8)\}$. Also note that this is the minimal leftmost-covering set, i.e., no other leftmost-covering set has fewer elements. However, in general a leftmost-covering set need not be minimal.

LEMMA 6.1 The leftmost occurrence of any tandem repeat type must span at least two

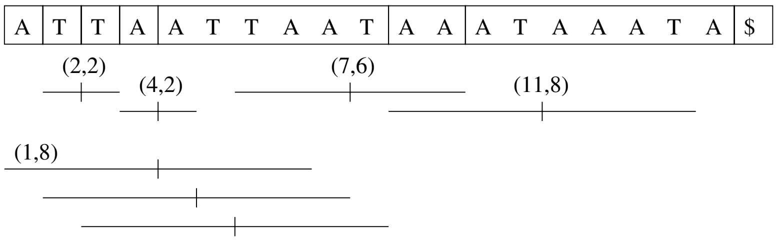


FIGURE 6.8: An example of leftmost-covering set. The leftmost occurrence of each tandem repeat is marked. The leftmost-covering set is $\{(1,8), (2,2), (4,2), (7,6), (11,8)\}$

blocks of the Lempel-Ziv decomposition.

Proof Let $\beta = \alpha\alpha = S[i..2|\alpha| + i - 1]$ be the leftmost occurrence of a type of tandem repeat. If β spans only one block in the Lempel-Ziv decomposition, then by definition of the Lemple-Ziv decomposition, there must exist $suff_j, j < i$, such that $lcp(suff_j, suff_i) \geq 2|\alpha|$. Then, $s[j..2|\alpha| + j - 1] = \beta$ must be an earlier occurrence of that type of tandem repeat.

LEMMA 6.2 The second half of any tandem repeat must not span more than two blocks of the Lempel-Ziv decomposition.

Proof If the second half of a tandem repeat spans more than two blocks of the Lempel-Ziv decomposition, then one block of the decomposition must lie completely inside the second half of the tandem repeat. But by the definition of Lempel-Ziv decomposition, this is impossible. If a block starts at position k of the second half of a tandem repeat, then the suffix starting at position k of the first half of the tandem repeat is sufficient to propel the block to the end of the tandem repeat.

COROLLARY 6.1 By Lemma 6.2, if a block of the Lemple-Ziv decomposition starts at a character that is part of the second half of a tandem repeat, then this block will last until at least the end of the second half of the tandem repeat.

From Lemmas 6.1 and 6.2, one of the following situations must occur for the leftmost occurrence of any tandem repeat type.

- There is a block starting at the same position as the start of the second half of a leftmost tandem repeat.
- There is a block starting after the start of the second half of a leftmost tandem repeat.
- There is no block starting on or after the first character of the second half of a leftmost tandem repeat.

Each of the three cases described above can be split into two sub-cases, based on whether there is another block contained in the left half of the tandem repeat or not. Figure 6.9 illustrates these six cases. Stoye and Gusfield presented two algorithms that will detect all tandem repeats with structures illustrated in Figure 6.9. The two algorithms are run for each block B of the Lempel-Ziv decomposition. Let h be the starting position of the current

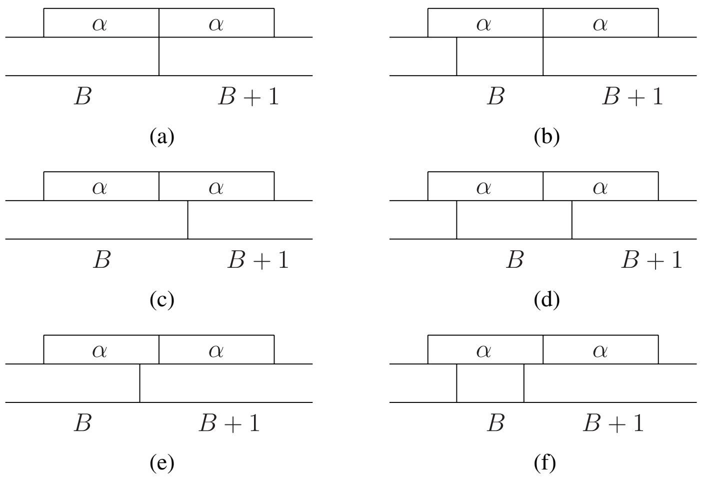


FIGURE 6.9: An enumeration of the possible cases

block, and h_1 be the starting position of the next block. Let $lcp\_r(suff_i, suff_j)$ denote the lcp in the reverse direction starting at positions i and j, i.e., the longest common suffix of prefixes ending at i and j. This can be easily calculated by reversing the string and building the suffix tree for it, along with the usual lcp algorithm.

To see how the first algorithm (Figure 6.10) works, suppose that block B starts at the ith character in the first half of the tandem repeat. Then k will eventually reach the ith character in the second half of the tandem repeat. At this point both k_1 and k_2 will be non-zero, and the length of the tandem repeat is $2k$. This corresponds to cases (b), (d), (e), and (f). On the other hand, the algorithm in Figure 6.11 starts from the ith character in the second half of the tandem repeat, and tries to detect the ith character in the first half; this detects cases (a), and (c).

The above two algorithms take $O(n)$ time because each block is processed once by Backward Detection (see Figure 6.11), and twice by Forward Detection (see Figure 6.10). Each position of the block takes constant time to process by each algorithm. Therefore, the total time is $O(n)$ so far. Also note that since the algorithm runs in $O(n)$ time, the

FIGURE 6.10: Forward Detection

```
Procedure Forward_Detection()
    for $k \leftarrow 1, |B| + |B+1|$
        $q \leftarrow h + k$
        $k_1 \leftarrow lcp(S_q, S_h)$
        $k_2 \leftarrow lcp\_r(S_{q-1}, S_{h-1})$
        if $k_1 + k_2 \geq k$ and $k_1, k_2 > 0$
            if $\max(h - k_2, h - k + 1) + k < h_1$
                Output $(\max(h - k_2, h - k + 1), 2k)$
            end if
        end if
    end for
end procedure
```

FIGURE 6.11: Backward Detection

```
**Backward_Detection**()
        for $k \leftarrow 1, |B|$
            $q \leftarrow h_1 - k$
            $k_1 \leftarrow lcp(S_q, S_{h_1})$
            $k_2 \leftarrow lcp\_r(S_{q-1}, S_{h_1-1})$
            If $k_1 + k_2 \geq k$ and $k_1 > 0$
                if $\max(q - k_2, q - k + 1) + k < h_1$
                    Output $(\max(q - k_2, q - k + 1), 2k)$
                end if
            end if
        end for
end procedure
```

number of tandem repeats reported is also $O(n)$. However the result may not be a minimal leftmost-covering set, i.e., some of the tandem repeats reported are either not the leftmost occurrence of its type, or are covered by other tandem repeats in the set, or both.

We have successfully computed a leftmost-covering set, and would now like to mark the tandem repeats in this set in the suffix tree. We begin by first sorting all the tandem repeats by their beginning position, and then by their length (from longest to shortest). This way all the tandem repeats starting from position i are next to each other and ranked according to their length. All such tandem repeats that start form position i are associated with the leaf node v, representing $suff_i$. We call this list of tandem repeats $p(v)$.

Let u be the parent of v, and let k be the string depth of node u. For each tandem repeat $(i, l) \in p(v)$, if $l \geq k$ then mark the position on the edge from u to v or on node u itself, and continue until $l < k$. Since $p(v)$ is sorted the amount of work is proportional to the number of tandem repeats processed. After all the children of node u are processed, then we need to calculate the list $p(u)$. It is not possible to merge all the lists of the children, because this will take $O(n)$ time for each node, and $O(n^2)$ total time.

Each node is labeled with the number of the suffix that has the smallest index in its subtree, i.e., we label node v with i if and only if $j > i$ for each suffix $suff_j$ in v's subtree. To compute $p(u)$, we simply adopt $p(v)$ where v is the child with the smallest label.

LEMMA 6.3 By adopting the list of the child with the smallest label, all the tandem repeats in the leftmost-covering set will be marked.

Proof By induction, assume that all the tandem repeats in the leftmost-covering set under a node u are marked correctly. This is true for internal nodes whose children are all leaf nodes, which serves as the base case. Now we show that the edge e between u and its parent v is marked correctly. Suppose that (i, l) is a part of the leftmost-covering set, and that a position on e should be marked as a result. Then (i, l) must be the first occurrence of that type of tandem repeat. Thus $suff_i$ is the first suffix with that type of tandem repeat as a prefix. Therefore (i, l) is an entry in $p(w)$, where w is the child with the smallest label.

Once the leftmost-covering set is marked in the suffix tree, any tandem repeat is covered by one of the tandem repeats in this set. Let $\beta = \alpha\alpha = a\gamma$, where a is a character. If there is a tandem repeat to its right with the same length, then this tandem repeat must be of the form γa. To mark this tandem repeat, if an internal node v has the path label $a\gamma$, one can travel from $a\gamma$ to γ in the suffix tree by using the suffix link from v. Otherwise, let u be the parent of v, and $a\gamma$ lies inside the edge label of the edge between u and v. Then, first go up to node u and travel to node u' using the suffix link from u. Then, travel down in the suffix tree. Note that for each edge encountered, every character of the edge label need not be compared. One can simply compare the first character, and move down by the length of the edge label. This marks all types of tandem repeats.

Although this compare-and-skip method allows us to traverse each edge in constant time, the number of the edges in the traversal could be large, and result in a non-linear time algorithm. In order to calculate how many times an edge is traversed in the algorithm, we first state the theorem presented in Fraenkel and Simpson [15].

THEOREM 6.1 *Each position i in string s can be the starting position of at most two rightmost occurrences of tandem repeats.*

From the above theorem we can deduce the following.

LEMMA 6.4 For each edge e between node u and node v, there can be at most two marked positions each being the endpoint of some tandem repeat.

Proof Suppose that an edge e between node u and its child node v has more than two marked positions. Let suffix $suff_i$ be the rightmost suffix in string s under node v, i.e., for all $suff_k$ in the subtree rooted at node v, $k < i$. Then position i is the starting position of the rightmost occurrence of more than two types of tandem repeats, a contradiction.

LEMMA 6.5 Each edge is traversed no more than $O(|\Sigma|)$ times in marking all the tandem repeat types.

Proof Let node u be the parent of node v, let u' be the internal node reachable from u using the suffix link labeled c, let v' be the internal node reachable from v using the suffix link labeled c. Since there is an edge between u and v, then there is a path between u' and v'; we call this a suffix link induced path. Let edge e be an edge on this suffix link induced path. By Lemma 6.4 there are only two marked positions between nodes u and v. As a result e will be traversed at most twice in order to mark the tandem repeats that are right rotations of the two tandem repeats ending between nodes u and v. Furthermore, any edge e can only be on $|\Sigma|$ number of suffix link induced paths. Thus each edge e is traversed $O(|\Sigma|)$ times.

By Lemma 6.5 each edge is traversed at most $O(|\Sigma|)$ times. Since there are $O(n)$ edges, the total runtime of the algorithm is $O(|\Sigma|n)$. For constant size alphabet, the runtime is $O(n)$.

6.8 Identification of Promoters and Regulatory Sequences

Gene expression, the process by which a gene is transcribed into corresponding mRNA sequences, is aided by promoters and other regulatory sequences usually located upstream of the transcribed portion of the gene. The upstream region typically consists of several important short subsequences, usually 4-10 nucleotides long, that play a role as binding sites for transcription factors. It is known that these sequences are often conserved between similar genes, and also genes that are similarly expressed. The problem of identifying multiple unknown patterns with flexible distance constraints between them is in general known as *structured motif identification problem.* By extracting potential motifs of regulatory sites in gene upstream regions, biologists can gain valuable insight into gene expression regulation. It is natural to use a suffix tree to identify motifs in DNA sequences, because of its suitability to find common substrings in multiple sequences. Marsan and Sagot [25] proposed algorithms to solve the sequence motif identification problem. We present a simplified version by focusing on identification of two patterns. For a more detailed treatment of motif identification problems the reader is referred to [25] and to Chapter 37 of this handbook.

Given a set of m DNA sequences each corresponding to the upstream region of a gene, if a nucleotide sequence of length k is found upstream in all the sequences, then this sequence is a possible motif. This is a simplified view of sequence motifs, because of the following: 1) Not all the m genes may have similar function, so that they might have different motifs. 2) Not all the upstream regions will have an identically common sequence due to evolution, and random mutations. 3) All the motifs should be a similar distance away from the gene. For example, if a sequence occurs 20 base pairs upstream from a gene, while the exact sequence occurs 1000 base pairs upstream from another gene, then it is more likely to be a coincidence than an actual motif. 4) It is possible that the set of m DNA sequence have the same subsequence upstream by chance, therefore we should restrict the motif to be more complicated than one short exact match.

We consider the two pattern motif problem: $((\beta_1, \beta_2), (d_{min}, d_{max}))$ is a motif if there is a subset of q sequences out of all the m input sequences that have substring matches β_1 and β_2, and the two substrings are at least d_{min} away from each other, and at most d_{max} away from each other. This definition can be relaxed, such that we do not need exact matches to β_1 and β_2, but allow a few mismatches. We can restrict the definition by setting a length k for β_1 and β_2.

Build a generalized suffix tree for all the m input sequences. Augment each internal node v of the suffix tree with a boolean array $sequences_v$ of size m, such that $sequences_v[i]$ is set to 1 if and only if a suffix from sequence i is a leaf in the subtree rooted at node v. We also augment each internal node v with a counter $count_v$, such that $count_v$ is the number of 1's in $sequences_v$. Then all the motifs can be identified by a tree traversal. Let p be a position inside the edge label of the edge (u, v) where node u is the parent of node v. If the string depth from the root of the suffix tree to p is between $2k + d_{min}$ and $2k + d_{max}$ and $count_v \geq q$, then the concatenation of all edge labels from the root to p is a potential motif. All potential motifs can be generated in $O(mn)$ time, where m is the number of sequences and n is the total length of all the sequences.

Suppose we would like to consider substrings with e number of mismatches as well. Then we can generate all strings of length k and test if a particular string s_i can be β_1 of the motif. Then we consider all paths beginning at the root of the suffix tree that are e mismatches away from s_i. To find the number of sequences similar to s_i, combine all $sequences_v$ arrays with a logical OR and count the number of 1's in the array. Figure 6.12 shows a generalized suffix tree of AGTACG$\$_1$ and ACGTCA$\$_2$. Suppose the pattern is AGT, and one mismatch is allowed, then the path AGT and CGT will be found. After β_1 is found, find downward paths below the position corresponding to the end of β_1, with lengths between d_{min} and d_{max}, and search for β_2. In the case of the example in Figure 6.12, assume $d_{min} = 0$, $d_{max} = 1$ and allow m_2 to be length 2. Then we can identify the motif ((AGT,CA),(0,1)) in strings AGTACG$\$_1$ and ACGTCA$\$_2$.

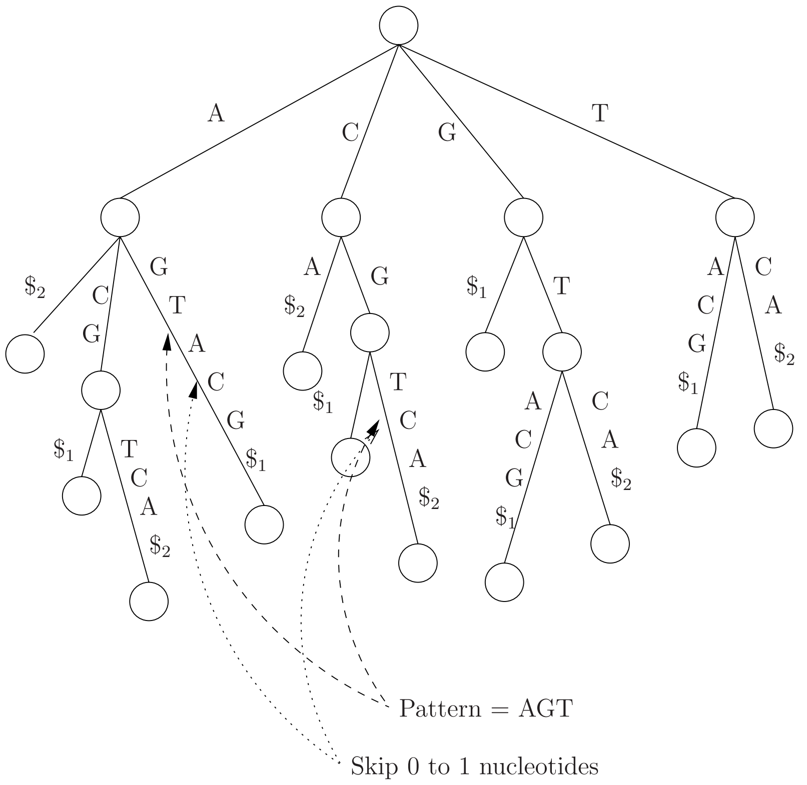


FIGURE 6.12: The generalized suffix tree of AGTACG$\$_1$ and ACGTCA$\$_2$. The search for the pattern AGT, allowing one mismatch yields the paths AGT and CGT. Then we skip 0 or 1 nucleotides and try to identify the other part of the motif. We then find ((AGT,CA),(0,1)) as a motif common to the strings.

6.9 Oligonucleotide Selection

Microarrays are useful in measuring the concentration levels of a target set of DNA sequences. They are based on the concept that two DNA sequences exhibiting complementarity hybridize to each other. If the sequences of the target DNA molecules are known, we can choose a unique oligonucleotide (a short DNA sequence) called a *probe* for each target DNA molecule and attach the probes to the microarray. It is important that the probe be unique in the sense that it hybridizes to only its intended target DNA. To measure the concentration of the target DNA molecules in a solution, they are separated into single stranded molecules, colored with a fluorescent dye, and allowed to hybridize with the fixed probes on the microarray. By using a laser to detect the fluorescence at each microarray spot, the intensity can be used to estimate the concentration of the target DNA molecule. DNA microarrays are commonly used to simultaneously measure the expression levels of tens of thousands of genes of an organism. They have also been used to detect the concentration levels of microorganisms by designing unique probes based on their genomic sequences.

The design of oligonucleotides is challenging because the probes must each be unique to a target sequence. Furthermore, a DNA sequence can hybridize to a probe that it does not match exactly. To account for this, we must select a set of probes such that each probe is unique up to k differences. Because of the hybridization process, if the two probes differ in the first or last k nucleotides and the remaining nucleotides are same, unintended hybridizations are still likely because hybridization can happen to the common part. Therefore, it is best to have the differing positions distributed evenly throughout the probe. Kurtz *et al.* [21] developed an algorithm to design probes as one of the many applications of their repeat finding software REPuter [22]. Subsequently, Kaderali and Schliep [20] have proposed a more complex model for probe design by further screening the unique sequences using their hybridization temperature. In this section we present the approach by Kurtz *et al.* to illustrate how suffix trees can be used in probe design. For a thorough treatment of probe design, the reader is referred to Chapter 24 of the handbook.

For ease of understanding, we restrict ourselves to the problem of designing probes for two target sequences S_1 and S_2. If the probes are too short, the sequences cannot be distinguished from each other; longer probes are harder to manufacture. To model this, let ℓ_{min} and ℓ_{max} be the minimum and maximum allowable length of the probes, respectively. As mentioned above, the probes should also include at least k mismatches, distributed as evenly as possible throughout the probes.

6.9.1 Maximal k-mismatch repeat

In order to design the probes we first look at the maximal k-mismatch repeat problem. Two substrings $s_1[i_1..j_1]$ and $s_2[i_2..j_2]$ are said to be a k-mismatch repeat if we can obtain one from the other by exactly k character replacements, i.e., they mismatch at exactly k positions. A k-mismatch repeat is said to be maximal if we cannot extend it at either end without incurring an extra mismatch.

To identify maximal k-mismatch repeats, first a generalized suffix tree is built for the two target sequences s_1 and s_2. Traverse the tree and mark each internal node u as mixed, if and only if u is the $lca(w_1, w_2)$ where w_1 and w_2 are leaves from s_1 and s_2 respectively. All the mixed internal nodes can be found in $O(n)$ time, where $n = |s_1| + |s_2|$. For each node u we maintain two Boolean values m_1 and m_2; m_1 is set to true if and only if there is a leaf corresponding to a suffix of s_1 in the subtree rooted at u, or u is a leaf node and corresponding to a suffix of s_1; m_2 is similarly defined. This can be done in $O(n)$ time with one post-order traversal of the tree. Then an internal node u is mixed if and only if m_1 at v_1 is true and m_2 at v_2 is true, where v_1 and v_2 are two of u's children. The mixed nodes can be identified with another post-order traversal in $O(n)$ time. In fact the two post-order traversals can be combined into one without changing the asymptotic run-time.

Suppose we are interested in maximal k-mismatch repeats of length at least l. This implies that the maximal k-mismatch repeat has a maximal exact match of length at least $\frac{l}{k+1}$. For each internal node v of string depth at least $\frac{l}{k+1}$ and marked mixed, identify a pair of leaves w_1 and w_2 such that $lca(w_1, w_2) = v$, where w_1 corresponds to a suffix of s_1, and w_2 corresponds to a suffix of s_2. This can be done by a bottom up traversal of the tree. For each node maintain a list of all the leaves of s_1 — call this $list_1$, and another list of all the leaves of s_2 — call this $list_2$. For a leaf node one of the lists is empty and the other has exactly one element. For an internal node v, all distinct pairs of leaves can be generated by choosing an element from $list_1$ of one of the children and an element from $list_2$ of another child. After all pairs of leaves are generated, $list_1$ for v can be constructed by joining all the $list_1$'s children, and $list_2$ for v can be constructed in the same manner. This step can be done in $O(n)$ space and $O(n + occ)$ time, where occ is the number of pairs generated. The space for the suffix tree is $O(n)$ and the total size of the lists is also $O(n)$. While the bottom up traversal takes only $O(n)$ time, the number of pairs generated can be $\Theta(n^2)$ in the worst case.

For each pair of leaves w_1 and w_2 generated, find the length of $lcp(w_1, w_2)$. Let $s_1[i_1..j_1]$ and $s_2[i_2..j_2]$ be the two substrings in s_1 and s_2, respectively, corresponding to $lcp(w_1, w_2)$. It is clear that $s_1[j_1 + 1] \neq s_2[j_2 + 1]$, because the lcp would be longer otherwise. Let $suff^1_{j_1+2}$ be the $(j_1 + 2)$th suffix of S_1, and $suff^2_{j_2+2}$ be the $(j_2 + 2)$th suffix of S_2. If $lcp(suff^1_{j_1+2}, suff^2_{j_2+2})$ is of length r, then substrings $s_1[i_1..j_1 + 1 + r]$ and $s_2[i_2..j_2 + 1 + r]$ are a maximal 1-mismatch repeat. We can repeat this procedure to find the maximal k-mismatch repeat by extending to the right and/or to the left. Given a pair of leaves as seed, we can identify a maximal k-mismatch repeat in $O(k)$ time, because finding each required lcp takes only constant time with preprocessing for lca. We can then check if the maximal k-mismatch repeat is within the specified length constraints.

6.9.2 Oligonucleotide design

From the algorithm presented above, we can generate all the maximal k-mismatch repeats and check if their length l is within l_{min} and l_{max} in $O(n + occ \cdot k)$ time, where occ is the number of pairs generated. Note that a maximal k-mismatch cannot be extended on either side without incurring an extra mismatch. However, the maximal k-mismatch can be shortened on either side if necessary by deleting nucleotides at either end without going as far as the the first mismatch position. This flexibility can be used to increase the chance of finding a k-mismatch probe within the specified length constraints. The probe selection algorithm for two sequences that is presented here can be extended to more than two sequences, with the same run-time of $O(n + occ \cdot k)$, where n is the total length of all the sequences. In the worst case, the number of pairs generated is $\sum_{i=1}^{k} \sum_{i<j} n_i \cdot n_j$, where n_i and n_j are the length of sequence i and j, respectively.

6.10 Protein Database Classification and Peptide Inference

6.10.1 Protein sequence database classification

As previously mentioned, the volume of biological sequence data has increased exponentially in recent years. To better facilitate the analysis of this data, efficient indexing is needed. Also due to high throughput sequencing, it is no longer efficient or even feasible to have researchers manually process the large number of data generated each day, and update sequence databases. With these two goals in mind, an automated process was designed and implemented by Gracy and Argos [16, 17] to classify an entire protein sequence database. In this section we present their process, and the role played by suffix trees.

Since our knowledge of the protein structure and their function is limited, data mining methods are used to discover similarities between individual proteins in a protein family or cluster. However, in order to apply these data mining methods, the protein sequences in the database must first be classified into homologous families. In order to achieve this goal, very similar protein sequences are first classified together by a composition similarity search, where a compositional vector is calculated for each protein sequence based on the number of amino acids and dipeptides. Then a pairwise composition distance is calculated by finding the L_1 distance of the compositional vectors associated with each pair of protein sequences. Only pairs with distances smaller than a threshold are selected into a family.

The composition distance gives a good starting point for further, more sensitive comparisons. From each family identified in the previous step, one protein sequence is selected as a representative. The goal of this step is to further reduce the number of families by grouping together representative sequences. To accomplish this goal efficiently, regions of local similarity need to be identified and used as an anchor. The local similarities in this case are equal length subsequences that share comparable prefixes. To identify these equal length comparable prefixes a generalized suffix tree is built for all the selected sequences. Then the nodes of the suffix tree are visited by a depth-first traversal. For each node v encountered, all the nodes u of equal depth are found and a similarity score is calculated for the path labels of each pair of nodes uv. The identified anchors are then extended to both ends. If the resulting match exceeds a cut-off point then the two protein families are potentially similar, and further checks are performed.

6.10.2 Experimental Interpretation

Background

Tandem mass spectrometry can be used to identify protein sequences. When a large number of experiments are conducted, it is impractical to interpret each experimental output manually. This tedious and repetitive process can be done by sequence database searches. Due to the large data size, an efficient database is needed to effectively identify potential candidates.

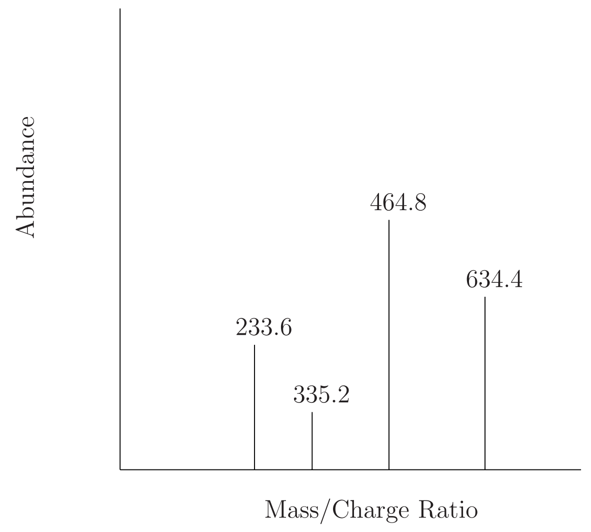
In protein studies, the first step is usually the identification of the protein sequence. A protein sequence is first digested with an enzyme to produce short peptides. A mass spectrometry (MS) is then used to measure the mass-to-charge (m/z) ratios of the resulting peptides and these ratios are used in further selecting peptides of interest. The selected peptides are fragmented by a pass through a collision cell, in a step referred to as collision-induced dissociation (CID). At this point the peptide is broken into shorter peptides and individual amino acids. Another mass spectrometry (MS) measures the m/z ratio of the resulting amino acids. This procedure allows us to deduce the mass of the amino acids in the peptide if the charges are known.

However, this does not directly tell us the sequence of the peptide. In order to find the sequence of the peptide, a database of possible peptides must be searched to produce the best answer. Further complicating this process, we may not know which enzyme is used to produce the initial peptides, so the leading/trailing amino acid is not known. Post-translational modification could change the mass of a peptide, which will also have to be taken into consideration during the construction or the search of the index. So clearly the goal is to build an indexing structure and a search routine so that the best interpretation of a tandem mass spectrometry experiment can be found quickly in the database. In this section we will study the indexing and searching method proposed by Lu and Chen [23].

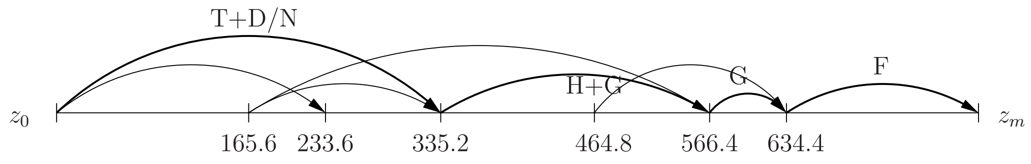
NC-spectrum graph

Given a mass spectrograph as an input, a NC-spectrum graph is constructed using the algorithm by Chen *et al.* [8]. Suppose there are k peaks in the mass spectrograph, then the peptide is broken into k fragments $I_1, \ldots, I_k$ with masses denoted by $w_1, \ldots, w_k$, respectively. Figure 6.13(a) shows an example mass spectrograph with four peaks.

The NC-spectrum graph with $2k + 2$ vertices is created on the real number line. Let $m = 2k + 1$, vertex z_0 and z_m correspond to zero mass, and the total mass of the peptide W, respectively. For each peak I_j, two vertices z_j and z_{m-j} are added to the graph, one at position w_j, and the other at position $W - w_j$, respectively. This is the same as assuming a fragment is either a prefix or a suffix of the peptide. Obviously, a fragment could be a substring in the middle of the original peptide. However, since we do not always know which enzyme was used to digest the protein, we lack the start and end points. Thus assuming a fragment is a substring in the middle of a peptide does not help us deduce the actual



(a) Mass spectrograph of a hypothetical peptide.



(b) The NC-spectrum graph of (a)

FIGURE 6.13: The total mass of the peptide is 800 amu, and the bolded edges forms a possible peptide. The symbol '+' means 'concatenation', and the symbol '/' means 'disjunction'. For example, the express 'T+D/N' means 'TD' or 'TN'.

sequence, and it will complicate the search effort because we do not know where to place it on the line. After the vertices are fixed on the line, edges are added to the graph. Suppose $z_i < z_j$ are two vertices in the graph. If the difference between z_i and z_j corresponds to the mass of some amino acid, an edge is drawn from z_i to z_j and is labeled with that amino acid.

Figure 6.13(b) shows an example of the NC-spectrum graph corresponding to the mass spectrograph of Figure 6.13(a). This NC-spectrum graph has only eight vertices instead of the ten vertices. This is because the peaks corresponding to molecular weights 335.2 and 464.8 map to the same two vertices on the line. It is easy to see that the concatenation of all the edge labels of a path from vertex z_0 to z_m in the NC-spectrum graph corresponds to a peptide. This peptide is one of the many peptides that could have produced the mass spectrograph.

The Peptide Inference Algorithm

Given a database of proteins, the goal is to identify a set of good candidate peptides from the database for a particular mass spectrograph. First a generalized suffix tree is constructed, consisting of all the proteins in the database. Then a depth first traversal is done on the

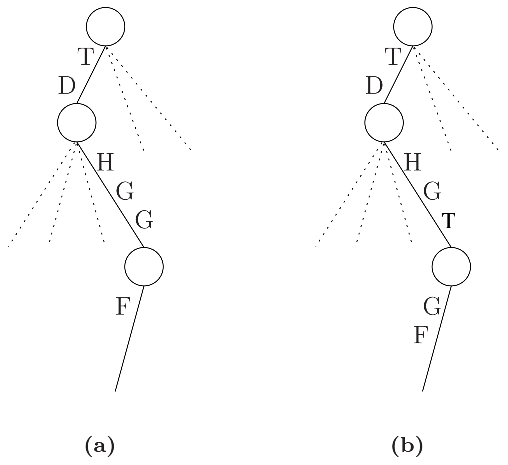


FIGURE 6.14: Suppose the NC-spectrum graph in Figure 6.13 (b) is used in our search. (a) A path that matches the peptide TDHGGF completely, when F is encountered in the search. The peptide TDHGGF is returned as a candidate peptide. (b) A path with an insertion T in the middle (between the two G's), the peptide is also returned as a candidate.

suffix tree by referring to the NC-spectrum graph. Start at the root of the suffix tree r, and vertex z_0 of the NC-spectrum graph. Let (r, u) be an edge in the suffix tree with edge label l_{ru}. We can map this edge onto the NC-spectrum graph by starting from z_0 and following the appropriate edge on the graph. Continue this until a position in the suffix tree is reached such that the corresponding path in the NC-spectrum graph reaches the last vertex z_m, or the path in the NC-spectrum graph can no longer be extended. In the first case the concatenation of the edge labels of the suffix tree from the root r to the current position is a candidate peptide. In the second case the current position in the suffix tree cannot yield a possible match. In this case, backtrack in the suffix tree and the NC-spectrum graph by taking a different edge in the suffix tree and continue; see Figure 6.14(a).

In order to account for experimental errors, the search can be relaxed by allowing errors. If the search is at node u in the suffix tree and vertex z_j in the NC-spectrum graph, but there is no outgoing edge from u in the suffix tree that has the same label as the edge from z_j we are interested in, we could skip one character from u and check if the edge label is available. For example, in Figure 6.14(b), we are searching for the peptide TDHGGF in the suffix tree. The prefix TDHG is located, however the next amino acid is T instead of G. This T is skipped and the search routine tries to locate the remaining sequence GF, which comes after T. So the peptide TDHGTGF is returned as a candidate. If a match cannot be found by skipping one character in the suffix tree, more characters can be skipped depending on the quality of the spectrograph.

This algorithm allows searching for all possible candidates in a protein database using $O(n + |G|)$ space, where n is the size of the protein database, and $|G|$ is the size of the NC-spectrum graph. Since we would like to return all the possible candidates in the protein database, a complete traversal of the suffix tree may be necessary. This takes $O(n)$ time. As the entire suffix tree may need to be traversed in the worst case, it is advantageous to use the linked list implementation of children of internal nodes to save space without increasing the worst case run-time.

After all the candidate peptides are located, a probability can be associated with each pep-

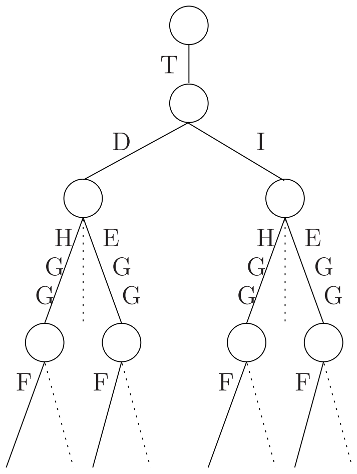


FIGURE 6.15: Post-translational modifications. Consider D can be modified to be I, H can be modified to E, and all other amino acids have no allowed modifications.

tide, and the peptide with the highest matching probability is output. Careful readers may notice we have not yet addressed the problem of post-translational modification. This can be done by modifying the search algorithm. Keep a table of all possible post-translational modifications such that for each amino acid a_i, a list of all its possible modifications are stored. During the depth first traversal of the suffix tree, instead of simply referring to the NC-spectrum graph to decide whether a path in the suffix tree is a candidate, each amino acid in the edge label of the NC-spectrum graph is also substituted with all its possible modifications to check if a path in the suffix tree can be a possible candidate. For example, in Figure 6.15, the only two modifications are from D to I and H to E. Suppose the path label in the NC-spectrum graph is TDHGGF, then the path TDHGGF will generate a match, and because of the modifications the paths TDEGGF, TIHGGF, and TIEGGF will also be considered as matches.

6.11 Conclusions

Suffix trees and its variants are used in many applications in computational biology. This chapter provides a diverse, but by no means exhaustive, sample of the many applications in which suffix trees have been used. Variants of suffix trees have also been developed for use in Markov models. These include prediction suffix trees [6] and probabilistic suffix trees [2, 5, 13].

Acknowledgements

This work was supported in part by the U.S. National Science Foundation under IIS-0430853.

References

[1] S.F. Altschul, T.L. Madden, A.A. Schäffer1, and J. Zhang *et al.* Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[2] A. Apostolico and G. Bejerano. Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. *Journal of Computational Biology*, 7(3):381–393, 2000.

[3] A. Apostolico and F.P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.

[4] S. Batzoglou, L. Pachter, J.P. Mesirov, and B. Berger *et al.* Human and mouse gene structure: Comparative analysis and application to exon prediction. *Genome Research*, 10(7):950–958, 2000.

[5] G. Bejerano, Y. Seldin, H. Margalit, and N. Tishby. Markovian domain fingerprinting: Statistical segmentation of protein sequences. *Bioinformatics*, 17(10):927–934, 2001.

[6] G. Bejerano and G. Yona. Variations on probabilistic suffix trees: Statistical modeling and prediction of protein families. *Bioinformatics*, 17(1):23–43, 2001.

[7] N. Bray, I. Dubchak, and L. Pachter. Avid: A global alignment program. *Genome Research*, 13(1):97–102, 2003.

[8] T. Chen, M.Y. Kao, M. Tepel, and J. Rush *et al.* A dynamic programming approach to de novo peptide sequencing via tandem mass spectrometry. In *Proc. 11th annual ACM-SIAM symposium on Discrete algorithms*, pages 389–398, 2000.

[9] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31, 2002.

[10] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[11] A.L. Delcher, S. Kasif, R.D. Fleischmann, and J. Peterson *et al.* Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[12] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[13] E. Eskin, W.S. Noble, and Y. Singer. Protein family classification using sparse markov transducers. *Journal of Computational Biology*, 10(2):187–213, 2003.

[14] A. Fire, S. Xu, M.K. Montgomery, and S.A. Kostas *et al.* Potent and specific genetic interference by double-stranded rna in caenorhabditis elegans. *Nature*, 391(6669):806–811, 1998.

[15] A.S. Fraenkel and J. Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998.

[16] J. Gracy and P. Argos. Automated protein sequence database classification. i. integration of compositional similarity search, local similarity search, and multiple sequence alignment. *Bioinformatics*, 14(2):164–173, 1998.

[17] J. Gracy and P. Argos. Automated protein sequence database classification. ii. delineation of domain boundaries from sequence similarities. *Bioinformatics*, 14(2):174–187, 1998.

[18] D. Gusfield and J. Stoye. Linear-time algorithms for finding and representing all tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–

546.

[19] Y. Horesh, A. Amir, S. Michaeli, and R. Unger. A rapid method for detection of putative rnai target genes in genomic data. *Bioinformatics*, 19(Suppl. 2):73ii–80ii, September 2003.

[20] L. Kaderali and A. Schliep. Selecting signature oligonucleotides to identify organisms using DNA arrays. *Bioinformatics*, 18(10):1340–1349, 2002.

[21] S. Kurtz, J.V. Choudhuri, E. Ohlebusch, and C. Schleiermacher *et al.* REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–3642, 2001.

[22] S. Kurtz and C. Schleimermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[23] B. Lu and T. Chen. A suffix tree approach to the interpretation of tandem mass spectra: applications to peptides of non-specific digestion and post-translational modifications. *Bioinformatics*, 19(Suppl. 2):ii 113–ii 121, 2003.

[24] M.G. Main and R.J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.

[25] L. Marsan and M.F. Sagot. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. *Journal of Computational Biology*, 7(3):345–362, 2000.

[26] B. Morgenstern. DIALIGN2: Improvement of the segement-to-segment approach to mulitple sequence alignment. *Bioinformatics*, 15(3):211–218, 1999.

[27] B. Morgenstern, O. Rinner, S. Abdeddaïm, and D. Haase *et al.* Exon discovery by genomic sequence alignment. *Bioinformatics*, 18(6):777–787, 2002.

[28] S. Schwartz, Z. Zhang, K.A. Frazer, and A. Smit *et al.* Pipmaker – a web server for aligning two genomic dna sequences. *Genome Research*, 10(4):577–586, 2000.

[29] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In *Proc. 9th Annual Symposium, Combinatorial Pattern Matching*, pages 140–152, 1998.

[30] E. Ukkonen. Approximate string-matching over suffix trees. In *Proc 4th Annual Symposium, Combinatorial Pattern Matching*, pages 228–242, 1993.