CS5238 Combinatorial methods in bioinformatics 2005/2006 Semester 1

Lecture 4: Suffix Tree and Suffix Array, September 2,2005

Lecturer: Wing-Kin Sung *Scribe: Zhang Zhiyong Melvin*

4.1 Introduction

Suffix tree is a fundamental data structure for combinatorial pattern matching[1]. In this note, we will introduce suffix tree, its construction and applications in computational biology problems. A related data structure, suffix array will also be presented. Finally in the last section of the notes, we will present a solution for the approximate matching problem using suffix array.

4.2 Suffix Tree

We first give a brief introduction on the trie data structure which is the predecessor of the suffix tree. This is followed by a detailed explanation of the suffix tree and finally we will present a simple algorithm to construct a suffix tree.

4.2.1 Suffix Trie

A trie (coming from the word re*trie*val) is a tree data structure for storing strings of a finite alphabet that allows for fast text searching. In a trie, each edge is labeled by a character and every path from the root to a leaf represents an input string. Suffix trie is simply a trie storing all possible suffixes of a string S. Table 4.1 shows all possible suffixes of a string $S = acacag\$$, where \$ indicates the end of S and the index starts from 1. The corresponding suffix trie can be found in Figure 4.1. In a suffix trie, each possible suffix is a path from the root to a leaf.

4.2.2 Suffix Tree

4.2.2.1 Definition

A suffix tree is a compacted trie of all the suffixes of a string S, where nodes with only one child are merged. The edge label of each edge in the suffix tree is the concatenation of the labels of the merged edges of the suffix trie. The path label of a node is defined as the concatenation of the edge labels from the root to the node. Based on these two definitions, the suffix tree made from the suffix trie in Figure 4.1 is shown in Figure 4.2. Note that edges that link to the leafs of the

Starting index	Suffix
1	$acacag\$$
2	$cacag\$$
3	$acag\$$
4	$cag\$$
5	$ag\$$
6	$g\$$
7	$\$$

Table 4.1: The suffixes of $S = acacag\$$

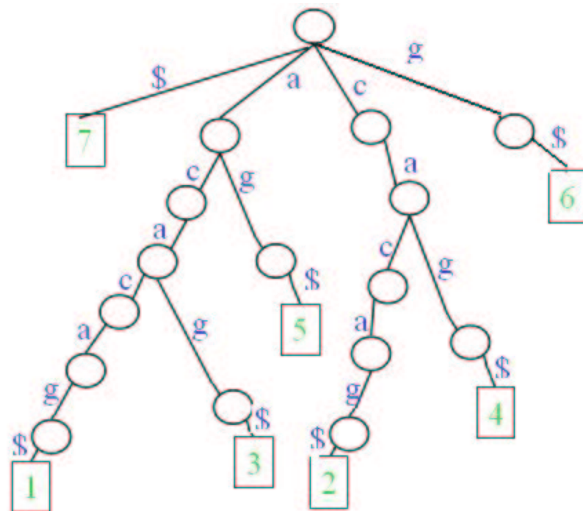


Figure 4.1: The suffix trie of $S = acacag\$$

tree are also called leaf edges and we define the string depth of a node to be the length of the path label of the node.

4.2.2.2 Space Complexity

Suffix tree has exactly n leaves and at most $2n$ edges, where n is the length of the string including \$. The label of each edge can be represented using a pair of indices, (a, b), where a is the starting position and b is the ending position of the label in the string. For example, in Figure 4.2, the edge "*a*" is represented by (1,1), the edge "*ca*" is represented by (2,3) and the edge "*cag*" is represented by (4,7). Notice that the end of every leaf edge should be 7, the last index of S. Thus for leaf edges we only need to store the start index. For example, the edge "*cag*" is represented by (4). Each index is an integer between 1 and n and hence it can be represented by $\log n$ bit. Thus a suffix tree which has at most $2n$ edges can be stored in $O(n \log n)$ bits.

Figure 4.2: The suffix tree of $S = acacag\$$

4.2.2.3 Generalized suffix tree

A generalized suffix tree is a suffix tree for two or more strings. For example, the generalized suffix tree of $S_1 = acgat\#$ and $S_2 = cgt\$$ is shown in Figure 4.3. The leaves with underlined labels are the leaves of S_2, and the rest of the leaves are the leaves of S_1. Note that we need to use different symbols to represent the end of each string.

Figure 4.3: The generalized suffix tree of $S_1 = acgat\#$ and $S_2 = cgt\$$

4.2.3 Construction of Suffix Tree

A suffix tree can be constructed in $O(n)$ time. Weiner[1] gave the first linear time construction algorithm for a constant size alphabet in 1973 however it requires a lot of memory. The first improvement to the original algorithm was developed by McGreight[2] in 1976, it uses quadratic space. Later in 1995, Ukkonen[3] presented a simplified online variant of the algorithm. In 1997, Farach[4] showed that even for a general alphabet, suffix tree can be constructed in linear time.

Ukkonen's algorithm is perhaps the most well known linear time algorithm for suffix tree construction. However, in this note, we will present a simple tree construction algorithm that takes $O(n^2)$ time. The algorithm is as follows: first initialize the tree with only a root node then incrementally insert $S[i..n]$ (where the entire string is $S[1..n]$, including the end of string) into the tree one by one for i from n down to 1. Each suffix is added by traversing down existing path, creating a new node for non-existing path and adding the remaining part of the string as the edge label. It will take $O(n)$ time for adding suffix $S[i..n]$. Given that there are n suffixes, the straightforward algorithm will take $O(n^2)$ time. Figure 4.4 show an example for constructing suffix tree for $S = acca\$$ and Figure 4.5 shows an example for constructing generalized suffix tree for $S = acca\$$ and $S' = c\#$.

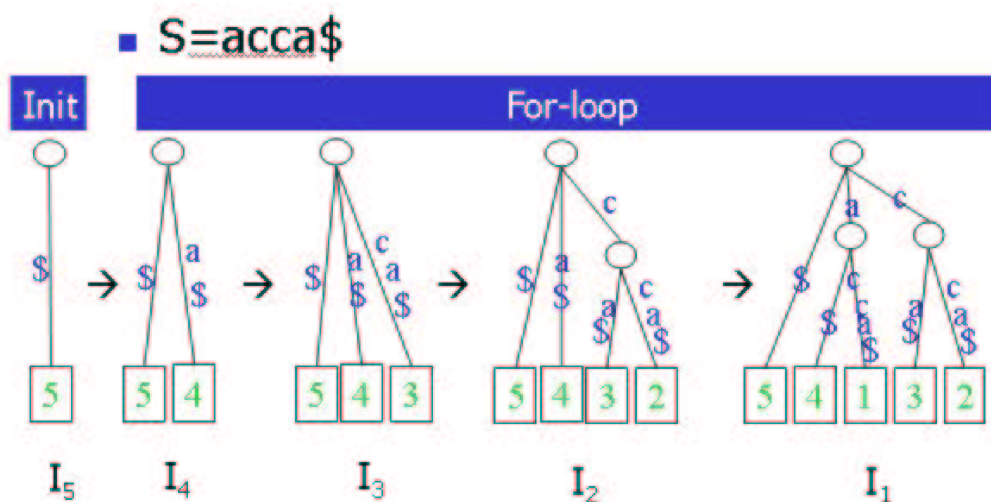


Figure 4.4: Example of constructing suffix tree for $S = acca\$$

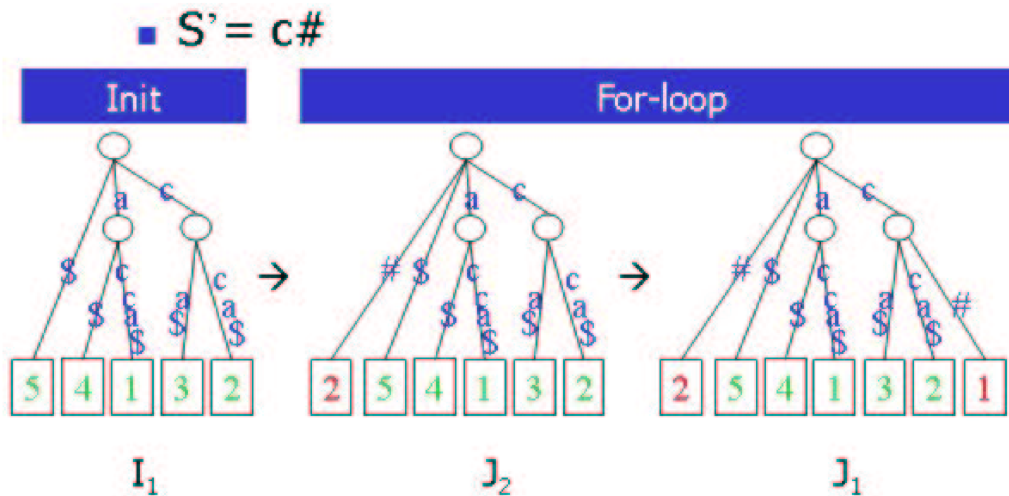


Figure 4.5: Example of constructing generalized suffix tree for $S = acca\$$ and $S' = c\#$

4.3 Applications of Suffix Tree

Suffix tree is a very powerful tool for solving many bioinformatics problems in linear time. Considering the massive amount of biological data in the form of sequences or strings, we can find many ingenious applications of suffix tree.

In this section, we will take a look at some of these applications of suffix tree. Some of them are variant of basic problems.

4.3.1 Exact String Matching Problem (ESM)

Applications in bioinformatics
In bioinformatics, ESM is use for finding patterns in DNA. For example, in gene signaling, scientist often want to identify the position of the codon *atg* (Methionine) which represents the start site for RNA transcription. ESM can then be used to identify the locations of *atg* within a very long DNA sequence.

Problem Definition
Given a string S of length n, for any query Q with length m, find all occurrences of Q in S.

Solution

1. Build a suffix tree for string S. This takes $O(n)$ time.

2. Starting from the root, traverse the suffix tree according to the path label to find the node x such that one of the prefixes of its path label is Q. If node x is not found, then Q is not in S. This step takes $O(m)$ time.

3. All the leaves in the subtree rooted at x are the occurrences of Q. To list them all, we need $O(occ)$ time, where occ is the number of occurrences of Q in S.

Thus, after the suffix tree is built in $O(n)$ time, it can be used to solved ESM problem in $O(m + occ)$ time. This is better than the best string matching algorithm which can solve ESM in $O(n)$ time since $m + occ << n$.

Example 1
Given $S = acacag$, find all occurrences of $Q = aca$ in S. $Q = aca$ is the prefix of suffix *acacag* (first suffix of S) and *acag* (third suffix of S). See Figure 4.6. Using suffix tree, we can find the answer in $O(m + occ)$ time.

Example 2
Given S = *acacag*, find all occurrences of Q = *acc* in S. In this case, when we traverse the suffix tree in Figure 4.6, we can find a path with label *ac*, but from this node we cannot find the path for the character *c*, thus we report that this pattern Q is not found in S.

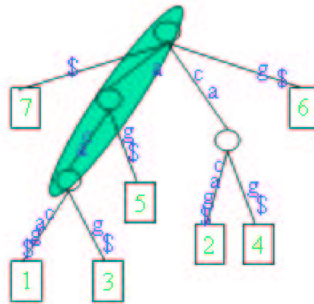


Figure 4.6: Finding *aca* in suffix tree of *acacag*\$

4.3.2 Longest Repeated Substring Problem (LRS)

Applications in bioinformatics
A gene sequence can contain repetitive substrings. These repeats can occur either adjacent to each other (tandem repeats) or far apart in the sequence. Some of the tandem repeats consist of very short subsequences, called microsatellites and today they have become the preferred markers in many genetic mapping efforts.

Longest Repeated Substrings are identical substrings in a sequence S, such that these substrings cannot be extended further without destroying the equality of the two substrings (they are maximal).

Finding these repetitive structures is an important problem in biological sequences. For example, most of the human Y chromosome consist of repeated subsequences. In addition, many diseases are caused by structural repeats.

In the past, without the use of suffix tree, computational biologist use an $O(n^2)$ time algorithm to solve LRS. With suffix tree, LRS can be solved in $O(n)$ time

Problem Definition
Given a string S of length n, find the longest repeated substring in S.

Solution

1. Build a suffix tree for the string S. This takes $O(n)$ time.
   In a suffix tree, the path label of each internal node is a repeat occurring at two (or more) locations.

2. Traverse the suffix tree to find the deepest internal node. This can be done in $O(n)$ time. The length of the repeat is the string depth of the deepest internal node.

Thus, suffix tree can be used to solve LRS problem in $O(n)$ time.

Example
Given $S = acacag$, the longest repeated substring of S is aca. As we can see in Figure 4.7, the deepest internal node is indeed the node with path label aca.

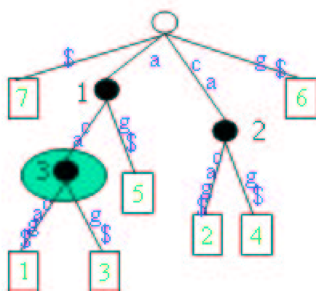


Figure 4.7: Suffix tree of $acacag\$$ with different internal node depth. aca is the deepest internal node.

4.3.3 Longest Common Substring Problem (LCS)

Applications in bioinformatics
Scientists are interested in finding similarity between two sequences. This problem can be modeled as finding the longest common substring between those sequences.

In 1970, before the existence of suffix tree, Don Knuth conjectured that a linear time algorithm for LCS is impossible. Now we know that it can be solved in linear time using suffix tree.

Problem Definition
Given two strings P_1 and P_2, find the longest common substring of these two strings. The total length of these two strings is n.

Solution

1. Build a generalized suffix tree for P_1 and P_2. This takes $O(n)$ time.
2. Mark each internal node with leaves representing suffixes of P_1 and P_2. This takes $O(n)$ time. This will ensure that the path label of the marked internal nodes are common substring of P_1 and P_2.
3. Report the deepest marked node. This step can be done in $O(n)$ time.

Example
Given $P_1 = acgat$, $P_2 = cgt$, the longest common substring of P_1 and P_2 is cg. See Figure 4.8.

Note
Longest Common <u>Substring</u> is not the same as Longest Common <u>Subsequence</u>!. Longest Common Subsequence find sequences of characters that need not be contiguous, whereas in Longest Common Substring, the characters must be contiguous!

4.3.4 DNA Contamination Problem

Applications in bioinformatics
DNA contamination is a common problem in bio engineering. Let S be a sequence reconstructed from a genome, says, C. elegans (worm). S is suspected to be contaminated by cloning vectors, PCR primers or genome of the host organism (e.g. yeast). How can we determine if S has been contaminated?

Problem Definition
Given a string S_1 (the sequenced string) and a string S_2 (the combined sources of

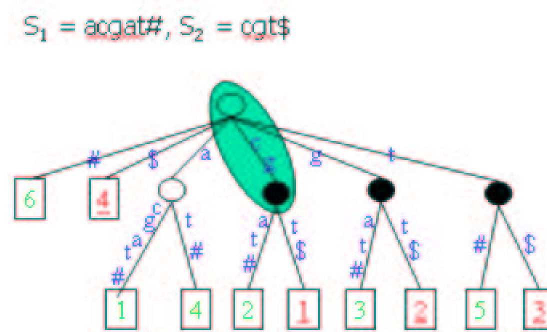


Figure 4.8: LCS of *acgat* and *cgt* is *cg*. Please note that longest common substring is not the same as longest common subsequence. For *acgat* and *cgt*, the longest common subsequence is *cgt*.

possible contaminator: cloning vectors, PCR primers, genome of host organism, etc). Find all substrings of S_2 occurring in S_1 which are longer than some given length x.

If some substrings of S_2 occurs in S_1, S_1 is likely to be contaminated!

Solution

1. Build a generalized suffix tree for S_1 and S_2. This takes $O(n)$ time.

2. Mark each internal node with leaves representing suffixes of both S_1 and S_2. This can be done in $O(n)$ time. This is will ensure that the path label of the marked internal nodes are common substring between S_1 and S_2

3. Report all marked nodes with string depth of x or greater. This takes $O(n)$ time.

As we can see, the approach for solving DNA contamination problem using suffix tree is just a variant of the approach that we use for LCS problem.

DNA contamination problem can be solved in $O(n)$ time.

Example
$S1 = acgatcttactacgatcctgttgaaccgggc$
$S2 = cgacctacggttgcgggat$
Determine if there are common substring longer than or equal to 4

Answer: Yes, S_1 is contaminated!
We found 3 common substrings with length ≥ 4

$S1 : acgatctta(\underline{ctacg})atcct(\underline{gttg})aac(\underline{cggg})c$
$S2 : cgac(\underline{ctacg})(\underline{gttg})(\underline{cggg})at$

Figure 4.9 shows a simple example of this solution.

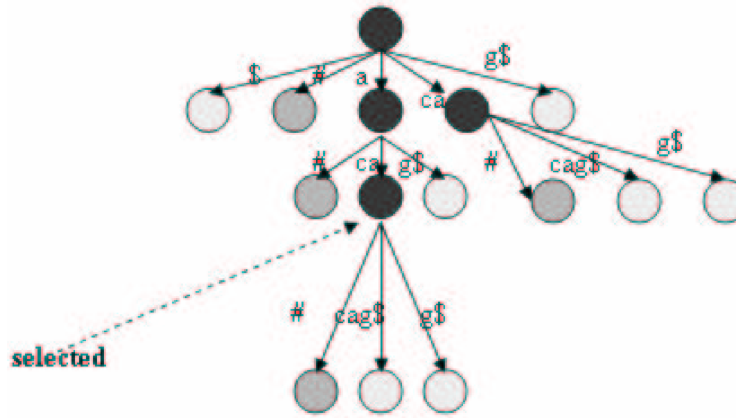


Figure 4.9: Example of DNA contamination problem. For $S_1 = acacag\$$, $S_2 = aca\#$, the set of substrings of S_2 of length greater than 2 that occurs in S_1 is $\{aca\}$

4.3.5 Common Substring of 2 or more strings

Applications in bioinformatics
Finding identical string between two or more sequences is an important issue in bioinformatics, such as identifying the conversed regions in two genomes. This is another variant of LCS problem.

Problem Definition
Consider K strings whose total length is n. For all $2 \leq k \leq K$, compute $l(k)$ which is the length of the longest substring common to at least k of these strings.

Solution

1. Assign distinct ending character to each string. This can be done in $O(k)$ time. This is to ensure that step 4 works correctly.

2. Build a generalized suffix tree T for all strings. This can be done in $O(n)$ time.

3. By traversing T, for each internal node v, compute its string depth. This takes $O(n)$ time.

4. By traversing T, for each internal node v, compute $C(v)$, where $C(v)$ is defined as the number of distinct terminating symbols in the subtree rooted at v. This can be done in $O(Kn)$ time.

5. Traverse T and visit every internal node v. For each v, if $l(C(v)) <$ string depth of v, set $l(C(v)) =$ string depth of v. $l(C(v))$ will store the length of the longest path label which are common to $C(v)$ strings. This step will take $O(n)$ time.

The overall time complexity is $O(Kn)$.

Example
Consider a set of 5 strings { sandollar, sandlot, handler, grand, pantry }

$l(4) = 4$ and the corresponding substring is "*and*" because "*and*" is the longest common substring for 4 strings: { s<u>and</u>ollar, h<u>and</u>lot, h<u>and</u>ler, gr<u>and</u> }, we omit *pantry* in this case.

$l(5) = 2$ and the corresponding substring is is "*an*" because "*an*" is the longest common substring for 5 strings: { s<u>an</u>dollar, h<u>an</u>dlot, h<u>an</u>dler, gr<u>an</u>d, p<u>an</u>try }

Using the same method you can check that $l(2) = 4$ and $l(3) = 3$. The corresponding substrings are "*sand*" and "*and*" respectively.

4.3.6 Remains of US military personnel

Applications in bioinformatics
The US military wants to identify their personnel's identity if he is killed. Thus, from every US army soldiers, a small interval of his DNA is extracted (which are likely to be unique for different people) and a database of all these DNA intervals is formed. When someone is killed, we can extract his DNA from his remains. However, due to the condition of the remain, we may not be able to get the complete information of the desired DNA interval and may only get a substring S. We wish to find all strings in the database containing S as a substring to help us determine the identity of the soldier.

Problem Definition
The remains of US military personnel can be formalized as the substring problem for a database of patterns. Given a set of strings (database) of total length n. For any query string Q with length m, find all strings in the database containing Q as a substring.

Solution

1. Build a generalized suffix tree T for all strings in the database.
2. Find all occurrences of Q in T using ESM.

Preprocessing takes $O(n)$ time while each query can be solved in $O(m+occ)$ time where occ is the number of occurrences of Q in T.

4.3.7 Longest Common Prefix (LCP)

Applications in bioinformatics
Solving LCP allows us to reduce the complexity for finding the common substring of 2 or more strings from $O(Kn)$ to $O(n)$.

Problem Definition
Given a string S of length n, for any i,j, find the length of the longest common prefix of suffixes i and j of S.

Solution
The key idea is that the longest common prefix of suffixes i and j is equal to the path label of the lowest common ancestor of suffixes i and j. This reduces the problem to finding the lowest common ancestor (LCA) of two nodes in a tree. The solution for LCA was first obtained by Harel and Tarjan [5] and later simplified by Schieber and Vishkin [6]. The solution is further simplified by Bender and Farach [13] in 2000. Their solutions allows us to preprocess the tree in $O(n)$ time and create a data structure that can answer LCA queries in $O(1)$ time.

1. Build suffix tree for string S. This takes $O(n)$ time.
2. Build the LCA data structure on the suffix tree. This can be done in $O(n)$ time.
3. Return the path label of the lowest common ancestor of suffix i and j. This can be done in $O(1)$ time.

Time complexity is $O(n)$ for preprocessing and $O(1)$ for each processing of suffix i and j.

4.3.8 Finding Palindrome

Applications in bioinformatics
In DNA a complemented palindrome is a sequence of base pairs that reads the same backwards and forward across the double strand. The enzymes that cut these specific sites are called restriction enzymes. Therefore by looking for complemented palindromes we can identify the binding sites for restriction enzymes.

Definitions

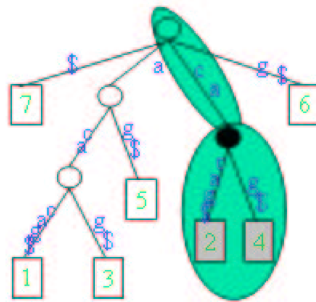


Figure 4.10: Longest common prefix of suffix 2 and 4 is *ca*

Palindrome: Given a string S, a palindrome is a substring u of S, such that $u = u^r$, where u^r denotes the reverse string of u. For example, *acagaca* is a palindrome as $acagaca = (acagaca)^r$.

1. If $S[i..i+k-1] = S^r[n-i+1..n-i+k]$, then $u = S[i-k+1..i+k-1]$ is an odd length palindrome (see Figure 4.11)

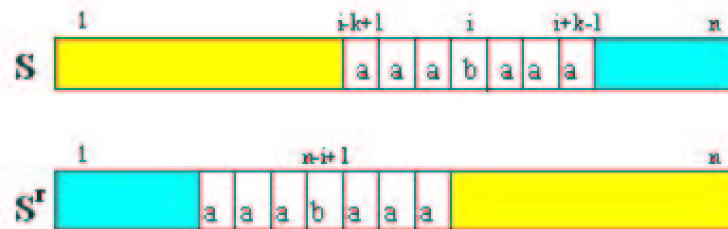


Figure 4.11: odd length palindrome

2. If $S[i..i+k-1] = S^r[n-i+2..n-i+k+1]$, then $u = S[i-k..i+k-1]$ is an even length palindrome. (see Figure 4.12)

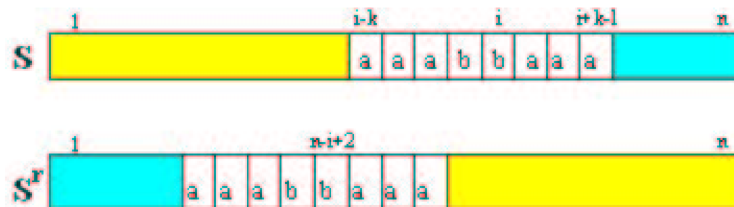


Figure 4.12: even length palindrome

Complemented palindrome: Given a string S, a complemented palindrome is a substring u of S, such that $u = \bar{u}^r$.where $\bar{u}$ is complement of u, e.g. *acaugu* is a complemented palindrome.

Maximal palindrome: Given a string S, a palindrome $u = S[i..i + |u| - 1]$ is called a maximal palindrome of S, if $S[i'..j']$ is not a palindrome for any $i' < i$ and $j' > i + |u| - 1$

With this definition of maximal palindrome, every palindrome is contained in a maximal palindrome. In other word, maximal palindromes are a compact way to represent all palindromes.

We can define maximal complemented palindrome similarly.

Problem Definition

1. Given a string S of length n, which represents the genome, the problem is to locate all maximal palindromes in S.

2. Given a string S of length n, which represents the genome, the problem is to locate all maximal complemented palindromes in S.

In this section, we only give a solution for problem 1, that is to find maximal palindromes. Note that maximal complemented palindromes can be found in a similar manner.

Solution
Preprocess S and S^r by building their suffix tree in $O(n)$ time so that any longest common prefix query can be answered in constant time.

For $i = 1 \to n$,

1. Find the longest common prefix for (S_i, S^r_{n-i+1}) in $O(1)$ time. If the length of the longest prefix is k, we have found an odd length maximal palindrome $S[i - k + 1..i + k - 1]$.

2. Find the longest common prefix for (S_i, S^r_{n-i+2}) in $O(1)$ time. If the length of longest prefix is k, we have found an even length maximal palindrome $S[i - k..i + k - 1]$.

The preprocessing takes $O(n)$ time and each longest common prefix query can be answered in $O(1)$ time. So we can find all the maximal palindromes in $O(n)$ time.

4.4 Suffix Array

Although suffix tree is a very useful data structure, it has very limited usage in practice. This is mainly due to its large space requirement of order $O(n|\Sigma|\log n)$ bits. For DNA, the size of $|\Sigma|$ is 4.

To solve this problem, Manber and Myers [7] proposed a new data structure called suffix array, in 1993, which has a similar functionality as suffix tree but only requires $n \log n$ bits space. For DNA, suffix array implementation should be roughly 4 times smaller than suffix tree implementation for the same string.

4.4.1 Definition

Let $S[1..n]$ be a string of length n over an alphabet Σ. We assume that $S[n] = \$$ is a unique terminator which is alphabetically smaller than all other characters. For any $i = 1, 2, \ldots, n$, $S[i..n]$ is a suffix of S. The suffix array ($SA[1..n]$) stores the suffixes of S in a lexicographically increasing order. Formally, S is an array of integers such that $S[SA[i]..n]$ is lexicographically the i-th smallest suffix of S. For example, consider $S = acacag\$$. Its suffix array is shown in Figure 4.13.

Each integer in the suffix array is less than n and can be stored using $\log n$ bits. Thus, the whole suffix array of size n can be stored in $n \log n$ bits.

Suffix	Position
$acacag\$$	1
$cacag\$$	2
$acag\$$	3
$cag\$$	4
$ag\$$	5
$g\$$	6
$\$$	7

i	$SA[i]$	Suffix
1	7	$\$$
2	1	$acacag\$$
3	3	$acag\$$
4	5	$ag\$$
5	2	$cacag\$$
6	4	$cag\$$
7	6	$g\$$

Figure 4.13: The suffixes and suffix Array for $S = acacag\$$

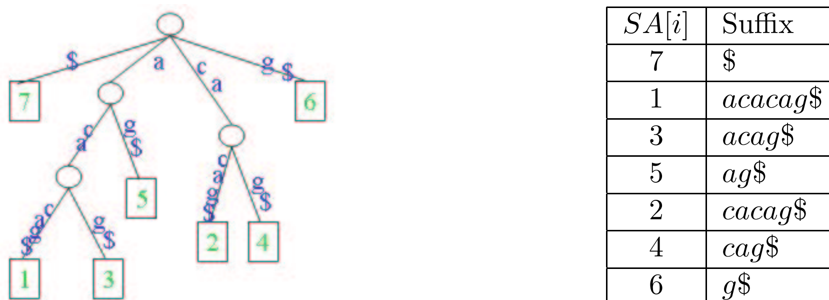
4.4.2 Construction of Suffix Array

Observe that the leaves of a suffix tree, if traversed in lexicographical depth-first order, would form the suffix array of that string. This is shown in Figure 4.14. Thus suffix tree T of $S[1..n]$ can be constructed in $O(n)$ time by first constructing the suffix tree T and then general the suffix array by traversing T using lexicographical depth-first traversal.

However, this naïve approach requires a large working space, because to build

suffix tree itself, we already require $O(n|\Sigma|\log n)$ bits space, thus this defeats the purpose of using suffix array.

To date, if we only have $O(n)$ bits of working memory available, the best known technique for constructing suffix array takes $O(n)$ time. Refer to [12] for more details.



$SA[i]$	Suffix
7	$
1	$acacag\$$
3	$acag\$$
5	$ag\$$
2	$cacag\$$
4	$cag\$$
6	$g\$$

Figure 4.14: The suffix tree and suffix array for $S = acacag\$$

4.4.3 Applications

Most applications using suffix tree can be solved using suffix array with some overhead. In this section, we will give one example of solving Exact String Matching (ESM) problem using suffix array.

Input:

1. The suffix array of a string S of length n
2. A query Q of length m.

Problem: Find all occurrences of Q in S.
Solution:
The idea is to perform a binary search on the suffix array of S. We can do binary search because suffix array is an array of sorted suffixes. See Figure 4.15

1. Let L and R be the left and right boundaries of SA such that suffix$_{SA[L]} \leq Q \leq$ suffix$_{SA[R]}$ (initially, $L = 1$ and $R = n + 1$, because the number of items in suffix array is $n+1$ (including the terminating symbol))

   if $(L > R)$, output Q not found.

   Let l be the length of the longest common prefix of Q and suffix$_{SA[L]}$.
   Let r be the length of the longest common prefix of Q and suffix$_{SA[R]}$.

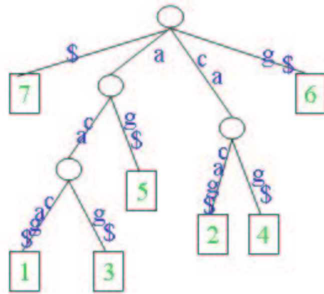
2. Let $M = (L + R)/2$, let $mlr = min(l, r)$
   Since suffix array is sorted, if we have mlr prefix matches of Q with suffix$_{SA[L]}$ and suffix$_{SA[R]}$, then the first mlr characters of suffix$_{SA[M]}$ and Q are also the same.

   Then, starting from the position mlr, find the last match position x.

   If $x = |Q|$, stop the algorithm. Suffixes $SA[M]$ contains the occurrence of Q. To find all the occurrences, from position L to R, list down all the suffixes with Q as their prefixes.

   Otherwise, find the lexicographical order of suffix$_M$ and Q by comparing suffix$_{SA[M]}[x + 1]$ and $Q[x + 1]$
   If suffix$_{SA[M]}> Q$, set ($R = M$ and $r = x$) else set ($L = M$ and $l = x$).

3. Goto 2.



$SA[i]$	Suffix	Binary search
7	$	(1)(2)
1	*acacag*$	(-)(2)(3), found *aca*
3	*acag*$	(-)(-)(3), found *aca*
5	*ag*$	(1)(2)
2	*cacag*$	(-)
4	*cag*$	(-)
6	*g*$	(1)

Figure 4.15: Suffix tree, suffix array and executing of binary search for *aca* in *acacag*

Using binary search on a suffix array of size n, we will performed at most $\log n$ comparisons. Each comparison takes at most $O(m)$ time. Therefore, in the worst case, the algorithm takes $O(m \log n + occ)$ time. However, it is reported that in practice, the actual time complexity is $O(m + \log n + occ)$ [7].

For solving ESM, this $O(m \log n + occ)$ time complexity using suffix array is slightly worse than $O(m + occ)$ time complexity of using suffix tree. However, when space is limited, suffix array is a good alternative to suffix tree.

4.5 FM-index

Although the space requirement of suffix array is less than that of suffix tree, it is still unacceptable in many real life situations. For example, to store the human genome, with its 3 billion base pairs, it takes up to 40 Gigabytes storage using

suffix tree and 13 Gigabytes using suffix array. This is clearly beyond the capacity of a normal PC. More space-efficient solution is required for practical solutions.

Two such alternatives were proposed, they are Compressed Suffix Array by Gross and Vitter [9] and FM-index by Ferragine and Manzini [10]. Both data structures only require $O(n)$ bits where n is the length of the string. Now we can store human genome data structure within 2 Gigabytes.

In the following section we will introduce the FM-index data structure.

4.5.1 Definition

FM-index is a combination of Burrows-Wheeler (BW) text [11] and an augmented structure to enable random access of the compressed data.

Let $S[1..n]$ be a string of length n, and SA be its suffix array. FM-index stores the following three data structures:

1. The BW is a string of characters defined as

$$BW[i] = \begin{cases} S[SA[i]-1] & (\text{if } SA[i] \neq 1) \\ S[n] & (\text{if } SA[i] = 1) \end{cases}$$

2. $C[x]$ array stores the total number of occurrences of each character which is lexicographically less than x.

3. A data structure which supports $O(1)$ computation of $occ(x, 1, i)$, where $occ(x, 1, i)$ is the number of occurrences of x in BW$[1..i]$. The details of this data structure is omitted.

For example, consider $S = acacag\$$. Its BW text is shown in Figure 4.16 and $C[a] = 1$, $C[c] = 4$, $C[g] = 6$, and $C[t] = 7$. We will now analyze the space

i	$SA[i]$	Suffix	$S[SA[i]-1]$
1	7	\$	g
2	1	$acacag\$$	\$
3	3	$acag\$$	c
4	5	$ag\$$	c
5	2	$cacag\$$	a
6	4	$cag\$$	a
7	6	$g\$$	a

Figure 4.16: Suffix array suffix and BW text for $S = acacag\$$

complexity of FM-index. In the case of DNA sequence, structure 1 can be stored $2n$ bits because we are storing n characters and they are 4 possible choices for each characters ($\log 4 = 2$). Structure 2 can be stored in $O(\log n)$ bit and structure 3 can be stored in $O(\frac{n \log \log n}{\log n})$ bits. Therefore, in total the size of FM-index is $O(n)$ bits.

4.5.2 Backward Search

One of the applications of FM-index is to determine whether a query string Q exists in string S, using backward search algorithm.

An important property of the FM-index which allows for efficient searching is that $C[x]+occ(x,1,i)$ is equal to the number of suffixes smaller than $xT[SA[i]..n]$, where $x \in \Sigma$. For example, for the FM-index of *acacag*\$, the number of suffixes smaller than $cT[SA[5]..n] = ccag\$$ is equal to $C[c] + occ(c,1,5) = 4 + 2 = 6$.

Input: The FM-index of S, query string Q of length m.
Aim: Determine whether Q exists in S.
Algorithm BW_exist($Q[1..m]$)

1. $c = Q[m]$, $i = m$
2. $sp = C[c] + 1$, $ep = C[c+1]$
3. while ($sp \leq ep$ and $i > 1$)
   - $c = Q[i-1]$;
   - $sp = C[c] + occ(c, 1, sp-1) + 1$;
   - $ep = C[c] + occ(c, 1, ep)$;
   - $i = i - 1$;
4. if $sp > ep$, then pattern is not found else the pattern is found.

Now we analyze the time complexity of backward search. To find a pattern $Q[1..m]$, we need to compute $i = m \rightarrow 1$ in the while loop, which iterates $m-1$ times. Each iteration of the loop can be computed in $O(1)$ time. Therefore, the whole backward search algorithm takes $O(m)$ time.

For example, let $S = acacag\$$ and $Q = aca$. Figures 4.17 to 4.19 shows the

computation.

First iteration (initial values),
$Q[3..3] = a$
$c =' a'$
$sp = C[a] + 1 = 1 + 1 = 2$
$ep = C[c] = 4$

	$SA[i]$	Suffix	$S[SA[i]-1]$
	7	$	g
→	1	$acacag$$	$
	3	$acag$$	c
→	5	$ag$$	c
	2	$cacag$$	a
	4	$cag$$	a
	6	$g$$	a

Figure 4.17: First iteration of searching Q in S

Second iteration,
$Q[2..3] = ca$
$c =' c'$
$sp = C[c] + occ(c, 1, sp_{old} - 1) + 1$
$= 4 + 0 + 1 = 5$
$ep = C[c] + occ(c, 1, ep_{old})$
$= 4 + 2 = 6$

	$SA[i]$	Suffix	$S[SA[i]-1]$
	7	$	g
	1	$acacag$$	$
	3	$acag$$	c
→	5	$ag$$	c
	2	$cacag$$	a
→	4	$cag$$	a
	6	$g$$	a

Figure 4.18: Second iteration of searching Q in S

Third iteration,
$Q[1..3] = aca$
$c =' a'$
$sp = C[a] + occ(a, 1, sp_{old} - 1) + 1$
$= 1 + 0 + 1 = 2$
$ep = C[a] + occ(a, 1, ep_{old})$
$= 1 + 2 = 3$
Pattern Q is found in S

	$SA[i]$	Suffix	$S[SA[i]-1]$
	7	$	g
→	1	$acacag$$	$
→	3	$acag$$	c
	5	$ag$$	c
	2	$cacag$$	a
	4	$cag$$	a
	6	$g$$	a

Figure 4.19: Third iteration of searching Q in S

4.6 1-Mismatch Problem

4.6.1 Motivation

Suffix tree and its variants are powerful data structures which has a lot of applications in Computational Biology. However it has two major problems. Firstly

it requires a large amount of memory and secondly it can only be used to solve exact match problems. Fortunately new data structures such as Compressed Suffix Array or FM-index can reduce the memory needed dramatically. The real difficulty now lies in making use of these data structures to solve approximate match problem, which are more realistic in the biological setting.

4.6.2 Problem Definition

Given a pattern $P[1..m]$, the 1-mismatch problem finds all occurrences of P in text $T[1..n]$ that has hamming distance at most 1. For example, $P = acgt$ and $T = a\underline{acgt}ggcca\underline{actt}gga$ the underlined substrings of T are the 1-mismatch occurrences of P.

4.6.3 Naïve Solution

A naïve solution for solving the 1-mismatch problem is to construct the suffix tree for T and generate all possible 1-mismatch of P and find the occurrences of every 1-mismatch pattern in the suffix tree. The number of possible 1-mismatch of P is $(|\Sigma| - 1)m$ and it takes $O(m)$ time to find occurrences of each 1-mismatch pattern. Hence this algorithm takes $O(|\Sigma|m^2 + occ)$ time in total, where occ is the total number of occurrences.

4.6.4 Trinh et al's Solution

More sophisticated solutions for solving the 1-mismatch problem has been proposed in the literature. In this note we will present the algorithm by Trinh et al [14] that solves this problem using an $O(n \log n)$ bit index and each query takens $O(|\Sigma|m \log n + occ)$ time.

4.6.4.1 Index

The data structures used in [14] are the suffix array (SA) of T and the inverse suffix array (SA^{-1}) of T, where $SA[SA^{-1}[i]] = i$. In other words, the suffix array is a mapping from the position in the lexicographical order to the index of the suffix while the inverse suffix array is a mapping from the index of the suffix to its position in the lexicographical order. See Figure 4.2 for an example of the suffix array and inverse suffix array of $S = acacag\$$.

4.6.4.2 Algorithm

The algorithm for finding all the 1-mismatch patterns using the suffix array and inverse suffix array is based on the following 3 lemma.

i	$SA[i]$	$SA^{-1}[i]$	Suffix
1	7	2	\$
2	1	5	$acacag\$$
3	3	3	$cacag\$$
4	5	6	$acag\$$
5	2	4	$cag\$$
6	4	7	$ag\$$
7	6	1	$g\$$

Table 4.2: The suffix array and inverse suffix array of $S = acacag\$$

Lemma 4.1 *(Forward Search) Suppose* $[st..ed] = range(T, P)$*, then we can compute* $[st'..ed'] = range(T, Pc)$ *in* $O(logn)$ *time.*

Proof: Use binary search on suffix array of T. ■

Lemma 4.2 *Suppose* $[st_1..ed_1] = range(T, P_1)$ *and* $[st_2..ed_2] = range(T, P_2)$ *then we can compute* $[st..ed] = range(T, P_1P_2)$ *in* $O(\log n)$ *time.*

Proof:
Let the length of P_1 be k
Observe that $T_{SA[st_1]}, T_{SA[st_1+1]}, ..., T_{SA[ed_1]}$ are lexicographically increasing.
Hence, $T_{SA[st_1]+k}, T_{SA[st_1+1]+k}, ..., T_{SA[ed_1]+k}$ are also lexicographically increasing.
Thus $SA^{-1}[SA[st_1] + k] < SA^{-1}[SA[st_1 + 1] + k] < ... < SA^{-1}[SA[ed_1] + k]$.
Therefore to find st and ed, we need to find

- the smallest st such that $st_2 < SA^{-1}[SA[st] + k] < ed_2$ and
- the largest ed such that $st_2 < SA^{-1}[SA[ed] + k] < ed_2$

This can be done using binary search. ■

Lemma 4.3 *(Backward Search) Suppose* $[st..ed] = range(T, P)$*, then we can compute* $[st'..ed'] = range(T, cP)$ *in* $O(\log n)$ *time.*

Proof: Let $P_1 = c$ and $P_2 = P$. By Lemma 4.2, $range(T, P_1, P_2)$ can be computed in $O(\log n)$ time. ■

The algorithm is as follows:

1. For $j = m \rightarrow 1$
   - By backward search, find $range(T, P[j..m])$
2. For $j = 1 \rightarrow m$

- By forward search, find $range(T, P[1..j])$

3. Report all occurrences of $range(T, P[1..m])$

4. For $j = 1 \to m$
   - Let $P_1 = P[1..j-1], P_2 = P[j+1..m]$
   - For every character $c \neq P[j]$
     - By forward search, find $range(T, P_1 c)$
     - By Lemma 2, find $range(T, P_1 c P_2)$
     - Report all occurrences of $range(T, P_1 c P_2)$

4.6.4.3 Time Complexity

Reporting the occurrences take $O(occ)$ time. Observe that Steps 1 and 2 takes $O(m \log n)$ time each. In Step 4 we enumerate all possible $(|\Sigma|-1)m$ mismatches, for each mismatch it takes $O(\log n)$ time to find the occurrences. Thus in total Step 4 takes $O(|\Sigma| m \log n)$ time. Hence the time complexity of the algorithm is $O(|\Sigma| m \log n + occ)$. Note that this algorithm can be generalized to handle k-mismatch in $O(|\Sigma|^k m^k \log n + occ)$ time.

4.7 Concluding Remarks

In conclusion, suffix tree and suffix array are powerful data structures for solving problems involving exact string matching. The current research direction is in extending these data structures to create practical algorithms for solving approximate match problems.

References

[1] P. Weiner. Linear Pattern Matching Algorithms. *Switching and Automata Theory*, 1-11, 1973.

[2] E. M. McGreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of ACM*, 23:262-272, 1976.

[3] E. Ukkonen. On-line Construction of Suffix Trees; *Algorithmica*, 14:249-260, 1995.

[4] Martin Farah. Optimal suffix tree Construction with Large Alphabets. *FOCS*, 1997.

[5] D. Harel and R. E. Tarjan. Fast Algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.

[6] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplifications and Parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.

[7] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[8] J. Larsson and K. Sadakane. Faster Suffix Sorting. Technical Report Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Lund University, 1999.

[9] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proc. ACM STOC*, pages 397–406, 2000.

[10] P. Ferragine and G. Manzini. Opportunistic Data Structures with Applications. In *Proc. IEEE FOCS*, pages 390–398, 2000.

[11] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, California, 1994.

[12] Wing-Kai Hon, Kunihiko Sadakane and Wing-Kin Sung. Breaking a Time and Space Barrier in Constructing Full-Text Indices. In *Proc. IEEE FOCS*, 2003.

[13] Michael A. Bender and Martin Farach-Colton The LCA Problem Revisited. In Latin American Theoretical INformatics, pages 88–94, 2000.

[14] Trinh N. D. Huynh, W. K. Hon, T. W. Lam and W. K Sung. Approximate String Matching Using Compressed Suffix Arrays. In *Proc. CPM*, 2004.