Implementation and Analysis of different Algorithms to find Tandem Repeats

Prithvi Monangi (1685 8394), Mohan Konyala (5005 3565), Deepak Addepalli (0620 0926)
Bio Informatics
University of Florida
Gainesville
pmonangi@ufl.edu, konyala@ufl.edu, deepak246@ufl.edu

***Abstract* — Tandem repeats are segments of short DNA repeated multiple times consecutively. They are believed to play a prominent role in regulating gene expression. Tandem repeats also have a much higher rate of variation than the rest of the genome (in terms of the number of copies), and this makes them ideal markers to distinguish one individual from another. It is therefore very important to have an efficient algorithm to identify if the given pattern is a tandem repeat in a given DNA sequence. In this paper, we discuss three different paradigms to solve this problem.**

Keywords: Tandem repeats, Suffix trees, Suffix Arrays, Dynamic Programming.

I. INTRODUCTION

Tandem repeats occur in DNA when a pattern of one or more nucleotides is repeated and the repetitions are directly adjacent to each other. An example would be, in ACGTCGTCGT, CGT is repeated three times. Tandem repeats describe a pattern that helps determine an individual's inherited traits. In the field of Computer Science, tandem repeats in strings can be efficiently detected using several techniques.

In this project, we will discuss three different algorithms based on Suffix trees, Suffix arrays and Dynamic programming to check if the pattern(substring) is a tandem repeat in a given DNA sequence or not. For suffix tree based method, we first discuss a linear time construction algorithm for Suffix tree and then discuss checking if the pattern is a tandem repeat or not, also in linear time. Then we proceed to discuss the construction of suffix arrays and finding the tandem repeats using it. Next, we talk about a dynamic programming technique to solve the same problem. Finally, we do a comparison of all the discussed algorithms and identify the pros and cons of each of them.

II. SUFFIX TREE-BASED ALGORITHM

Suffix Tree is very useful in numerous string processing and computational biology problems. We will discuss Ukkonen's Suffix Tree, a linear time construction algorithm. A suffix tree **T** for m-character string S is a rooted directed tree with exactly **m** leaves numbered 1 to **m** will have the following properties.

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of S that starts at position i, i.e. S[i…m]. We add $ to the end of string to handle cases when a suffix is prefix of another suffix.

A naïve algorithm to construct a suffix tree takes O (n^2) time. We shall discuss Ukkonen's algorithm which constructs the tree in linear time.

A. *High Level Description of Ukkonen's algorithm*

Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix S [l...i] of S (Sequence of length m).It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, T_m using mth character. Implicit suffix tree T_i+1 is built on top of implicit suffix tree T_i. The true suffix tree for S is built from T_m by adding $ at the end. At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property. Time taken is O (n).

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m). In phase i+1, tree T_i+1 is built from tree T_i. Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S [1...i+1]. In extension j of phase i+1, the algorithm first finds the end of the path from the root labelled with substring S [j...i]. It then extends the substring by adding the character S (i+1) to its end if it is not there already. The pseudo code below expresses the above steps in a concise way.

```
▪ Construct the tree T1
▪ For i from 1 to m-1 do
▪ begin {phase i+1}
    ▪ For j from 1 to i+1
        ▪ begin {extension j}
        ▪ Find the end of the path from
          the root labelled S[j..i] in the
          current tree.
        ▪ Extend that path by adding
          character S[i+l] if it is not
          there already
    ▪ end;
end;
```

If the path labels are represented as characters in string it will take $O(n^2)$ space to store the path labels. To avoid this, we can use pair of indices (start, end) on each edge for path labels, instead of substring itself. With this, suffix tree needs O(n) space. Apart from this, the algorithm uses suffix links, active points and few other tricks to keep track of existing suffixes and add a new node only if necessary. Suffix links essentially provide a shortcut to add new characters into the tree.

B. *Pattern search in Suffix Trees:*

1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
- For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.
- If there is no edge, print "pattern doesn't exist in text" and return.
2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print "Pattern found".

C. *Tandem repeat search in suffix trees:*

Based on the suffix tree constructed so far, to find a tandem repeat the key observation would be following:

For each internal node of the tree if the difference between any two of it's leaf node indices is equal to the pattern length i.e., the node depth then we can say that at that two indices the pattern is repetitive and adjacent (Tandem Repeat). We find all such adjacent pattern indices for a given pattern to find its tandem repeats.

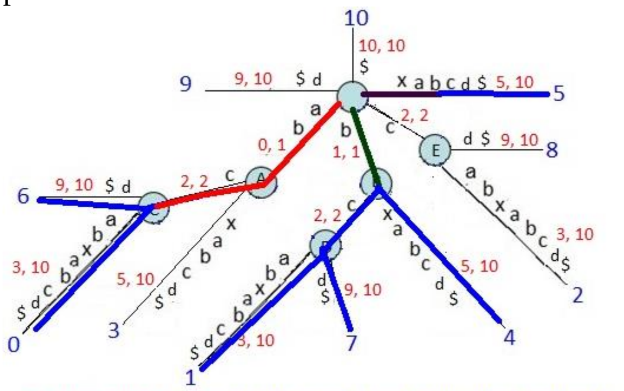
To search for tandem repeats in a suffix tree given a pattern of size 'm', we follow the following 3 steps:

1) *First of all, we check if the given pattern really exists in string or not. For this, traverse the suffix tree against the pattern which takes O (m) time.*

2) *If you find the pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string*

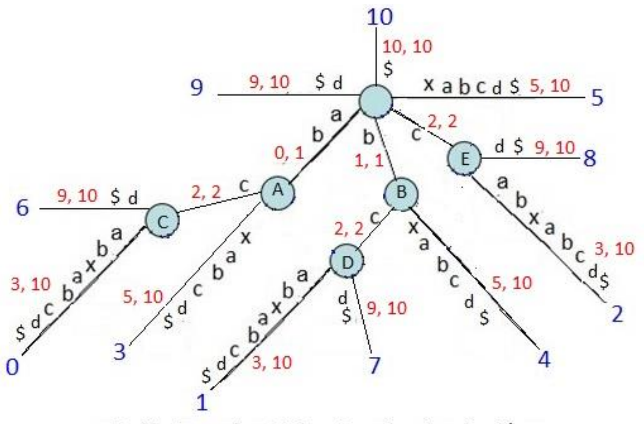
3) *Now, check the indices if they differ by a count of the size of the pattern, it is considered as a tandem repeat. Have a count of all such consecutive repeats which is the tandem repeat count of the given pattern against the given sequence.*

The figure 2 below shows the suffix tree structure which helps us find all the occurrences of a given pattern.



1. Substring abc is found, subtree traversal shows that it is at indices 0 and 6.
2. Substring b is found, subtree traversal shows that it is at indices 1, 4 and 7.
3. Substring xab is found, subtree traversal shows that it is at index 5.

Figure 2



Suffix Tree for String S = abcabxabcd$

Figure 3

Consider the string abcabxabcd, its suffix array would be 0 6 3 1 7 4 2 8 9 5.

III. Suffix Array Based Algorithm

Suffix trees and **suffix arrays** are data structures for representing texts that allow substring queries like "where does this pattern appear in the text" or "how many times does this pattern occur in the text" to be answered quickly. Both work by storing all suffixes of a text, where a *suffix* is a substring that runs to the end of the text. Of course, storing actual copies of all suffixes of an n-character text would take O(n^2) space, so instead each suffix is represented by a pointer to its first character in case of Suffix arrays.

A suffix array stores all the suffixes sorted in dictionary order. The actual contents of the array are the indices in the left-hand column; the right-hand shows the corresponding suffixes.

The time complexity of naive method to build suffix array is O(n^2logn) if we consider a O(nlogn) algorithm used for sorting. Alternatively, having constructed a Suffix tree using Ukkonen's algorithm, we can construct a Suffix Array in linear time by doing a lexicographic order Depth First Search traversal and storing all the suffix indices in resultant suffix array, except the very 1st suffix index. Because a suffix tree of string of length n will have at most n-1 internal nodes and n leaves. Traversal of these nodes can be done in O(n) for a string of length n.

A. *Tandem repeat search in suffix arrays:*

Suppose we have a suffix arrays corresponding to an n-character text sorted in lexical order and we want to find all occurrences in the text of an m-character pattern. Since the suffixes are ordered, we can do a binary search for the first and last occurrences of the pattern (if any) using O(log n) comparisons. Each comparison may take as much as O(m) time, since we may have to check all m characters of the pattern. So the total cost will be O(m log n) in the worst case.

To find the tandem repeats we go through all the pattern indices and apply the tandem repeat principle as discussed above.

IV. Dynamic Programming Based Algorithm

To motivate the use of a dynamic programming, we have made a modification to the Smith-Waterman method for local alignment can be used exactly one time to locate all the repeats within a string. Since we are filling the values in an n x n matrix, this method has time and space complexity as O(n^2) when compared to brute force method which yields O(n^6) time algorithm.

The main idea is to align the given sequence with a copy of itself and compute the best local alignment ending at every possible point. The modifications to the dynamic programming matrix for finding the tandem repeats are:

1. Place the given string both on top and to the left of matrix to align the string with itself,
2. Set all the diagonal elements of the matric to 0 to avoid aligning characters with themselves,
3. Compute only the upper triangular matrix since both the strings are identical *(M[i,j]=M[j,i])*.

Pseudo code:

```
for i=0 to n
  M[0,i]=0;
  M[i,i]=0;
for i=1 to n
  for j=i+1 to n
    M[i.j]=max{M[i-1,j-1]+s(x_i,x_j),M[i-1,j]+s(x_i,-), M[i,j-1]+s(-,x_j),0}
```

where M is the matrix and s(a,b) is the scoring function defined on all characters in the string and a gap penalty s(a,-), s(-,a) defined for all characters in the string. The following is the definition for the scoring function which was used:

$$s(a,b)=\begin{cases}1 & \text{if } a=b\\ -1 & \text{otherwise}\end{cases} \qquad s(a,-)=s(-,a)=-1$$

V. Results

A. Input Data:

The input data has been collected from Gene bank NCBI website. We used the input with varying number of genome expression characters and observed the run times for all three algorithms mentioned above. Some of the input include GenBank: AC139763.4, AH003105.2, DQ112151.1 etc.

Then, we discussed on the run times observed and the reasons behind it. Finally, we conclude the paper by giving the comparison among the results of different algorithms described so far.

B. Implementation and Testing

We have implemented the three algorithms using c language. After implementing the algorithms each of us tested them on the c99 compiler. The algorithms of suffix array and suffix tree based takes input of the string and the pattern (pattern itself should not be a tandem repeat ex: AGAG) to find whether the given pattern is a substring of the input string or not, if yes it prints all the indices where the pattern is a tandem repeat. The dynamic programming approach takes the input string and gives indices of tandem repeats.

C. Time complexities

	Suffix tree	Suffix Array	DP
Construction	O(n)	O(n)	O(n^2)
Pattern Search + Tandem repeat check	O(m+k)	O(m*log n + k)	O(n^2)
Total time	O(n+m+k)	O(n + m*log n+k)	O(n^2)
Space Complexity	O(n)	O(n)	O(n^2)

Table 1

n - Size of Sequence, m - Size of pattern(m << n), k - Number of pattern occurences in the sequence

D. Test results comparision using graph

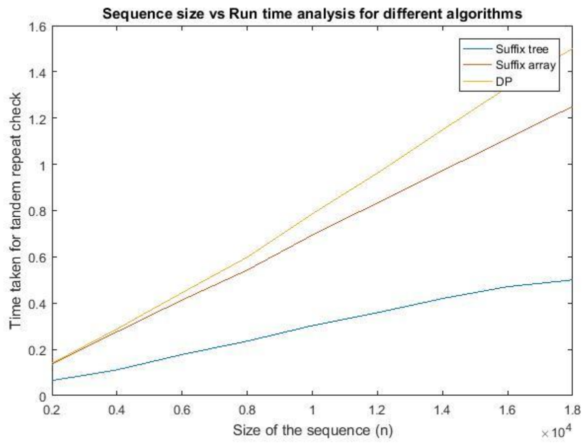


Figure 4

E. Conclusion

Suffix Array based algorithm runs in linear time but it takes some additional time when compared to that of Suffix Tree based one because to generate Suffix array in linear time we are first constructing suffix tree so it will differ by O(n) more time.

From Table 1 and Figure 4 we can compare the run times of the three algorithms. Dynamic programming is not very optimal as n value gets large because of $O(n^2)$ time complexity. The other two algorithms both run linear in the size of their input but Suffix Array comparatively takes less memory space as we are freeing the memory occupied by suffix tree (visit the tree in post order and free the memory) after construction of suffix array and only store the indices of suffix arrays in lexical order. So, based on the space and time parameters available we can use the suffix tree algorithm (more space - less time) or suffix array algorithm (less space - more time) accordingly. There are other efficient algorithms which helps to construct the Suffix Arrays in linear time without help of suffix trees, using them would save both space and time.

F. *Work Distribution:*

The work done for this project is equally distributed among three of us. We parallelly worked on implementation, Prithvi worked on Suffix array based algorithm, Mohan worked on Suffix tree based algorithm while Deepak did the Dynamic Programming based algorithm. Mohan and Prithvi collected sample DNAs from Gene bank for testing while Deepak prepared the report. We discussed the results with each other analyzed the complexities, finalized a conclusion and made the report.

VI. REFERENCES

[1] http://www.cs.yale.edu/homes/aspnes/pinewiki/SuffixArrays.html

[2] http://www.inf.fu-berlin.de/lehre/WS02/ALP3/material/sufficTree.pdf

[3] http://www.sciencedirect.com/science/article/pii/S0304397501001219 - references

[4] DIMACS educational module series, Module 09-2, Finding repeats with Strings, Rutgers University.

[5] Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree - by Jens Stoye and Dan Gusfield

[6] Space Efficient Linear Time construction of Suffix Arrays - by Pang Ko and Srinivas Aluru