Survey of Different Algorithms to Find Tandem Repeats

Sai Rahul Kasula (45617441), Sai Charan Kadari (51749229) and Saideep Korrapati (92348134)
Bio Informatics
University of Florida
Gainesville

***Abstract* — Tandem repeat is a sequence of two or more DNA base pairs that are repeated multiple times consecutively. They are generally associated with non-coding DNA. Tandem repeats are helpful to determine the inherited traits of a gene from its parent and in determining the ancestor relationship. They are proven to be biologically significant as they can help in the discovery of dynamic mutations for genetic diseases. In this paper, we discuss five different methods to find tandem repeats in each sequence.**

Keywords: Tandem repeats, Brute force, 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 ACGTACGTACGT, ACGT 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 computer science it essentially reduces to finding longest repeated substring.

In this project, we discuss five different algorithms namely brute force, Suffix trees, Suffix arrays, Suffix array with Manber and Myers algorithm and Dynamic programming to check if the given DNA sequence has a tandem repeat.

II. Brute Force Algorithm

In the brute force algorithm, we first generate all the possible substrings from the given sequence, resulting in $O(n^2)$ substrings. For each substring we check if it is the longest repeating substring in the input sequence. This results in an algorithm with time complexity $O(n^6)$. Such exponential time complexity doesn't scale well for large inputs. Below described methods have better time and space complexity compared to brute force algorithm.

III. Suffix Tree-Based Algorithm

Suffix Tree is very useful in numerous string processing and computational biology problems. A naive algorithm to construct a suffix tree takes $O(n^2)$ time. We discuss Ukkonen's Suffix Tree which takes $O(n^1)$ time.

A. High Level Description of Ukkonen's algorithm

Ukkonen's algorithm constricts a sequence of implicit suffix trees, the last of which is converted to a true suffix tree of the string S. The implicit suffix tree for any string S will have fewer leaves than the suffix tree for string S$ if and only if at least one of the suffixes of S is a prefix of another suffix. The terminal symbol $ was added to the end of S precisely to avoid this situation. However, if S ends with a character that appears nowhere else in S, then the implicit suffix tree of S will have a leaf for each suffix and will hence be a true suffix tree. The algorithm can be summarised as below:

- *Construct the tree T1*
- *For i from 1 to m-1 do*
- *begin {phase i+1}*
    - *For j from 1 to i+1*
        1. *begin {extension j}*
        2. *Find the end of the path from the root labelled S[j..i] in the current tree. If needed, 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 is optimized by using suffix links, active points and few other tricks.

B. 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 its 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.

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

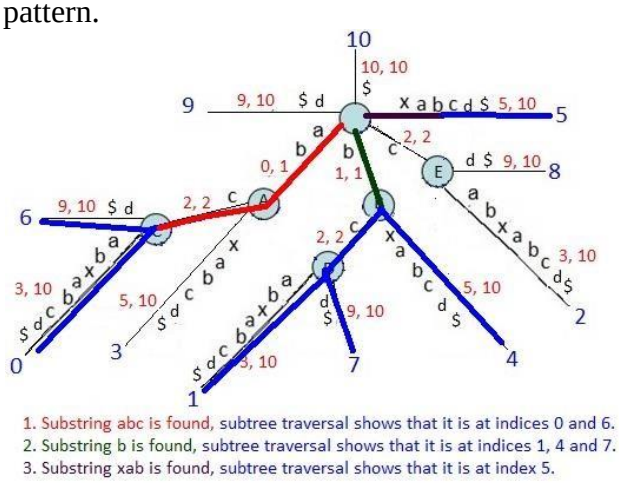


Figure 1

IV. Suffix Array Based Algorithm

Suffix arrays are data structures for representing texts in a format such that queries like "where does this pattern appear in the text" or "how many times does this pattern occur in the text" can be answered effectively. It works by storing all suffixes of a text. Instead of storing actual copies of the substring, we store pointer to its first character. This helps in reducing the space complexity from $O(n^2)\ to\ O(n)$.

In a suffix array all the suffixes are sorted in dictionary order with the indices in one column and the corresponding suffixes in another column.

The time complexity of a naive method to build suffix array is $\Omega(n^2 \log n)$ if we consider a $\Omega(n \log n)$ complexity of the algorithm used for sorting.

A. Tandem repeat search in suffix arrays:

Suppose we have a suffix arrays corresponding to an n-character text sorted in lexicographical order and we want to find the tandem repeats. Since the suffixes are sorted, we can do a binary search for repeats with $O(\log n)$ complexity. Hence the total cost would be $O(n \log n)$.

V. Suffix Array With Manber And Myers Sorting

Previously in suffix arrays, we used a sorting technique with complexity $\Omega(n \log n)$. This results in overall complexity of $\Omega(n^2 \log n)$.

Sorting suffixes differs from ordinary string sorting in that the elements to sort are overlapping strings of length linear in the input size n. This implies that a comparison-based algorithm which uses $\Omega(n \log n)$comparisons may require $\Omega(n^2 \log n)$ time.

Linear time for sorting can be achieved by building a suffix tree and obtaining the sorted order from the leaves. However, a suffix tree involves considerable overhead, particularly in space requirements, which commonly makes it too expensive to use for suffix sorting alone.

Manber and Myers suggested an algorithm that is in principal a radix sort, but where the number of passes is reduced to at most $\log n$ by taking advantage of the fact that each suffix is a prefix of another one: the order of the suffixes in the previous sorting pass is used as the keys for preceding suffixes in the next pass, each time doubling the number of considered symbols per suffix. This yields an algorithm which $O(n \log n)$ time in the worst case. A brief description of the algorithm is as follows:

1. Sort "I" using x_i as the key for i. Regard I as partitioned into as many groups as there are distinct symbols in X. Set k to 1.
2. Sort each group of size larger than one with a comparison-based algorithm, using the first position of the group containing x_{i+k} as the key for i when i + k.
3. Split groups between non-equal keys.

4. Combine sequences of unit-size groups so that these can be skipped over in subsequent passes.
5. Double k, and if there are any groups larger than one left, go to 2. Otherwise stop.

Finding the tandem repeats after this is similar to suffix arrays as described above.

VI. Dynamic Programming Based Algorithm

By making modification to the Smith-Waterman method for local alignment we can used it 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 of $O(n^2)$.

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. Align the input string with itself in the matrix.

2. Fill only the upper triangular matrix. The diagonal elements will be filled with 0.

Algorithm:

for (i=0 to n)
- Score [0, i] = 0;
- Score [i, i] = 0;

for (i=1to n)
  for (j=i+1 to n)
- **if** (charAt(i-1)==charAt(j-1))
  - Score [i, j] = Score [i-1, j-1] + 1;
- **else**
  Score [i, j] =0;

  }

}

To find tandem repeat, search for the highest score in the matrix and backtrack.

VII. Results

A. Input data

We collected nucleotide sequences from NCBI. Some of the sequences are AD000092.1, NM_001300741.2, AC139763.4 etc. We ran all the algorithms for the above sequences and noted various runtimes.

B. Implementation

All the codes are implemented in Java. The "parser.java" is used to read the sequence from a FASTA format file. The code can be found in this GitHub repository: https://github.com/rahulsai2341/TandemRepeats.

C. Execution results

We recorded three timings for each algorithm.

- Total execution time
- Time taken to build the data structure
- Time taken to get the tandem repeat from the data structure.

The results are summarized in the tables below for few sequences. The tables are sorted based on length of the input sequence from high to low. '-' below indicates that the program either crashed due to no memory or was running for very long time.

For the sequence AD000092.1: (*113,396* characters, 111KB)

	Brute Force	Dynamic programming	Suffix array	Suffix array – Manber	Suffix Tree
Execution time	_	_	224	115	144
Build time	_	_	201	100	144
Query time	_	_	23	15	0

For the sequence AC139763.4: (*14533* characters, 15KB)

	Brute Force	Dynamic programming	Suffix array	Suffix array – Manber	Suffix Tree
Execution time	_	1155	35	18	13
Build time	_	1073	29	14	12
Query time	_	81	5	4	0

For the sequence NM_001300741.2: (*3618* characters, 4KB)

	Brute Force	Dynamic programming	Suffix array	Suffix array – Manber	Suffix Tree
Execution time	93359	140	18	9	7
Build time	22320	121	14	6	5
Query time	71037	19	4	3	0

D. Observations

The execution times observed above are as per our understanding. With $O(n^6)$ time complexity and $O(n^2)$ space complexity, brute force method quickly becomes impractical for larger inputs.

Dynamic programming improves on time complexity to $O(n^2)$, but with space complexity of $O(n^2)$ it too becomes impractical for larger inputs.

Suffix array has time complexity of $\Omega(n^2 \log n)$ if we consider $\Omega(n \log n)$ algorithm used for sorting and space complexity of $O(n^1)$. Due to lower space complexity and lower query time, it can run for larger input sequences and faster compared to dynamic programming algorithm.

Suffix array with Manber and Myer's sorting algorithm improves on the sorting and thus reduces construction complexity. This results in total algorithm complexity of $O(n^2 \log n)$. This effect can be seen in execution times of these algorithms in the above tables.

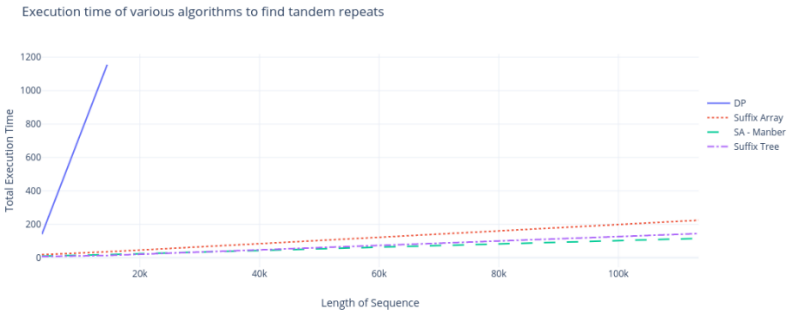
Suffix tree with Ukkonen's algorithm takes only $O(n^1)$ construction time and linear time search for tandem repeats. Space complexity is also linear but has more overhead compared to suffix arrays. Since query time for tandem repeats is lesser compared to suffix array, execution times better for smaller inputs. It seems to increase for larger inputs though.

The complexities are summarized in the below table:

	Brute Force	DP	Suffix Array	Suffix Array-Manber	Suffix Tree
Construction time complexity	$O(n^2)$	$O(n^2)$	$\Omega(n^2 \log n)$	$O(n^2 \log n)$	$O(n^1)$
Query time complexity	$O(n^6)$	$O(n^2)$	$\Omega(n \log n)$	$O(n \log n)$	$O(n^1)$
Total time complexity	$O(n^6)$	$O(n^2)$	$\Omega(n^2 \log n)$	$O(n^2 \log n)$	$O(n^1)$
Space complexity	$O(n^2)$	$O(n^2)$	$O(n^1)$	$O(n^1)$	$O(n^1)$

E. Graphs

The graphs are plotted between total execution time of the algorithms and length of the input sequence. Each colour represents a particular algorithm.



Zooming into the graph by eliminating dynamic programming:

VIII. Conclusion

Brute force, with its exponential time complexity and quadratic space complexity is not practical as input sequence length increases. Execution time of dynamic programming is better compared to brute force but is limited by its quadratic space complexity.

All of the suffix-based methods have improved space complexities.

Among them, Suffix Tree initially appears to be faster, but, Suffix array with Manbers and Myer's sorting algorithm is faster for larger inputs.

IX. Workload Distribution

Initially, all of us researched together for different algorithms available to find tandem repeats. Later, we divided the tasks among ourselves. Rahul was responsible for Brute Force algorithm, partly responsible for Suffix Array and Suffix Tree. Charan was responsible for gathering different datasets, partly responsible for Suffix array and Suffix Array with Manbers and Myer's algorithm. Saideep was responsible for summarizing all the results and plotting the graphs, partly responsible for Dynamic programming, Suffix Tree and Suffix

Array with Manbers and Myer's algorithm. Finally, we all contributed in drafting the project report and observations.

X. REFERENCES

1. *Notes on Suffix Sorting, N. Jesper Larrson, Lund University, Sweden*
2. *Algorithms, Fourth Edition, Robert Sedgewick and Kevin Wayne*
3. *http://www.cs.yale.edu/homes/aspnes/pinewiki/SuffixArrays.html*
4. *https://sandipanweb.wordpress.com/2017/05/10/suffix-tree-construction-and-the-longest-repeated-substring-problem-in-python/*
5. *Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree - by Jens Stoye and Dan Gusfield*