ABSTRACT

MST BASED AB INITIO ASSEMBLER OF EXPRESSED SEQUENCE TAGS

By Yuan Zhang

In the thesis we present a new algorithm for the assembly of ESTs based on a minimum spanning tree construction. By representing the EST input as a graph, quickly assigning edge weights with a new distance heuristic for detecting edge overlap, and calculating a minimum spanning tree, we find that we can use a quick transversal of this tree to accurately order the EST set, filter out extraneous ESTs, correct errors and infer the actual transcript sequence from which the set was generated. Implementing this into a software tool, we find that our tool gives us results better than CAP3, the leading EST assembly tool in the literature. In this paper we will explain the EST assembly problem, outline the assembly algorithm, and present a quantitative comparison against the results of CAP3.

MST BASED AB INITIO ASSEMBLER OF EXPRESSED SEQUENCE TAGS

A THESIS

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Computer Science
Department of Computer Science and System Analysis
by

Yuan Zhang
Miami University
Oxford, Ohio

2010

Advisor Dr. John Karro

Reader Dr. Mufit Ozden

Reader Dr. Chun Liang

List of Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to sincerely thank Dr. Karro for all his help and advices throughout the thesis process. Dr. Karro provided his kind help in each step of my thesis, from algorithm design, system optimization, coding, system evaluation to testing. Dr. Karro not only advises me on my research, but also encouraged me and set a very good model of a computer scientist for me. I would also like to thank Dr. Ozden and Dr. Liang for willing to be in my thesis committee and spending time on advising and reviewing my thesis. Thank Dr. Rao and Mr. Moler for all the help they have given to me on the thesis and my research.

1. Introduction

Expressed Sequence Tags, or ESTs, represent large numbers of short, randomly-selected, partial single-pass transcript sequences derived from one or more organisms. Using EST information, we have gained insight into the structure and function of transcribed genes. ESTs have also been used to enable gene discovery, aid gene structure identification, guide single nucleotide polymorphism (SNP) characterization, facilitate proteome analysis and detect alternative splicing[1,21].

There are four main stages when trying to discover genes from a set of ESTs: EST cleaning, clustering, assembling and annotation. Since ESTs are prone to sequencing errors, in the cleaning stage we need methods to reduce random noise by removing errors from EST data, returning high-quality sequences. Clustering algorithms are used to partition ESTs so that all the ESTs from one gene transcript are put into the same cluster and each cluster only includes information from one transcript[2]. Following this, we use an assembly algorithm to generate a tentative gene sequence from the cluster of ESTs (the subject of this research). Finally, the process of attaching biological information to a gene profile is called EST annotation.

ESTs represent essentially complete or partial segment of an mRNA transcript, the subset of all mRNA molecules expressed by an organism. A gene is a genomic sequence comprised of exons and introns. During the process of *transcription*, a pre-mRNA is transcribed using the gene as the template. Then introns are spliced out and exons are connected together to form a mature mRNA.

In this project, we will focus on EST assembly: the problem of reconstructing a transcript sequence from the cluster of EST fragments. Ideally, we can cluster all ESTs from a single transcript into the same cluster and derive a single consensus sequence to represent this transcript.

The classical approaches to the sequence assembly problem use the “overlap–layout–consensus” approach; that is, they use pair-wise sequence alignments to find a best fit, and build consensus sequences by merging reads that overlap[3]. In this project, we aim to find a new *Minimum Spanning Tree* (MST) based approach to assemble a single transcript. Specifically, we begin with an unsorted set of EST reads corresponding to one transcript and an MST for the set of EST reads. Then we use the MST to guide us through the ESTs and determine the sequential relationship among the reads. Based on the information, we assemble all the EST reads and make one or more consensus sequences.

We named our assembly tool EAST (EST Assembly from Spanning Trees). We tested EAST on simulated data. The type of the data includes Sanger reads[13] (700-1000 bases long), 454 reads[14,15] (~400 bases long) and Illumina reads[16] (50-100 bases long). The tests proved that EAST can assemble all of the three kinds of reads. And it is better than CAP3, one of the primary tools for the assembly of Sanger sequences in the literature[3]. EAST generates consensus sequences with better quality than CAP3, but using much less time for Sanger reads. EAST also generates consensus sequences with even much better quality than CAP3 for both 454 and Illumina reads.

2. Background

In this section we will discuss both the biology and the computational work predating our assembly tool. We begin with an overview of the biology underlying our project. We then introduce existing assembly tools against which we will compare EAST. Following this we will outline some of the underlying algorithms we will need to measure similarity and distance of strings.

2.1 Biological background

A gene is the basic unit of heredity in a living organism. We can view genes as an information storage system that holds the information needed to build and maintain cells and pass genetic traits to offspring. The information directs the construction of proteins, which are the essential parts of organisms and participate in virtually every process within a cell.

The natural process of protein production is divided into three parts: transcription, splicing and translation[12]. Figure 1 shows the three stages. In the transcription stage the information in the DNA is transcribed into a mRNA molecule, while in the splicing stage certain portions of that information are removed. Finally, in the translation stage, the MRNA is translated into a protein.

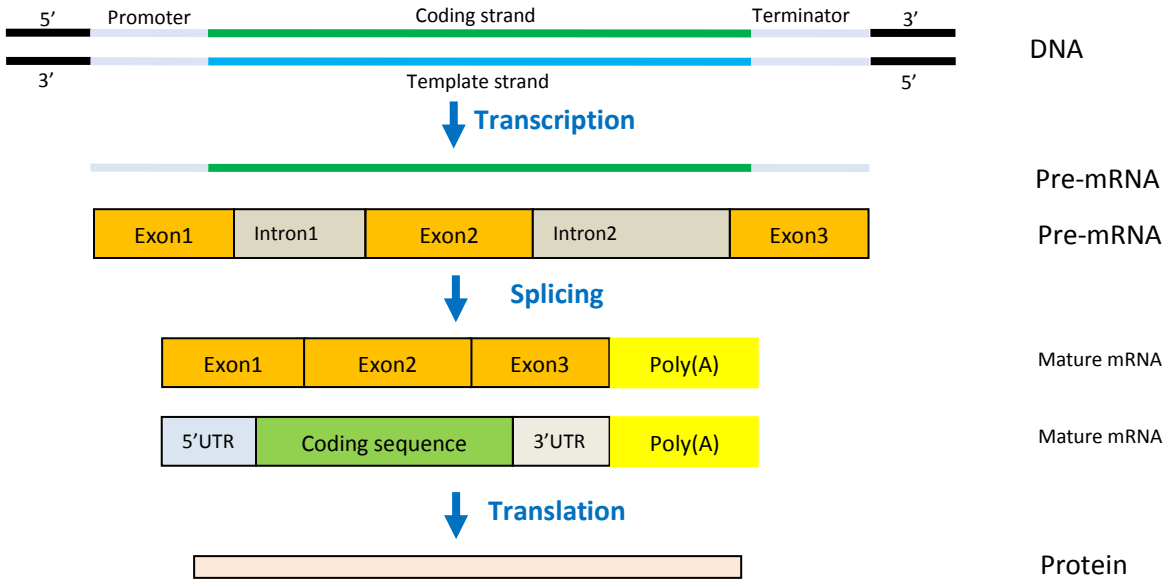


Figure 1: The natural process of protein production

In the process of transcription, the gene sequence is transcribed into an mRNA transcript. Specifically, the RNA polymerase binds to the promoter sites, usually on the 5'side of the gene to be transcribed. Working with protein transcription factors, the RNA polymerase opens the DNA double helix, proceeds to read one strand, assembles ribonucleotides into a strand of RNA and stops when it reaches the terminator sequence on the DNA. DNA has two strands: within the gene, the strand that is used for transcription is called the *template* strand, while the other strand is called *the coding strand*. When transcription is complete, the transcript, now called a pre-mRNA, is released from the polymerase and the polymerase is released from the DNA.

Pre-mRNA is transformed into mRNA (or a mature mRNA) in the process of splicing. Most genes are split into segments. The stretches of DNA which will be removed from the RNA transcript before protein translation are called *introns*. Those stretches of DNA that are kept in mRNAs are called *exons*. During splicing, the spliceosome removes introns and splices together the exons and the poly(A) tail, a stretch of adenine (A) nucleotides added to the end of the RNA transcript. This completes the mRNA molecule, which is now ready to be exported to the cytosol.

In some cases pre-mRNA can be spliced in different ways (depending on current conditions within the cell), allowing for the possibility of multiple transcripts – a phenomenon known as alternative splicing[12]. The combination of different exons from a gene form different mRNAs. Thus one gene can direct the generation of multiple mRNAs. Figure 2 shows an example of alternative splicing: four exons in a gene produce two mRNAs after alternative splicing, leading to two different protein isoforms.

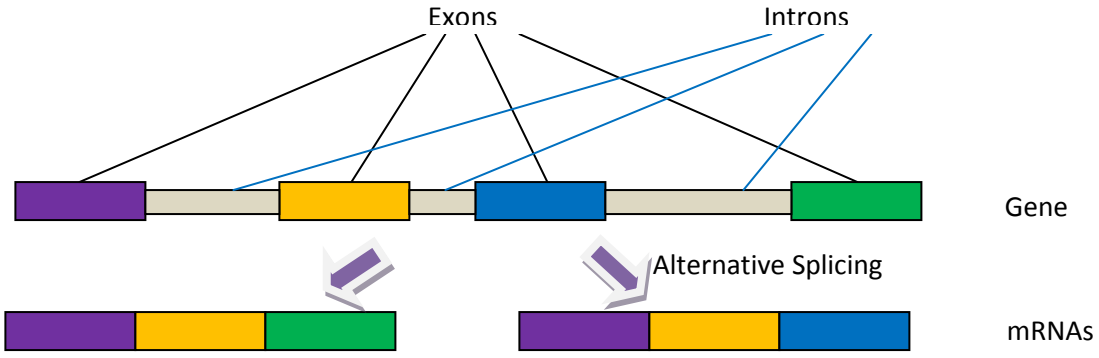


Figure 2: Alternative Splicing

ESTs are small pieces of DNA sequence that are generated by sequencing different parts of an expressed gene transcript. The process of generating ESTs is error prone, leading

to mistakes in the EST sequences. Generally there are three kinds of errors in ESTs: substitutions, insertions and deletions. A substitution error occurs when one base is replaced by another. For example, the sequence "ACTG" might be transformed to "AGTG" when a "G" is mistakenly substituted for the C in the generated EST. An insertion error occurs when one base is added into a sequence: the sequence "ATG" might be transformed to "ACTG" when the second base "C" is inserted into the generated EST. The deletion error occurs when one base is deleted from a sequence: the sequence is "ACTG" might be transformed to "ATG" when the base "C" is deleted in the generated EST.

There are three common kinds of sequencing methods to generate reads for assembler, producing different read lengths. Sanger sequencing was invented in 1975[13], while 454 sequencing was brought to commercial use by 454 Life Sciences in 2004[14,15]. The new sequencing methods generated reads much shorter than Sanger sequencing: initially about 100 bases, now 400 bases and expected to grow to 1000 bases. Since 2006, the Illumina sequencing is also available generating reads with the length of up to 150 bases[16].

2.2 String similarity and distance measures

To assemble ESTs, we need specific standards to measure the similarity of two strings, or alternatively define a distance between two ESTs that reflects their similarity. The similarity level of two ESTs quantifies the similarity in their sequence content helping us identify partial overlap. Conversely, the distance measure provides insight into the differences in the sequences of two ESTs. Simply put, string similarity and string distance are two aspects of the same characteristic, viewed from two different perspectives. The higher the similarity score, the smaller the distance score; the higher the distance score, the lower the similarity score.

String similarity measures are often based on alignment algorithms. An alignment of two strings is the paring of the characters. For example, given the strings:

```
GTGCCGGTTGTAATTCTATGCTAC
GGCCGGTCCTGTTTCTATGCTGTAC
```

We can align the strings to minimize letter-mismatches as follows:

```
GTGCCGGT--TGTAATTCTATGCT–AC
| ||||||  |||  ||||||||| ||
G-GCCGGTCCTGT--TTCTATGCTGTAC
```

We use similarity score to evaluate an alignment. The score is a function of the number of gaps (dashes), the number of matching characters and the number of mismatching characters in the alignment. Higher score indicates a better alignment. Given a specific scoring system, we can calculate the optimal alignment of two strings using one of two alignment algorithms: Needleman–Wunsch (a global alignment algorithm) and Smith-Waterman (a local alignment algorithm)[9,10]. Global alignment was designed to compute the best score and find the best alignment of two complete strings, while local alignment was designed to find the highest possible score using only a section of each string.

1) The Needleman–Wunsch algorithm[10]
   The Needleman–Wunsch algorithm was published in 1970 by Saul Needleman and Christian Wunsch. It performs a global alignment (that is, an alignment using the strings in their entirety) on two strings and is often used to align protein or gene sequences. Using a dynamic programming method, the algorithm searches for the highest scoring alignment of every character in a specific scoring system. Generally, the scoring system includes three parts: match point, mismatch point and gap penalty. For example, if we set match point to be 1, mismatch point to be -1 and gap penalty to be -2, we will get the highest score for the two strings s1="TATTGCCAGTTC" and s2="GATTCCCTTC". The global alignment corresponding to the scoring system looks like:

```
TATTGCCAGTTC
 |||  ||   |||
GATTCCC--TTC
```

   Score = -1 + 1 + 1 + 1 + (-1) + 1 + 1 + (-2) + (-2) + 1 + 1 + 1 = 2

2) The Smith-Waterman algorithm[9,10]
   The Smith-Waterman algorithm was published in 1981 by Temple Smith and Michael Waterman. It performs a local alignment (i.e. it finds the best matching substrings) and is often used for determining similar regions between two protein or gene sequences. Similar to Needleman–Wunsch, the goal of Smith-Waterman is still to obtain the highest score on the two strings. But while Needleman–Wunsch looks at the whole of each string, Smith-Waterman finds a segment from each to optimize the total score and ignores the rest. In the example from above the local alignment (marked in red) of s1 and s2 will look like:

```
TATTGCCAGTTC
 ||| ||
GATT-CCTTC
```

Score = 1 + 1 + 1 + (-1) + 1 + 1 = 4

3) Bounded alignment algorithm

   If we believe the two strings are similar, we may be able to use a bounded alignment algorithm to align them faster. If two strings *s1* and *s2* are same, the optimal alignment corresponds to the diagonal in the dynamic programming table. This is because all of the back pointers will be pointing diagonally. Therefore, if *s1* and *s2* are similar, we expect the alignment not to deviate much from the diagonal. So instead of looking at all the elements in the dynamic programming table, we only look at part of it which is in a specified distance from the same-line element on the diagonal.

Denoting the lengths of two strings with *n* and *m*, the runtime of the standard alignment algorithm will require *O(mn)* time. Let the width of the band in the bounded alignment algorithm to be *x*, the runtime of the bounded alignment algorithm would be *O(xn)*. While this is an acceptable runtime for computing the alignment of two sequences, our problem requires computing a number of alignments that is quadratic in the problem size. In the following example, the average length of each string is 500 bases. We have 100000 strings and we need align each pair of them. The total number of alignments will be 10^{10}, and will require $10^{10}*2.5*10^{5}=2.5*10^{15}$ operations. Assume that each alignment takes 1 second. The total time spent on the alignments will be 10^{10}seconds ≈ 317years. That is too long. Hence we need a faster measurement of similarity or distance.

To avoid slowness of the alignment algorithms we turn to the d^2 distance function[8], used to quickly estimate distances between strings. The d^2 function measures local similarity as a distance: two strings that have a sufficiently large common region will map to a small distance value, making it appropriate for use in identifying overlapping sequences.

Hide et al. first introduced the d^2 algorithm for High-Performance Sequence Comparison in 1994[8]. For d^2 calculations we define a window as any substring of an EST with a specified length. Given two strings *s1* and *s2*, $d^2(s1, s2)$ is computed by considering the frequency of all the words in each possible pairs of windows in *s1* and *s2*. Specifically, for any two windows, we compute the difference in the number of each possible word, taking the sum of the square of the differences. The d^2 algorithm looks at every possible window pairing between the two strings, and returns the minimum score of all these pairings.

Let $c_u(w)$ denote the number of times word w occurs in string u. We search for similarity between strings u and v by looking at the difference between $c_u(w)$ and $c_v(w)$ for different words w (with the length k). We use the following formula to get d^2 distance of two strings x and y:

$$d^2(x,y) = \underset{\substack{u \subseteq x \\ v \subseteq y \\ |u|=|v|=r}}{Min} \left(\sum_{|w|=k} (c_u(w) - c_v(w))^2 \right)$$

Here we look at the similarity between each pair of substrings *(u, v)* from x and y respectively (u is a substring of x, v is a substring of y, u and v have the same length r. r is called the length of the window), choosing the smallest pair as the d^2 distance of x and y. In the EAST algorithm, we set k to be 6.

A low d^2 score means that the two strings share a sub-segment of high similarity. However, the d^2 score cannot tell us if the two strings overlap. Take the following two strings as an example.

AAGGCTGCATTTGACCGTTCTATATGAGGGACCCGTCCTAGTGGACGTTAAACTAGCTATT
TTCGAACTTTTTGACCGTTCTATATGAGGGACCCGTCCTAGTGGACGTTAAACTCGAAGCC

These strings share a common substring (marked in red), and hence may have a lower d^2 score, but the lack of overlap at either end indicates they are not from overlapping ESTs. So we cannot determine overlap just by d^2 score. If we are to detect overlap exactly, we will need to employ an actual alignment algorithm, requiring a tradeoff between speed and precision.

2.3 Sequence assembly tools

Sequence assembly tools are used to assemble ESTs into the gene sequences. The most extensively used sequence clustering and assembly programs are CAP3[5], Phrap (*http://www.phrap.org/phrap.docs/phrap.html*) and TIGR Gene Indices[6, 7].

Phrap compares sequences by searching for pairs of reads with perfectly matching words (sequences of a fixed length) (*http://www.phrap.org/phrap.docs/phrap.html*). Using Swat, a program for rapid comparison of sequences, it computes *swat* score for the pairs that have matching words. Phrap then constructs contig layouts by aligning the reads in decreasing score order using a greedy algorithm and constructs a consensus sequence as a mosaic of the highest quality parts of the reads.

The basic idea of TIGR Gene Indices is to construct gene indices by first clustering, then assembling ESTs for the targeted species[6,7]. Specifically, sequences are compared pair-wise and grouped into clusters based on sequence similarity. Each cluster is assembled and the consensus sequence is generated using CAP3.

The CAP3 assembly algorithm performs three major steps[5]. First, it removes poor ends of EST reads and false overlaps by computing overlaps between reads. Then, the algorithm joins reads together to form contigs. Finally, the algorithm constructs a consensus sequence from the contigs. Base quality values (a quantification of our trust in the accuracy of base content) can be used in the computation of overlaps and construction of multiple sequence alignments.

Liang et al. showed that CAP3 outperformed TIGR Gene Indices and Phrap in the quality of its assembly solution[1]. To evaluate the performance of the three programs, the paper used a 600-base segment as the reference sequence to generate a set of simulated data. From the reference sequence they generated a set of model EST data each having the length of 450 to 550 bases, errors rates ranging from 1% to 8% and sequence coverage ranging from 5- to 50-fold. The best consensus sequence produced by each program was compared with the original sequence and its fidelity was assessed using A-score:

A-score = (2×sequence length) - (15×no. of insertions) -
(15×no. of deletions) - (5×no. of substitutions)

In this situation, a perfect assembly would have an A-score of 1200 (twice the actual length). Based on A-score measure, the *Liang et al.* paper concluded that CAP3 consistently worked better.

Liang et al. also used real ESTs from known genes and compared the consensus sequences from each assembly program[1]. One gene used is the *cytochrome c oxidase subunit II* gene of the rat mitochondrial genome. The evaluation is based on two standards: that the assembly tool accurately reproduce the gene, and that the tool uses all the ESTs to produce a single consensus. The paper showed that CAP3 was able to use all the data to produce a single consensus same as the gene. TIGR assembler used some of the lower quality sequences to produce a second consensus different from the gene. Phrap assembled all the ESTs into a single consensus which contained some insertion and substitution errors.

2.4 Clustering tool

Clustering tools such as WCD and PEACE are essentially “pre-assembly” tools used to cluster before assembly[11,22]. Our tools is designed specifically to be used with Peace, a clustering tool developed by Rao *et al*.[11], which is based on the use of Minimum Spanning Trees to define clusters.

PEACE works through the representation of ESTs and their relationships in a graph[11]. A vertex in the graph represents an EST, and the weight of an edge is the d^2 distance of two ESTs. A low edge weight indicates that the adjacent ESTs have a large similar portion. By calculating an MST of this graph and removing all edge weights exceeding some threshold value, PEACE can then infer clusters based on the resulting connected components. These clusters, and the corresponding MSTs, become the input to our assembly tool, EAST.

3. Algorithm Design

EAST is an *ab initio* assembler. That means it assembles ESTs together to form the full (unknown) transcript sequence without any knowledge of the genome. The assembly algorithm underlying EAST compares each EST with a selected subset of the other ESTs to predict their relative position on the transcript. Following this it identifies the *left end EST* – the one that can be mapped to the left-most position on the transcript. Starting from the identified left end, EAST combines these ESTs from left to right, forming one or more consensus sequences that represent the predicted transcripts.

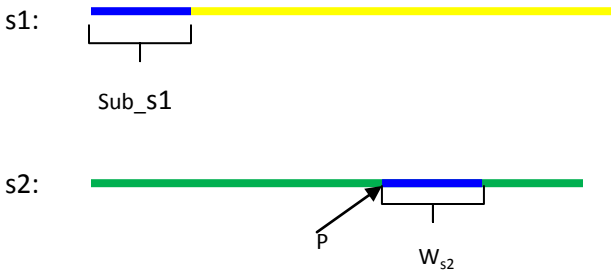
In the following sections we describe the assembly algorithm under the assumption that all ESTs are oriented in the same direction, and that we have an EST set from one transcript. The first is a correct assumption, as the pre-assembler, PEACE, can identify relative direction, thus allowing us to reorient those ESTs that are "backwards". Generalizing the algorithm to allow for multiple clusters requires only minor modifications, which are described in the Section 3.6.

3.1 Overlap Distance Calculation

We need a method to estimate the distance of two ESTs in terms of their physical *end-segment* overlap in the assembly, a value which will serve as the basis for our algorithm. While the d^2 distance function will indicate that two sequences share some common area, we need more specific information: whether they overlap at their ends. Overlap distance serves this purpose; the larger the overlap distance, the smaller the possibility that the two strings actually overlap at the ends. For two overlapping ESTs, we use overlap length to describe the length of overlapping portions. The Overlap Distance Calculation module is designed to compute overlap distance and overlap length.

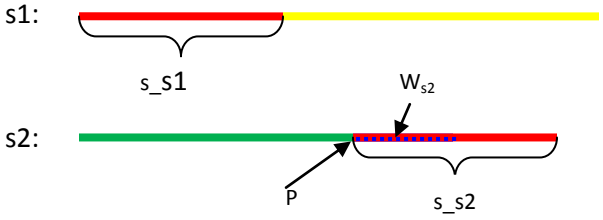
The input to the module is two strings (or two ESTs): s1 and s2. The output of the module is an overlap distance and an overlap length. The detailed description of the algorithm is as follows.

1) Calculate d^2 distance of (*sub_s1, s2*), where sub_s1 is the first window on string s1. Find the substring w_{s2} in s2 that minimizes the d^2 distance to *sub_s1* (in other words, w_{s2} is the substring of s2 that results in the d^2 score), and let *P* be the coordinate of the left-end of w_{s2}.



Let $D = d^2(sub\_s1, s2) = \sum_{|w|=6}(c_{sub\_s1}(w) - c_{Ws2}(w))^2$, summing over every possible word *w*. Here $c_{sub\_s1}(w)$ *and* $c_{s2}(w)$ denote the frequency of the word *w* in the window *sub_s1* and w_{s2}. If there are multiple w_{s2} having the same value D we will record the multiple *P into a set* S_p.

2) For each $P \in S_p$, we calculate the value *tDis* for *P* as follows, and assign the smallest such value to Dis_{left}.
   i. Take the substring *s_s2* of *s2* which starts at *P* and extends to the end of *s2.*
   ii. Take the substring *s_s1* of *s1* which starts from left end of *s1* and has the same length as *s_s2*.
   In the following figure, from *s2* we take the substring *s_s2* of length 80 starting at *P=107* (which was defined by the placement of w_{s2}). And we take the substring *s_s1* from *s1* which starts from the left end of s1 and also has the length of 80.



   iii. Compute the optimal alignment score of *s_s1* and *s_s2* (as found by the Needleman-Wunsch alignment algorithm), and assign the score to *a*. We use the simple scoring matrix with match=1, mismatch=-1 and gap penalty=-2. In our case, the user can specify to use standard Needleman-Wunsch or its bounded version to speed up.
   iv. Calculate the distance *tDis* for (*s_s2, s_s1*) with the formula:

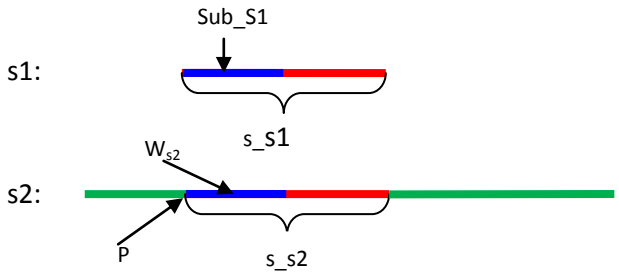
$$tDis = 1 - a/b$$

where *"a"* is the alignment score from the last step, and *"b"* is the length of string *s_s1* used to normalize the score (noting that *s_s2* and *s_s1* have the same length).

v. If *tDis* is bigger than some specified threshold *T*, we assume they do not overlap and set $tDis = \infty$.

3) Repeat steps 1-3 for the opposite ends of *s1* and *s2* to calculate Dis_{right}.
4) We conclude that s1 is included in s2 if:
   - ✓ The overlap length of *s1* and *s2* is equal to the length of s1;
   - ✓ The length of *s1* is less than *s2*;
   - ✓ Either Dis_{left} or Dis_{right} is less than a threshold T'.

   The following figure illustrates how we determine that s1 is included in s2. The overlap part between s1 and s2 is s1. That is, |s1| = |s_s1| = |s_s2| (satisfying the above first and second condition), $Dis_{left} = Dis_{right} = 0$(satisfying the third condition).



5) If *s1* neither includes nor is included in *s2*, we will compare Dis_{left} and Dis_{right}, returning the smaller one as the overlap distance of *s1* and *s2*. If *s2* is on the left of *s1* (that is, they overlap at the left-end of *s1* and the right end of *s2*), record this information by making the returned value negative. (Thus a positive value will indicate that *s2* is on the right of *s1*, as that is the only other possibility.)
   If *s1* includes or is included in *s2*, we will set their overlap distance to be $-\infty$.

Let *w* be the d^2 window size, *n* be the size of s2, and *l* be the average overlap length. The asymptotic bound of runtime of calculating overlap distance is $O(2*w*n + l^2)$, or $O(n+l^2)$, as *w* is generally a fixed value.

3.2 Main Process

Here we describe the main logic of the overall system. The algorithm first reads from the two input files: the EST file (containing the ESTs to be assembled) and MST file (containing the minimum spanning tree generated by PEACE). It scans the MST, collects specific information on their potential for overlap, and stores them in a set of *six-tuples* that describes the possible relationship of each node to its neighbors in the tree, and identifies the left ends. Following that, it reconstructs the consensus sequences, starting from the identified left ends. Figure 3 illustrates the procedure.

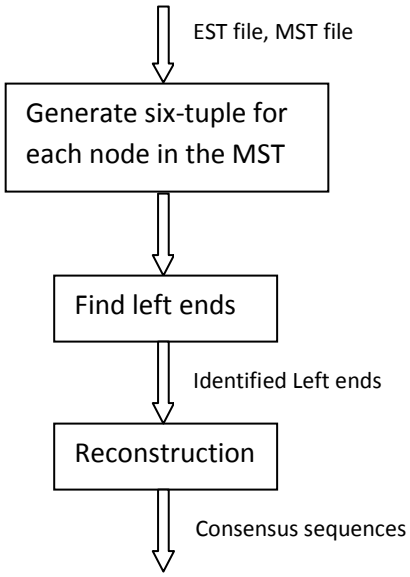


Figure 3: Main logic of the system

Below is the detailed algorithm describing how we calculate the six-tuples and determine which ESTs correspond to the left ends of a consensus sequence.

1) Each node represents an EST. Starting from an arbitrarily designated root node, we transverse the MST and calculate the overlap distance for every pair of adjacent nodes (thus calculating one alignment for each of the *n-1* edges).
   If the EST of one node is included in the EST of another, we remove the included node from the MST.
   In the example below, *N_P* is the parent node of *N*. *N* has three children *N_C1, N_C2, N_C3*. If *N's EST* is included within *N_P*'s EST, we will remove *N* from the MST and save it for later processing, promoting all of the children of *N* to children of *N_P*. The following figure compares the two MSTs before and after removing *N*.

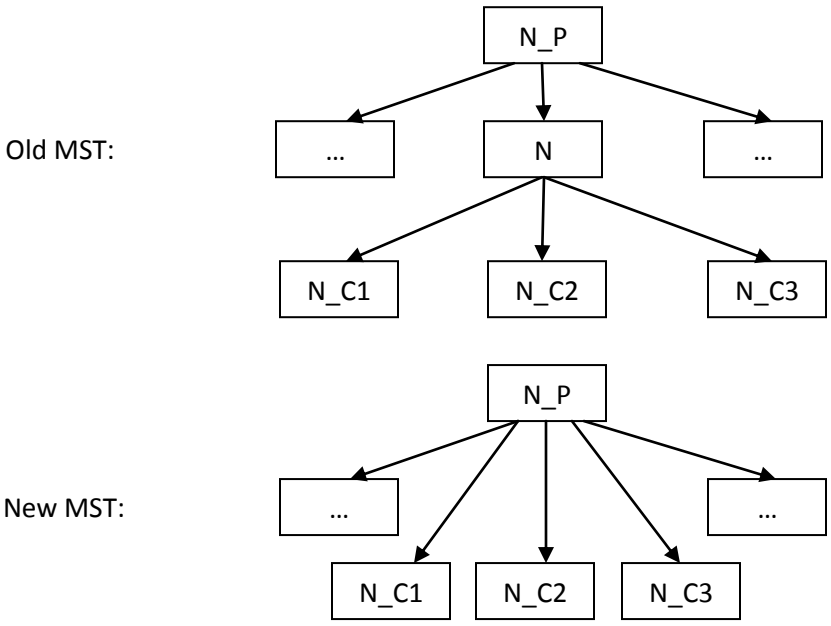


Figure 4: Comparison between old MST and new MST

2) For each EST *N*, we look for its neighbor with the *smallest positive* overlap distance (i.e. the EST closest to *N* on the right), and its neighbor with the *largest negative* overlap distance (i.e. the EST closest to *N* on the left). We use these values to create a six-tuple N_T = (u, l_L, d_L, v, l_R, d_R). Here "*u*" is the index of the node of the closest EST on the left of *N*, "l_L" is the overlap length of *N* and *u*, "d_L" is the overlap distance of them. "*v*" is the index of the node of the closest EST on the right of *N*, "l_R" is the overlap length of the two ESTs, and "d_R" is their overlap distance.

   For example, EST *A* has neighbors with corresponding ESTs *P*, *C1* and *C2*. The overlap distances of *(A, P), (A, C1), (A, C2)* are 3, -1, and 5 respectively; the corresponding overlap lengths are 15, 17, and 12. The six-tuple of *A* will be*(C1, -17, -1, P, 15, 3)*.

   If *u* exists (i.e. there is a EST to *A*'s left), l_L and d_L are less than or equal to zero; otherwise l_L is zero and d_L is ∞. If *v* exists, l_R and d_R are greater than or equal to zero; otherwise l_R is zero and d_R is $-\infty$.
3) Read every node in the MST and create the six-tuple for its corresponding EST.

4) We say one EST is a presumptive left end if there does not exist "*u*" in the six-tuple for this EST. For example, if the six-tuple for EST *w* is *(null, 0, -∞, 70, 15, 3)* we will see *w* as a presumptive left end. Here the *"null"* value tells us that there is not any EST located to the left of *w.* We call it "presumptive" because we have not verified whether it is a real left end or not – it remains possible that it overlaps some EST to its left that is not represented by an adjacent node in the MST.

   Starting with the set of presumptive left ends S_{l0} we check all the elements in the set and remove false left ends with the following three steps.

   I. For every presumptive left end *E* in S_{l0}, examine all nodes that can be reached from *E* in the MST by a path of at most three edges; we calculate a new six-tuple for *E* by updating its six-tuple with respect to ESTs for each of these nodes. Note that the old six-tuple has *null* as its first element (indicating that no adjacent node in the tree has been found to the left)*;* hence we are effectively expanding our search for a left overlap past the adjacent nodes in the tree*.*

      Having matched each presumptive end against its local neighborhood, we eliminate those ends for which a left-match was found, leaving us with a new set of presumptive left ends $S_{l1} \subseteq S_{l0}$. We cannot yet label them as actual left ends, as we cannot be sure that they do not have a match at a greater distance in the tree.

   II. For every presumptive left end *E* in S_{l1}, we expand our search radius out one more edge, look at all nodes four edges away. If a left-match was found, we will reset the six-tuple and determine that *E* is not a left end. If we do not find it in the 4^{th} level, we extend the search to the 5^{th} level, and continue up to some limiting value (we have found 15 to be effective, but users can set this up the diameter of the tree — possibly improving result quality, but at the cost of speed). Upon completion of the search, we are left with a new set of presumptive left ends $S_{l2} \subseteq S_{l1}$.

   III. A presumptive left end *E* in S_{l2} is a false left end if it appears as the fourth element in any other six-tuples, implying that this node overlaps *E* on the left. For example: *EST i* has the six-tuple: *(-1, 0, -∞, 4, 8, 9)*, but *EST j* has *(7, 5, 3, i, 7, 6)*, this means the overlap distance of *(j,i)* is 6. So node *i* is not a left end because node *j* is to its left.

      Remove all the false left ends in S_{l2}, leaving us with our set of (true) left ends $S_l \subseteq S_{l2}$.

5) Starting from each left end in S_l, calculate the starting position of the ESTs and reconstruct the transcript sequence.

   In Section 3.3 we explain how we use these six-tuples to calculate starting positions for each EST. In Section 3.4 we explain how we perform reconstruction.

Let m be the number of nodes in the MST, n be the average length of ESTs, l be the average overlap length, and t be the average degree of each node. We spend $O(m)$ time traversing the tree, $O(n+l^2)$ time at each edge to compute the overlap distance. We record the overlap distance between each pair of adjacent nodes, so the time to construct the six tuple for each node is $O(n+l^2+t)$. We add t to the runtime because we have to make t comparison to find the smallest distance as the elements in six tuple. The total runtime of constructing six-tuples for all the nodes is $O(m(n+l^2+t))$ time.

The runtime required to remove false left depends on the number of presumptive left ends and the depth limit placed on the MST search. Let k be the number of elements in S_{l0} (the initial set of presumptive left ends) and d be the search depth (*i.e.,* for any left end E_i in S_{l0}, we will search all nodes within a path length of d edges.) As before, let m be the number of nodes in the MST, n be the average length of ESTs, l be the average overlap length and t be the average degree of each node. In the worst case, when we have to search all d levels, we search t^d ($t^d \leq m$) nodes for each of the k ends. We spend $O(t^d(n+l^2))$ time at each presumptive left end to construct the new six tuple. The total run time of removing false left ends would be $O(kt^d(n+l^2))$.

3.3 Starting Position Calculation

Based on the six-tuples calculated previously, we construct a new directed minimum spanning tree connecting our EST-nodes. Each node in the tree corresponds to an EST and the weight of an edge represents the overlap distance of the ESTs represented by the two adjacent nodes. An edge from node A to B means B is the closest neighbor of A on the right of B. Once constructed, we can transverse the MST and calculate starting positions for each node. The algorithm works as follows:

1) Extract nodes and overlap distance information from all the six-tuples and construct a directed graph.

   If EST w has the six-tuple (u, l_L, d_L, v, l_R, d_R), we will get $(u, w, -d_L)$ and (w, v, d_R) from it. $(u, w, -d_L)$ indicates that the distance from EST u to w is $-d_L$ (recall we made d_L negative to record the fact that it is on the left). Similarly, (w, v, d_R) represents that the distance from w to v is d_R. We take u, v and w as nodes, take $-d_L$ and d_R as edge weights between (u,w) and (w,v) respectively, and generate a

directed, weighted graph *G* whose topology is defined by our six-tuples reflecting EST proximity.

2) Starting from the root of the tree (which must be a left end), use Prim's algorithm[19] to construct a minimum spanning tree based on the directed graph *G*. Then calculate the starting positions for all the nodes in the tree (explained in step 3). If two edges have the same length, Prim's algorithm will make arbitrary choices between them.
3) Starting from the root *r*, we calculate the starting positions of all nodes in the tree in a breadth-first manner – calculating a node's position before that of its children. Note the tree is directed, with each edge starting from a parent node N_s and ending at a child node N_c, where N_c overlaps the right end of N_s.
   Let *w* be any node in the tree which is not *r*, and let *w_p* be the parent node of *w*. The starting position of *w* is:

   $$s1 + l1 - l2$$

   where *s1* is the starting position of *w_p*, *l1* is the length of the EST which *w* represents, and *l2* is the overlap length from *w_p* to *w* (information stored in *w*'s six-tuple).

   We set starting position of *r* to be zero.

3.4 Sequence reconstruction

Now we have a set of left ends – ESTs where we have not found any other ESTs overlapping on their left. When the error rate is low, there are two possible relationships between two left ends L_1 and L_2: $L_1$and L_2 have an inclusion relationship (one includes the other), or L_1 and L_2 do not overlap. When the error rate is high (generally beyond 0.06, refer to the experiments in *Section 4* for details) or when the search depth for left ends check is not big enough, it is possible that EAST erroneously identifies a left end.

We want to put those left ends with an inclusion relationship together to avoid performing reconstruction from each of them. The method we use is to identify and group those nodes with an inclusion relationship into one cluster, and perform construction for the cluster.

Thus given a set of left ends, we will check their relationship and group them accordingly. In an ideal situation (a low error rate and a sufficient search depth, nodes in different clusters will not overlap. For example, we have the set of left ends {L_1, L_2, L_3, L_4,

L_5} whose relationship is illustrated in the figure. In this example, there are two clusters of left ends {L_1, L_3, L_5} and {L_2, L_4}.

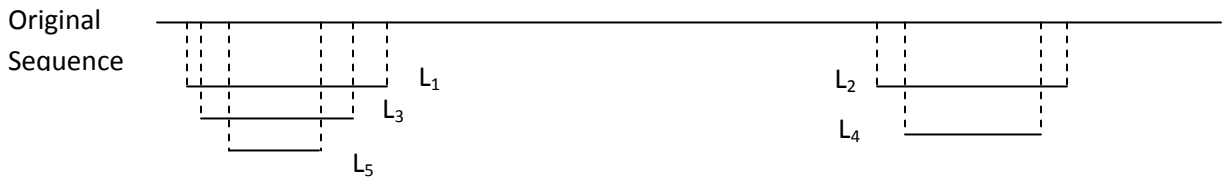


Figure 5: Relationship of left ends

We will reconstruct each cluster from the set of left ends {L_1, …, L_n}. For each left end L_i in a cluster, set its starting position to be zero. Starting from L_i construct a Minimum Spanning Tree from the set of six tuples, and then calculate the starting position for all the nodes in the tree using the algorithm in *Section 3.3*. For each node N_i with a coordinate P_i, assign, as an initial estimate, the value P_i+1 as the start coordinate of those nodes which are included by N_i. Recall that in step 1 of *Section 3.2* we removed all ESTs which were included by another EST from the input MST. Now we add them back for use in the reconstruction

We now sort these nodes in ascending order according to their starting positions. For example, if nodes N_1, N_2, N_3, N_4, N_5 have the following starting positions respectively: 18, 240, 58, 180 and 74, then they will be sorted at this order: N_1, N_3, N_5, N_4, N_2.

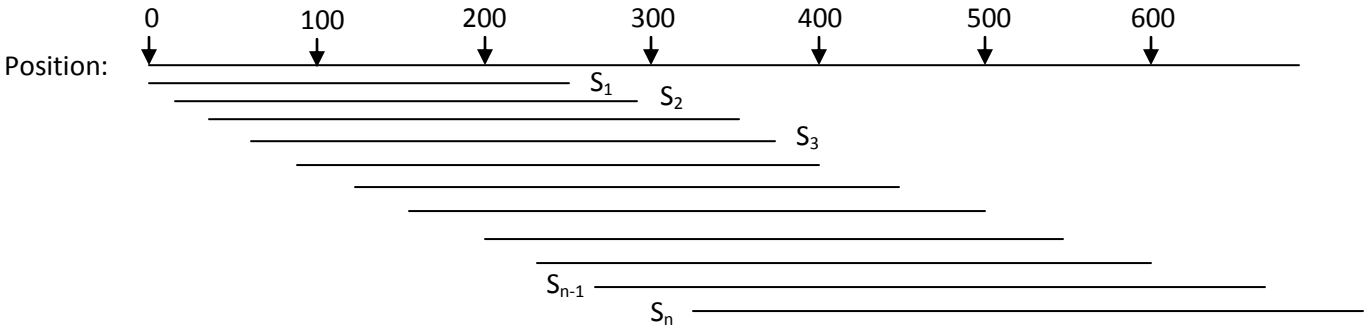


Figure 6: Illustration of reconstruction

Given a set of sorted nodes {S_1, S_2, S_3, …, S_n}, get a temporary consensus sequence S_2' from S_1 and S_2. Take a fixed-length string S_2''(starts from some position in S_2' and ends at the last character of S_2') from S_2' and generate another temporary consensus

sequence S_3' from S_2'' and S_3, then S_4' from S_3'' and S_4, so on and so forth until get the final consensus sequence from S_{n-1}'' and S_n.

We record all the characters at each position *p* to aid us decide the temporary consensus sequence. For example, We start from S_1="ATCTGGATCG" with the length of 10, then the characters at *p=0* are {A}, *p=1* are {T}, …, *p=9* are {G}. The current temporary consensus sequence is equal to S_1.

Next, we make a local alignment on S_1 and S_2 and get alignment strings sS_1 and sS_2. Find the corresponding coordinate of sS_1 on S_1 (the current consensus sequence) *cor_ss1*, record characters in sS_2 into their corresponding position (the position of the first base in sS_2 is *cor_ss1*). Record those characters in S_2 not appearing in sS_2 into their corresponding positions. For example, S_2="AGGATCGATTC". sS_1="GGATCG", sS_2= GGATCG", *cor_ss1*=4. After records all the characters of S_2, we will have: *p=0* {A}; *p=1* {T}; *p=2* {C}; *p=3* {T,A}; *p=4* {G,G}; *p=5* {G,G}; *p=6* {A,A}; *p=7* {T,T}; *p=8* {C,C}; *p=9* {G,G}; *p=10* {A}; *p=11* {T}; *p=12* {T}; *p=13* {C}. We construct a temporary consensus sequence S_2' from the set of recorded bases. We take the character that occurring the most times as the base at this position. For example, the base is "C" at *p=8.* If different characters exist in one position and have the same number of occurrence, we will take one of them. So S_2'="ATCTGGATCGATTC". Give S_2', we can use the same process to generate temporary consensus sequence S_3' from S_2' and S_3.

Note that if there is an insertion error in S_2, it is possible that the inserted character will appear in S_2'. But because other following ESTs (S_3, S_4, …)will likely not have the same insertion errors, the character occurring the most times at this position will be a gap– leading to a gap at the position in the resulting temporary consensus sequences that will be deleted. Hence our algorithm is able to correct an insertion error. Similarly, if there is a deletion error in S_2, possibly S_2' will miss the base at this position. But because other ESTs will likely not have the same deletion errors, the deleted character would still be included in the resulting temporary consensus sequences. In this way our algorithm is able to correct a deletion error. Likewise, our algorithm can correct a substitution error because the most occurrences at a position should be the correct character although there is some substitution errors obscure it.

Let *n* be the average length of ESTs, m be the number of ESTs and *j* be the fixed length, then the asymptotic runtime of reconstructing from a set of sorted nodes is $O(m\log(m) + mjn) = O(mjn)$. ($m\log(m)$ is the time spent on calculating starting positions, *mjn* is the time spent on assembling consensus sequences.)

3.5 Runtime Analysis

We have introduced the details of EAST. Now we will turn to analyze the runtime of EAST as a whole.

We define the following variables:

w: the d^2 window size.

m: the number of nodes in the MST.

n: the average length of EST.

l: the average overlap length.

t: the average degree of each node.

k: the number of presumptive left ends.

d: the search depth for checking left ends.

j: the fixed length for reconstructing consensus sequences.

The runtime used to get overlap distance is $O(n+l^2)$, so the runtime on constructing six-tuples for all the ESTs will be $O(m(n+l^2+t))$. Then EAST takes $O(kt^d(n+l^2))$time to find actual left ends. Given that $td \leq m$(that is, in the worst case we will search all the nodes in the tree), the total upper bound of runtime on generating six-tuples and finding left ends is $O(km*(n+l^2+t))$. Adding that time with reconstruction time, the total runtime of EAST is bounded by $O(km*(n+l^2+t) + mjn)$.

3.6 Multiple Clusters

As previously mentioned, PEACE is a clustering tool which aims to put ESTs from one transcript into one cluster. In the above description, we assumed we were working with only one cluster. In reality, our input will contain ESTs from multiple clusters, which PEACE represents as a single MST and differentiates cluster with edges exceeding some threshold T_c. That is, A and B are not in the same cluster if there is an edge on the path between node A and B with weight exceeding the threshold T_c.

Thus the actual inputs to EAST are one EST file and one MST file which contain ESTs from multiple clusters. We need to separate these clusters, perform the assembly for each cluster, and finally return the set of consensus sequences constructed from the clusters.

The algorithm for multiple clusters has a slight difference from the one for single cluster in the following two procedures.

- In the procedure of constructing six-tuple for each node, we compare the edge weight against T_c. An edge from node *A* to *B* with bigger weight than T_c tells us

this is the boundary of the current cluster (*A* is a leaf in the cluster, and *B* does not belong to the cluster), indicating we do not need search past *A*.

- In the procedure of finding actual left ends from S_{l0} (the set of presumptive left ends), we will stop searching along any edge with weight exceeding T_c.

The algorithm for multiple clusters adds a function in the reconstruction section used to handle multiple clusters.

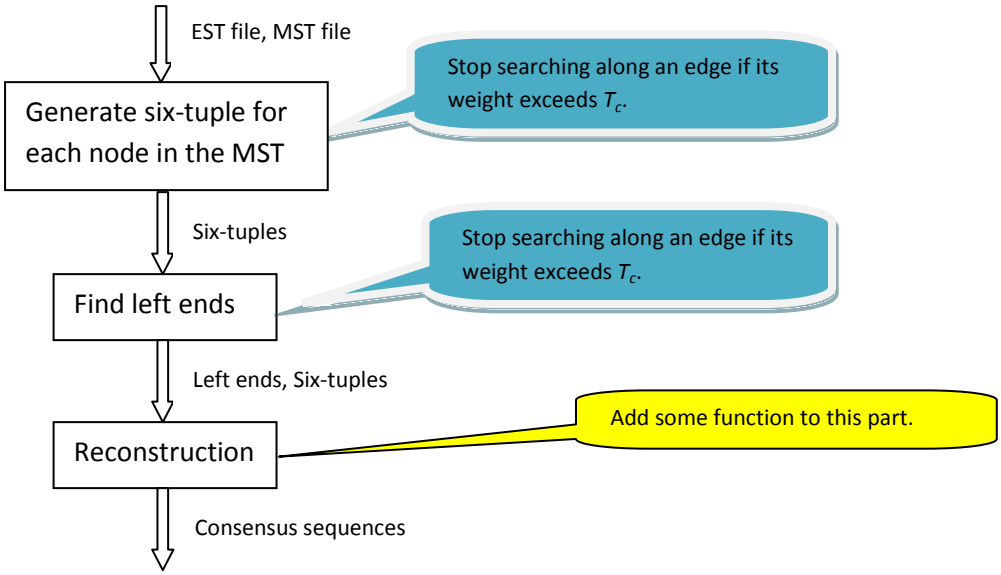


Figure 7: The procedure for handling multiple clusters

The method for handling multiple clusters in the reconstruction is as follows.

1) Separate the clusters from the input MST according to the PEACE threshold T_c;
2) Associate left ends with clusters. A left end is associated with a cluster if the left end belongs to the cluster;
3) For each cluster, use the "reconstruction" algorithm in section 3.4 to generate consensus sequences;
4) Return all the consensus sequences.

As before, let *m* be the number nodes in the MST. Runtime on step 1 and 2 is *O(m)* since we have to scan the MST. Let *c* be the number of clusters and *R* be the runtime of performing reconstruction on one cluster, the runtime on step 3 would be *O(cR)*. The total time on handling multiple clusters is bounded by *O(m+cR).*

4. System Evaluation and Results

Over the past three decades, Sanger sequencing has been the primary method used in DNA sequencing. However, in the last few years, new high-throughput sequencing technologies, such as 454 and Illumina, have emerged with the ability to generate large numbers of sequences at a very low cost, but producing much shorter segments[25]. With Sanger sequencing still an important method, but short-read sequencing quickly overtaking it in popularity, it is necessary to test the effectiveness of EAST on the product of each of these technologies.

East was tested on simulated data. We simulated Sanger reads of 700-1000 bases long with different error rates and coverage depth using the ESTSim, a program for generating simulated Sanger EST sequences based on a model developed by Hazelhurst et al.[20]. We also tested EAST on simulated 454 and Illumina reads using another simulator, MetaSim, which has developed its own model specifically for those technologies[23].

4.1 Test pipeline

The purpose of the pipeline is to generate simulated data, input the data to the tools and analyze the results.

In each run the pipeline randomly picks a specified number of transcripts and employs the simulator (either ESTSim or MetaSim) to generate a set of ESTs. The pipeline then uses these ESTs as input for both PEACE+EAST and CAP3. PEACE and EAST work together as an assembly tool: PEACE clusters the data, produces an MST corresponding to the input data, and passes each cluster and the MST to EAST. EAST will assemble the ESTs in each cluster and return the consensus sequences. CAP3 performs both clustering and assembly. It accepts a set of ESTs and generates the corresponding consensus sequences. After obtaining results from EAST and CAP3, the test pipeline analyzes them and prints the comparison to an output file. The procedure is shown in **Error! Reference source not found.**.

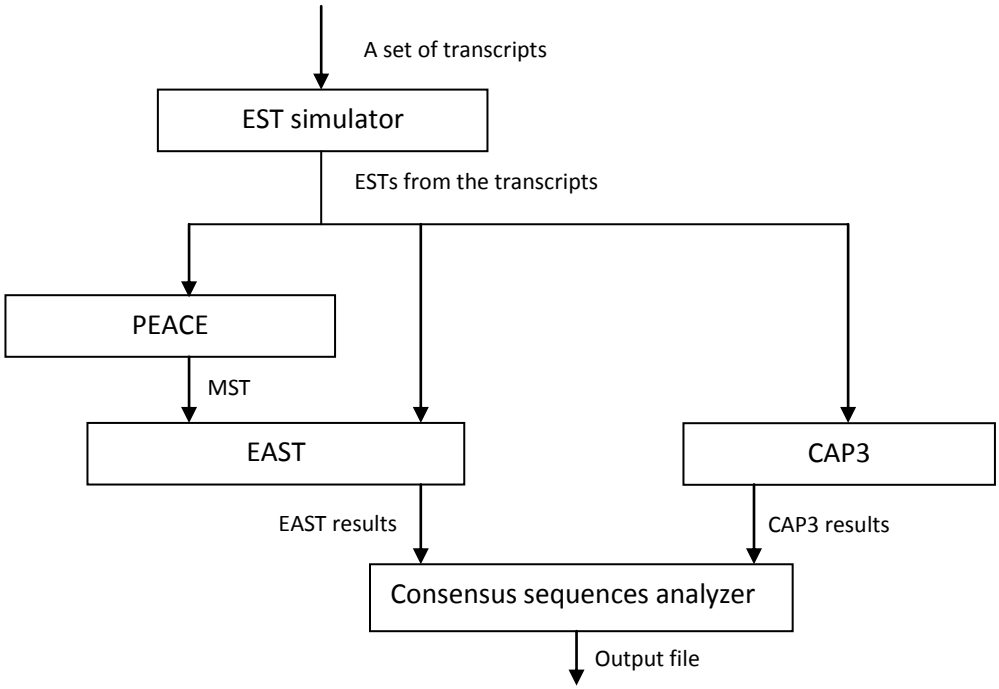


Figure 8: Pipeline for simulated data

4.2 Evaluation standards

We use the following criteria to compare and evaluate the assemblers:

1) **Normalized number of consensus sequences(*NN*)**: the quotient of the total number of consensus sequences and the number of selected transcripts. The optimal value of a normalized score is equal to 1, with lower quality indicated by values further from 1.

   NN = the number of consensus sequences/the number of transcripts

   (Perfect value of *NN* = 1)

2) **Mean of Normalized A-Score(*NA*):** For each transcript, we find the consensus sequence with the largest A-score from all its corresponding consensus sequences. A normalized A-score is the quotient of the A-score of this consensus sequence and the perfect A-score (defined in Section 2.3 as a value twice the length of the original transcript). The A-score of a consensus sequences is less than or equal to the perfect A-score, so the closer to 1 the normalized A-score, the better the consensus sequence. We take the mean value of all the normalized A-scores as *NA*.

$$NA = \frac{\sum_{\text{transcript}} (\text{A - score}/2 * \text{transcript length})}{\text{number transcripts}}$$

3) **Time(*T*):** the runtime spent on each run by a tool.

T = time spent on each run

4) Duplication differentiation:

We also test the capability of EAST and CAP3 to differentiate transcripts derived from duplicated genes. In a given test, we randomly choose 1 transcript, copy it, and introduce changes at each base with probability *p* (specified by the user). Errors are simulated by replacing the base with a randomly chosen alternative. We run these two transcripts through our pipeline and look at the ability of each tool to correctly partition and assemble the two transcripts separately, despite the high similarity.

4.3 Experiment results and analysis

In the following we outline the results for each type of experiments, with supporting figures when appropriate, and more detailed figures in the appendix. Note that, in all figures, we will reflect EAST-related plots with blue lines and CAP3-related plots with green lines.

4.3.1 Simulated data

We used three types of simulated data: Sanger sequences, 454 sequences and Illumina sequences. The number of ESTs in each experiment can be estimated using the following formulas:

Sanger sequences experiment: coverage×7000×15/800
454 sequences experiment: coverage×7000×15/230
Illumina sequences experiment: coverage×7000×5/62

For the coverage of 30, the number of ESTs that the assembly tool deals with is about 3938, 13696, and 16935 for the Sanger, the 454, and the Illumina tests, respectively.

4.3.1.1 Sanger sequences:

For simulated Sanger sequences, we were able to measure the quality of our results against those of CAP3 when applied to the same data sets. We use ESTSim to simulate Sanger reads. The errors introduced into the simulated ESTs include insertion, deletion and substitution errors; the exact parameters provided to the ESTSim simulation tool are discussed in Appendix II.

We set two parameters (coverage depth and error rate) for each run. The coverage depth ranges from 10 to 50 by increments of 10, while the error rate is varied from 0% to 6% in increments of 1%. In each experiment the pipeline randomly selected 15 out of a set of 42 transcripts and generated the ESTs from the set of transcripts using ESTSim with test parameters (error rate and coverage). Next, it fed the ESTs to EAST and CAP3, and collected the results.

As the simulations are stochastic, the results of each experiment (i.e. combination of parameters) are an average over fifty runs of that experiment.

Success in reconstructing whole transcripts: In this test, each tool is given ESTs derived from 15 randomly selected genes of comparable length, covered to an extent that it should be possible to reconstruct each of the transcripts in full – the success of which is reflected in our *NN* and *NA* statistic (defined above). *NN* reflects the number of consensus sequences constructed by the tool, which should be exactly equal to the number of genes. *NA* reflects the quality of the consensus sequence: how faithfully the transcript has been inferred (as measured by the normalized A-score statistic). In both cases, the ideal value is 1. We tested both tools with varying coverage and error rate, with results in Figure 12 until Figure 17 (Appendix II).

From Figure 12 to Figure 17, we see that EAST is faster than CAP3 for all the combination of error rate and coverage – with EAST keeping its runtime under 500 seconds while CAP3 going beyond 1200 seconds when coverage is 50 and error rate is 2%. For a moderate coverage (bigger than 10), both tools do quite well for low-error sequences, but as those error rates increases past the (fairly low) rate of 2%, CAP3 results quickly deteriorate while EAST is unaffected in terms of both *NN* and *NA* – with EAST still producing exactly the 15 transcripts at an error rate of 6%, against CAP3's 450, and EAST still keeping 1 (a perfect reconstruction) for *NA* at an error rate of 6% against CAP3's 0.45 for its best consensus sequence.

- **Comparison of normalized number of consensus sequences:**
  We compared the normalized number of consensus sequences (*NN*) with two types of plots. First, we fixed the error rate and plot *NN* against a range of coverage depths. Second, we fixed coverage depth and plot *NN* against a range of error rates. We can see in Figure 12 that as the error rate increases, EAST finds the expected 15 sequences (or comes very close), while CAP3 splits up the 15 transcripts into more and more contigs. The same thing happened when coverage depth increases in Figure 13.

For example, at a 0.06 error rate and 30 coverage depth, EAST produces 15 sequences (*NN*=1) to Cap3's 450 (*NN*≈50) – a 97% improvement, and implying that EAST is better suited for assembling error-prone data.

- **Comparison of normalized A-Score:**
  Similarly, we calculated Mean of Normalized A-Score(*NA)* with two types of plots.
  In Figure 14, we can see that when coverage depth is equal to 10, CAP3 works better than EAST when error rate is lower than 0.02. When coverage depth becomes bigger or equal to 20, EAST works better than CAP3 in all error rates from 0 to 0.06.
  In Figure 15, we can see that when the error rate is less than 0.02, EAST and CAP3 get similar normalized A-scores for coverage depth bigger than 10. But when error rate passes 0.02, EAST gets larger normalized A-scores than CAP3.
- **Comparison of runtime:**
  We calculated Runtime (T*)* with the same two types of plots. From Figure 16 and Figure 17, we can see that EAST uses less time than CAP3 on all the combination of coverage depth and error rate. For example, when error rate is 3% and coverage is 30, EAST takes an average runtime of 180 seconds to reconstruct 15 transcripts with an average length of 6900 bases against CAP3's 510 seconds – i.e., EAST is 65% faster.

4.3.1.2 454 sequences:

For simulated 454 sequences, we measured the quality of our results against those of CAP3 when applied to the same data sets. We use MetaSim to simulate 454 reads. Different from simulated Sanger reads, we employed the default error model provided by MetaSim for 454 sequences test.

We set one parameter (coverage depth) for each run. The coverage depth ranges from 10 to 50 by increments of 10. In each experiment the pipeline randomly selected 15 out of a set of 42 transcripts and generated the ESTs from the set of transcripts using MetaSim with test parameters (coverage). Next, it fed the ESTs to EAST and CAP3, and collected the results.

As the simulations are stochastic, the results of each experiment (i.e. combination of parameters) are an average over thirty runs of that experiment. We use the same four evaluation standards as Sanger sequences test to compare the EAST to CAP3.

From Figure 18 to Figure 20, we see that CAP3 results quickly deteriorate while EAST is unaffected in terms of *NN* – with EAST still producing exactly the 15 transcripts when

coverage increases. For those different coverage EAST keeps *NA* within the range of 0.9-1.0 against CAP3's 0.1-0.3.

- **Comparison of normalized number of consensus sequences:**
  We plotted *NN* against a range of coverage depths. We can see in Figure 18 that as the coverage increases, EAST finds the expected 15 sequences (or comes very close), while CAP3 splits up the 15 transcripts into more and more consensus sequences.
  For example, at a 30 coverage depth, EAST produces 15 sequences (*NN*=1) to Cap3's 2100 (*NN*≈140) – a 99% improvement.
- **Comparison of normalized A-Score:**
  In Figure 19, we can see the *NA* from EAST is averagely 0.7 higher than CAP3 for all different coverage depths.
- **Comparison of runtime:**
  From Figure 20, we can see that EAST uses longer time than CAP3 on all the coverage depths. Since EAST gets considerably better results than CAP3, it is worth the sacrifice in runtime.

4.3.1.3 Illumina sequences:

We also use MetaSim to simulate Illumina reads. Similar to 454 reads, we employed the default error model provided by MetaSim for the test.

We set the same parameter (coverage depth) for each run as 454 test. In each experiment the pipeline randomly selected 5 out of a set of 42 transcripts and generated the ESTs from the set of transcripts using MetaSim with test parameters (coverage). Next, it fed the ESTs to EAST and CAP3, and collected the results.

As the simulations are stochastic, the results of each experiment (i.e. combination of parameters) are an average over thirty runs of that experiment. We use the same four evaluation standards as Sanger sequences test to compare the EAST to CAP3.

From Figure 21 to Figure 23, we see that EAST consistently works better than CAP3 in terms of *NN* and *NA* although both of them do not work for data sets with coverage less than 20.

- **Comparison of normalized number of consensus sequences:**
  We plotted *NN* against a range of coverage depths. We can see in Figure 21 that as the coverage increases, both tools move closer to the perfect value 5 – but EAST maintains a significant lead throughout.

For example, at a 30 coverage depth, EAST produces 24 sequences (*NN*=4.8) to Cap3's 611 (*NN*≈122.2).

- **Comparison of normalized A-Score:**
  In Figure 19, we can see the *NA* from EAST converges towards 1 when coverage increases, while CAP3 is limited to an NA value of 0.1 for all different coverage depth.
- **Comparison of runtime:**
  From Figure 20, we can see that EAST uses longer time than CAP3 on all the coverage depths. Since EAST gets considerably better results than CAP3, it is worth the sacrifice in runtime.

4.3.1.4 Duplication differentiation Test

In the duplication differentiation test, we create an "artificial" duplicated transcript by copying it and allowing it to change at a specified rate, and then generate ESTs of size 700-1000 at a coverage depth of 30 through the ESTSim tool covering both transcripts. *p*, the probability of a base that is changed in the copy, ranges from 0 to 0.8 in increments of 0.1. We used *the Normalized number of consensus sequences (NN)* and *the Mean of Normalized A-Score (NA)* to evaluate the tools.

The results of the test indicate that CAP3 works better than EAST in differentiating duplicated transcripts in terms of *NA* and *NN*. The left plot of Figure 10 shows that CAP3 produces consensus sequences with almost a constant value of *NA* (that is also the perfect value 1), but EAST displays a lower *NA* than CAP3 for all the base change rates. The right plot shows that CAP3 successfully differentiate the two duplicated transcripts in most cases (*p≥0.1*) while EAST fails to get perfect value in all the cases.

The result is not a surprise: it has been established that PEACE fails to differentiate clusters under the same condition. This result demonstrates only that EAST is not able to correct the problem.

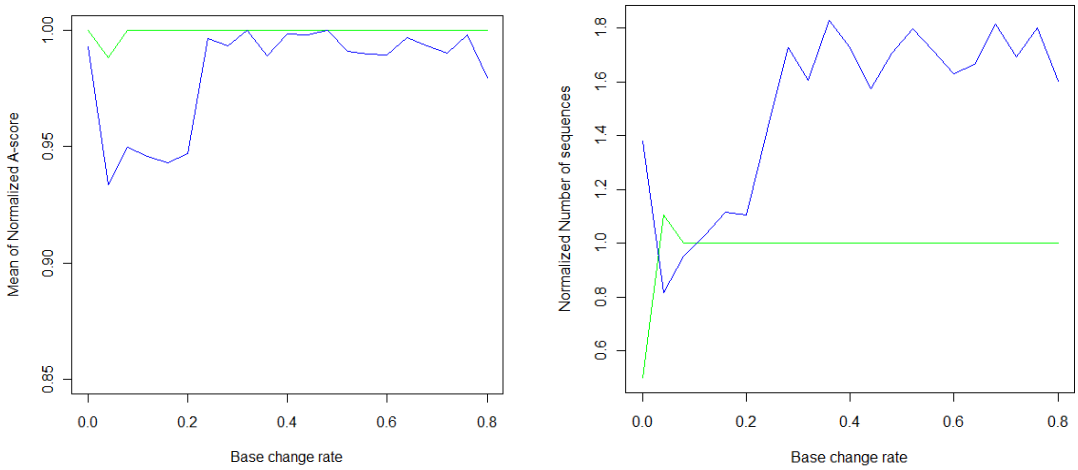


Figure 9: duplication differentiation test

5. Summary

In the thesis, we implemented EAST - a tool for the *de novo* assembly of Expressed Sequence Tags into transcripts. Accepting an EST file and a corresponding MST file describing the MST generated from the EST file, EAST assembles all the ESTs and generates consensus sequences to estimate the transcripts.

EAST is a *de novo* assembly tool which does assembly without the knowledge of the genome. We developed a new MST based algorithm to analyze the relationship among the ESTs and do reconstruction from them. We tested EAST on both simulated data. The experiments showed that EAST works for Sanger sequencing reads, 454 reads, and Illumina reads. Our tests also proved that EAST works better than CAP3 in terms of the quality of generated consensus sequences.

Bibliography

1. Liang, F., et al., *An optimized protocol for analysis of EST sequences.* Nucl. Acids Res., 2000. **28**(18): p. 3657-3665.

2. Burke, J., D. Davison, and W. Hide, *d2_cluster: A Validated Method for Clustering EST and Full-Length cDNA Sequences.* Genome Research, 1999. **9**(11): p. 1135-1142.

3. Malde, K., E. Coward, and I. Jonassen, *A graph based algorithm for generating EST consensus sequences.* Bioinformatics, 2005. **21**(8): p. 1371-1375.

4. Heber, S., et al., *Splicing graphs and EST assembly problem.* Bioinformatics, 2002. **18**(suppl_1): p. S181-188.

5. Huang, X. and A. Madan, *CAP3: A DNA Sequence Assembly Program.* Genome Research, 1999. **9**(9): p. 868-877.

6. Quackenbush, J., et al., *The TIGR Gene Indices: analysis of gene transcript sequences in highly sampled eukaryotic species.* Nucl. Acids Res., 2001. **29**(1): p. 159-164.

7. Quackenbush, J., et al., *The TIGR Gene Indices: reconstruction and representation of expressed gene sequences.* Nucl. Acids Res., 2000. **28**(1): p. 141-145.

8. Hide et al., *Biological Evaluation of d2, an Algorithm for High-Performance Sequence Comparison*. Journal of Computational Biology (1994) vol. 1 (3) pp. 199-215.

9. Smith, TF, Waterman MS, *Identification of Common Molecular Subsequences*. Journal of Molecular Biology, Volume 147, Issue 1, 25 March 1981, p. 195-197. doi:10.1016/0022-2836(81)90087-5

10. Needleman SB, Wunsch CD, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. Volume 48, Issue 3, 28 March 1970, p. 443-453. doi:10.1016/0022-2836(70)90057-4

11. D.M. Rao, J.C. Moler, M. Ozden, Y. Zhang, C. Liang and J.E. Karro, *"PEACE: Parallel EST Analysis and Clustering Engine"*, in preparation

12. Genetics: a Conceptual Approach, Pierce, B., 3rd edition

13. Sanger F, Nicklen S, Coulson AR (December 1977). "DNA sequencing with chain-terminating inhibitors". Proc. Natl. Acad. Sci. U.S.A. 74 (12): 5463–7. doi:10.1073/pnas.74.12.5463. PMID 271968

14. Schuster, Stephan C. (2008). Next-generation sequencing transforms today's biology. 5. Nature Methods. pp. 16–18. doi:10.1038/nmeth1156

15. http://www.454.com/

16. http://www.illumina.com/

18. Judith Zimmermann, Zsuzsanna Liptak, Scott Hazelhurst, "A Method for Evaluating the Quality of String Dissimilarity Measures and Clustering Algorithms for EST Clustering," bibe, pp.301, Fourth IEEE Symposium on Bioinformatics and Bioengineering (BIBE'04), 2004

19. R. C. Prim: Shortest connection networks and some generalizations. In: Bell System Technical Journal, 36 (1957), pp. 1389–1401

20. Hazelhurst and Bergheim. ESTSim: A tool for creating benchmarks for EST clustering algorithms. Dept. of Computer Science, Univ. of Witwatersr and (South Africa), Tech. Rep. CS-2003-1 (2003)

21. Nagaraj et al. A hitchhiker's guide to expressed sequence tag (EST) analysis. Brief Bioinformatics (2007)

22. Scott Hazelhurst, Winston Hide, Zsuzsanna Lipt´ak, Ramon Nogueira, and Richard Starfield. An overview of the wcd est clustering tool. Bioinformatics, 24(13):1542–6, Jul 2008.

23. D.C. Richter, F. Ott, A.F. Auch, R. Schmid and D.H. Huson, MetaSim- A Sequencing Simulator for Genomics and Metagenomics, 2008 PLoS ONE, accepted.

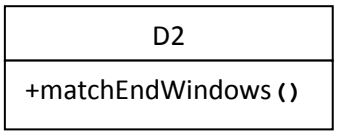
24. http://www.pesolelab.it/easycluster/

25. Hall N (May 2007). "Advanced sequencing technologies and their wider impact in microbiology". *J. Exp. Biol.* **210** (Pt 9): 1518–25.

Appendix I: System Implementation

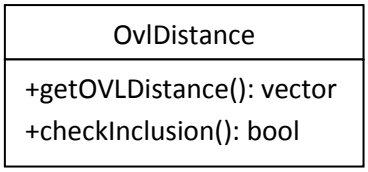
The system is written in C++. The main classes in the system include:

1) D2: enclose all the methods related to d^2 distance calculation.



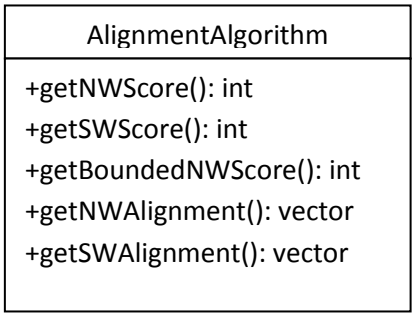
matchEndWindows() method accepts two strings, and returns all the windows which contribute to their d^2 score.

2) OvlDistance: enclose all the methods related to overlap distance calculation.



getOVLDistance() method accepts two strings, and returns their overlap distance and overlap length.
checkInclusion() method accepts two strings, and returns true of false to indicate if one strings includes another.

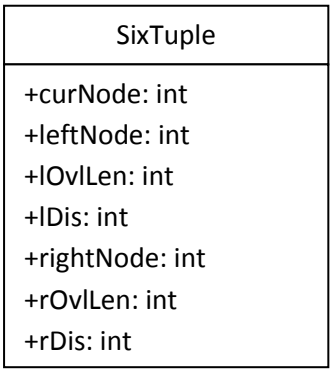
3) AlignmentAlgorithm: enclose all the methods related to global alignment and local alignment



All the above methods accept two strings. getNWScore() returns the global alignment score using the Needleman-Wunsch algorithm. getSWScore()returns the local alignment score using the Smith-Waterman algorithm. getBoundedNWScore() returns the global alignment score using bounded dynamic programming to save time.

getNWAlignment() and getSWAlignment() return the global alignment and local alignment respectively.

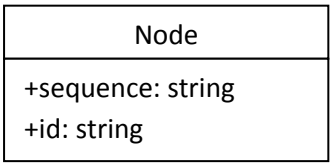
4) SixTuple: store the six-tuple of a node



curNode is the index of the current node. leftNode is the index of the node which is to the current node's left. lOvlLen is the overlap length between leftNode and curNode. lDis is the overlap distance between leftNode and curNode.

Similarly, rightNode is the index of the node which is to the current node's right. rOvlLen is the overlap length between rightNode and curNode. rDis is the overlap distance between rightNode and curNode.

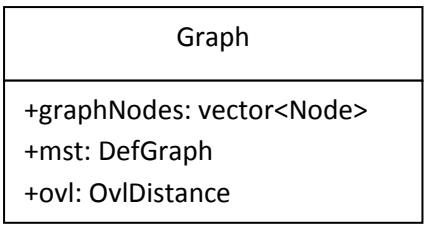
5) Node: store a node



sequence is the EST string which corresponds the node.

id records the relevant information for the current node. The content depends on the input EST file, typically it includes the information like what gene the EST comes from, the starting position and the ending position of the EST in the gene.

6) Graph: store the input MST and enclose all the relevant operation on the MST.



graphNodes is a vector of Node objects. Each element in it is a node in the input MST. mst is a data structure storing the input MST. ovl is an OvlDistance object.

7) SixTuplesGeneration: enclose all the methods related to generating six-tuples for all the nodes.

SixTuplesGeneration
+g: Graph +alignArray: vector<SixTuple>
+init() -createAlignArray() -processAlignArray()

g is a Graph object. alignArray stores all the six-tuples for all the nodes.

creatAlignArray() method creates six-tuples for each node. processAlignArray() finds out all the left ends based on the six-tuples.

init() call creatAlignArray() and processAlignArray() to finish the procedure of six-tuples generation and left ends identification.

8) Reconstruction: enclose all the methods related to do reconstruction from the six-tuples.

Reconstruction
+leftMostNodes: vector<SixTuple> +alignArray: vector<SixTuple>
+getConsensus() -reconstuctForEachCluster(): string -processLeftEnds(): string -processLeftEndsWithInclusion(): string -reconstructFromEnds(): string -reconstructSeq(): string

leftMostNodes stores the six-tuples for all the left ends. alignArray stores the six-tuples for all the nodes.
getConsensus() method reconstructs the sequence from all the nodes and write the consensus sequences and singletons into the specified files.
reconstuctForEachCluster() methods separates multiple clusters and do reconstruction for each cluster.

processLeftEnds() is designed to process a single cluster. It evaluates the relationship of those left ends in the cluster and groups them according to their inclusion relationship. All the related left ends are put into one group.

processLeftEndsWithInclusion() is designed to do reconstruction for each group of left ends. It assigns a starting position for each left end.

reconstructFromEnds() accepts a set of left ends. Starting from each left end, it calculates the starting positions for all the nodes which can be directly or indirectly connected to the left end.

reconstructSeq() does reconstruction from a set of nodes. Given the starting positions of these nodes, this method progressively aligns these nodes and generates a consensus sequence from them.

The invoking relations between the above methods are as follows:

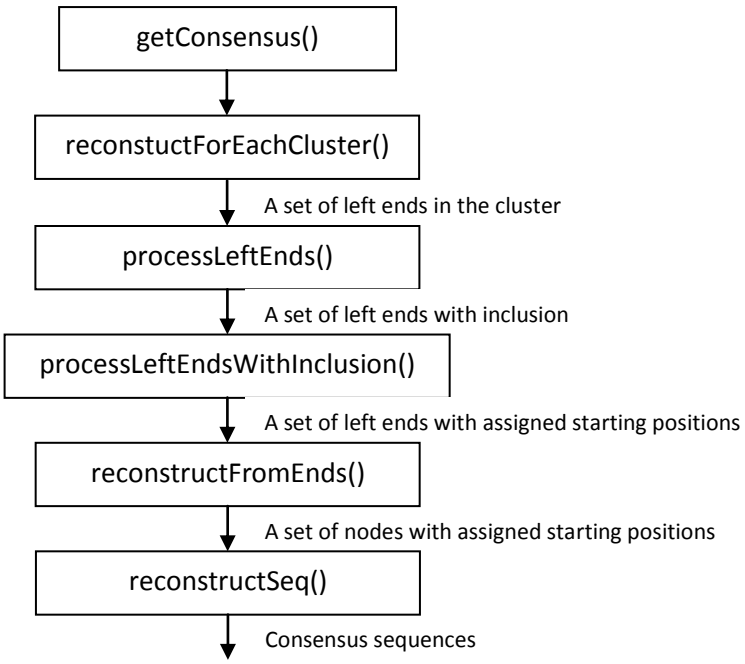


Figure 10: The invoking structure of methods in the class

9) ESTAssembly: main logic of the system. It reads into the input EST file and MST file, call the method in SixTuplesGeneration to generate six-tuples, and then call the method in Reconstruction to assemble all the ESTs.

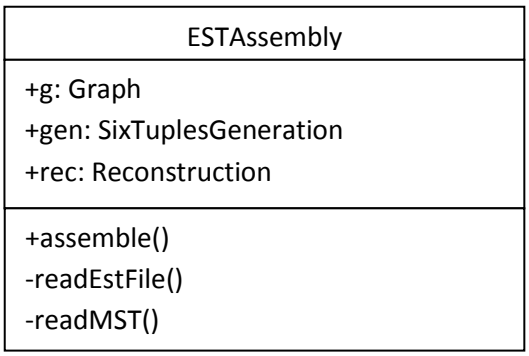


Figure 11: Data Structure of the system

Appendix II: Experiment results

Part I: parameters of ESTSim

ESTSim provides a list of parameters to simulate errors in the simulated data. Below is the explanation to these parameters[20].

α: Probability that an arbitrary base changes randomly.
β: Margin at beginning of the EST where errors are most likely to happen.
γ: Probability that an error happens in the margin.
ζ: Effect of polymerase decay at end of read.
ξ: The rapid polymerase decay factor.
κ: proportion of errors that are substitutions.
λ: Gives proportion of changes that are deletions
μ: Gives proportion of changes that are insertions
ν: Gives proportion of changes that are Ns

If an error does happen, there are four things that could happen. A base could be arbitrarily changed, deleted or inserted, or an N could be inserted. κ, λ, μ, ν are given as ratios with respect to each other. Let $E = \kappa + \lambda + \mu + \nu$. Then, κ/E is the probability that if an error occurs it is a substitution. Similarly, λ/E is the probability that if an error occurs it is a deletion.

The corresponding parameters we used in the test are:

α = error rate
$\beta = 20$
γ = error rate
$\zeta = 1$
$\xi = 1$
$\kappa = 10$
$\lambda = 1$
$\mu = 1$
$\nu = 2$

The error rate in our test ranges from 0 to 0.06 increasing by 0.01. If an error happens, it has 5/6 possibility to be a substitution error, 1/14 to be an insertion error, 1/14 to be an deletion error, and 1/7 to be converted to 'N'.

Part II: Figures for Sanger sequencing test

- Figure 12:

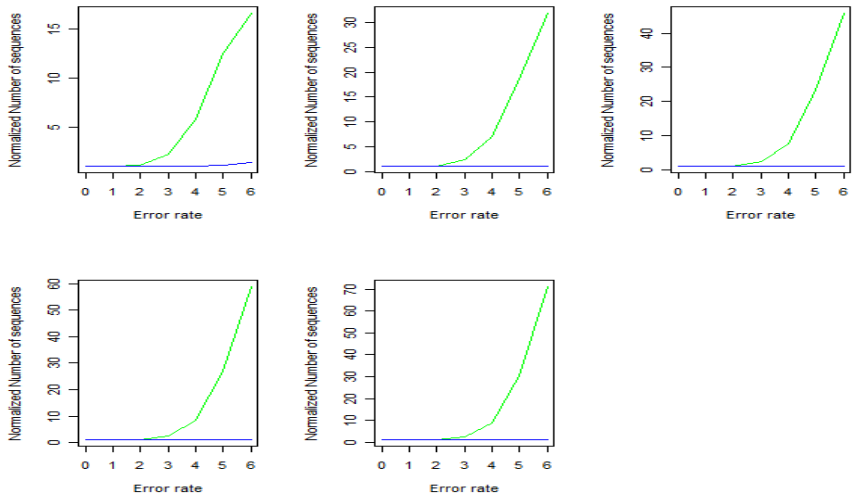


Figure 12: Fixed coverage depth (10-50 from left upper corner to right corner separately), plot for error rate to *NN*.

➢ Figure 13:

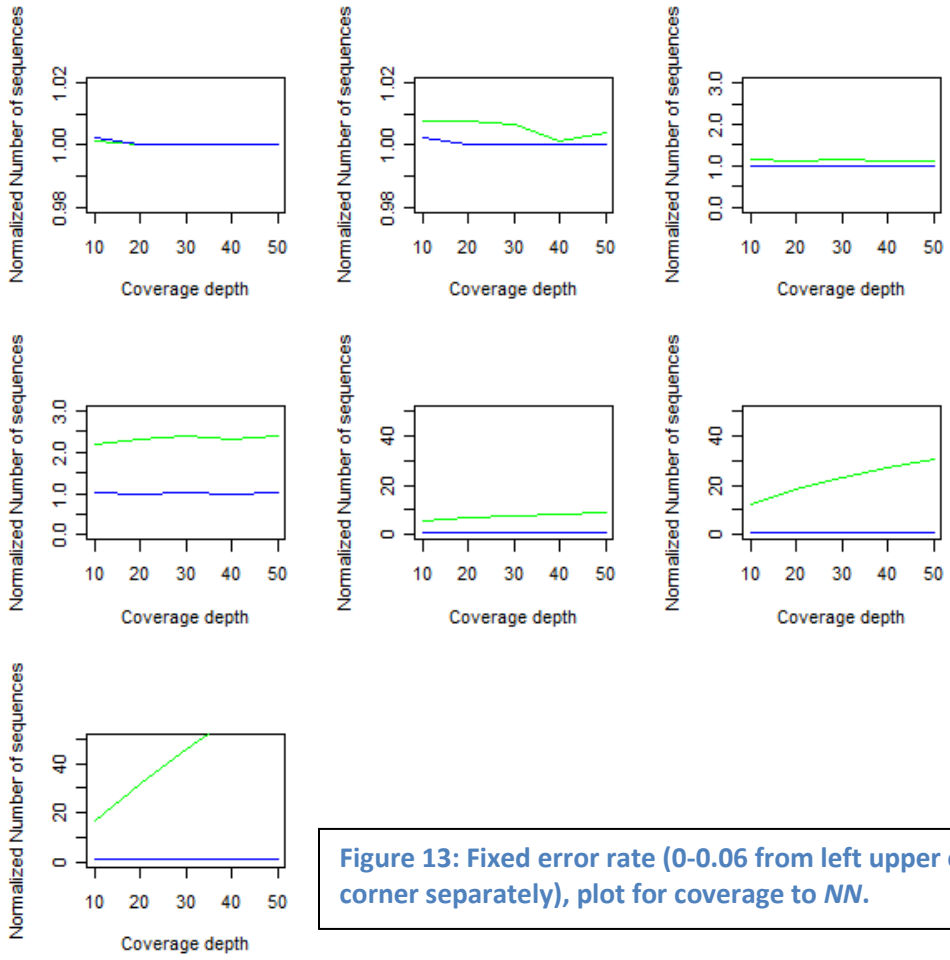


Figure 13: Fixed error rate (0-0.06 from left upper corner to right corner separately), plot for coverage to *NN*.

➢ Figure 14:

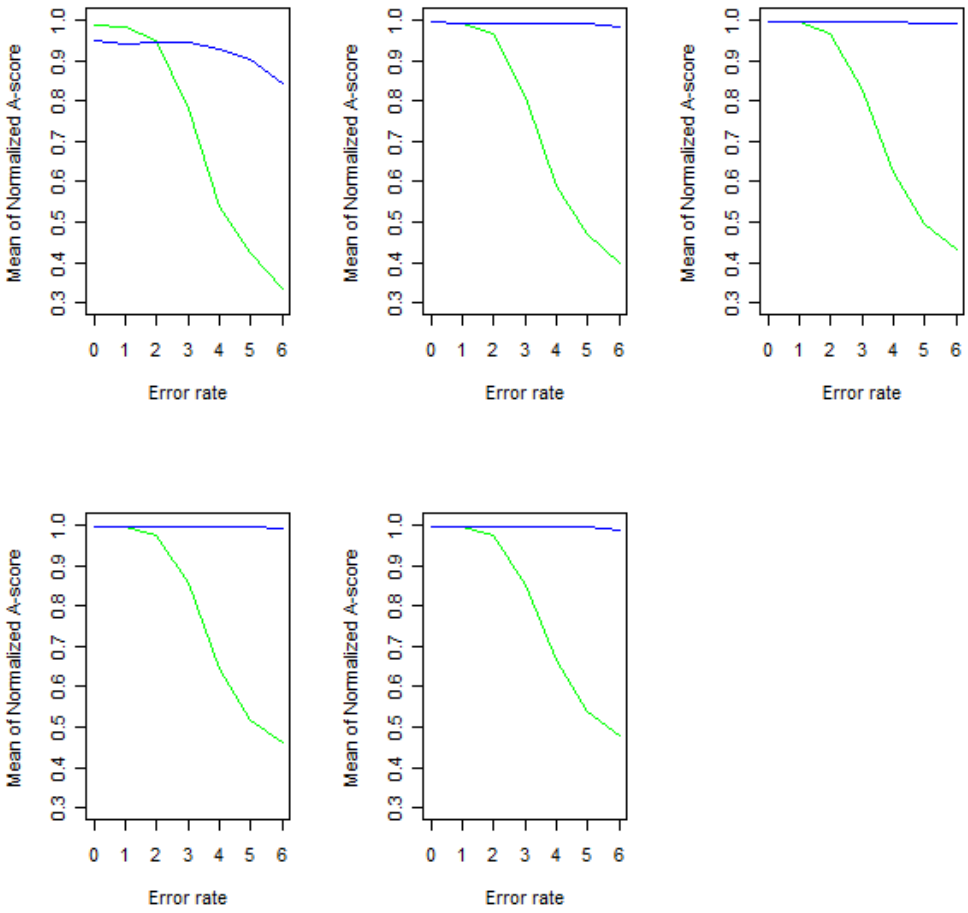


Figure 14: Fixed coverage depth (10-50 from left upper corner to right corner separately), plot for error rate to *NA*.

- Figure 15:

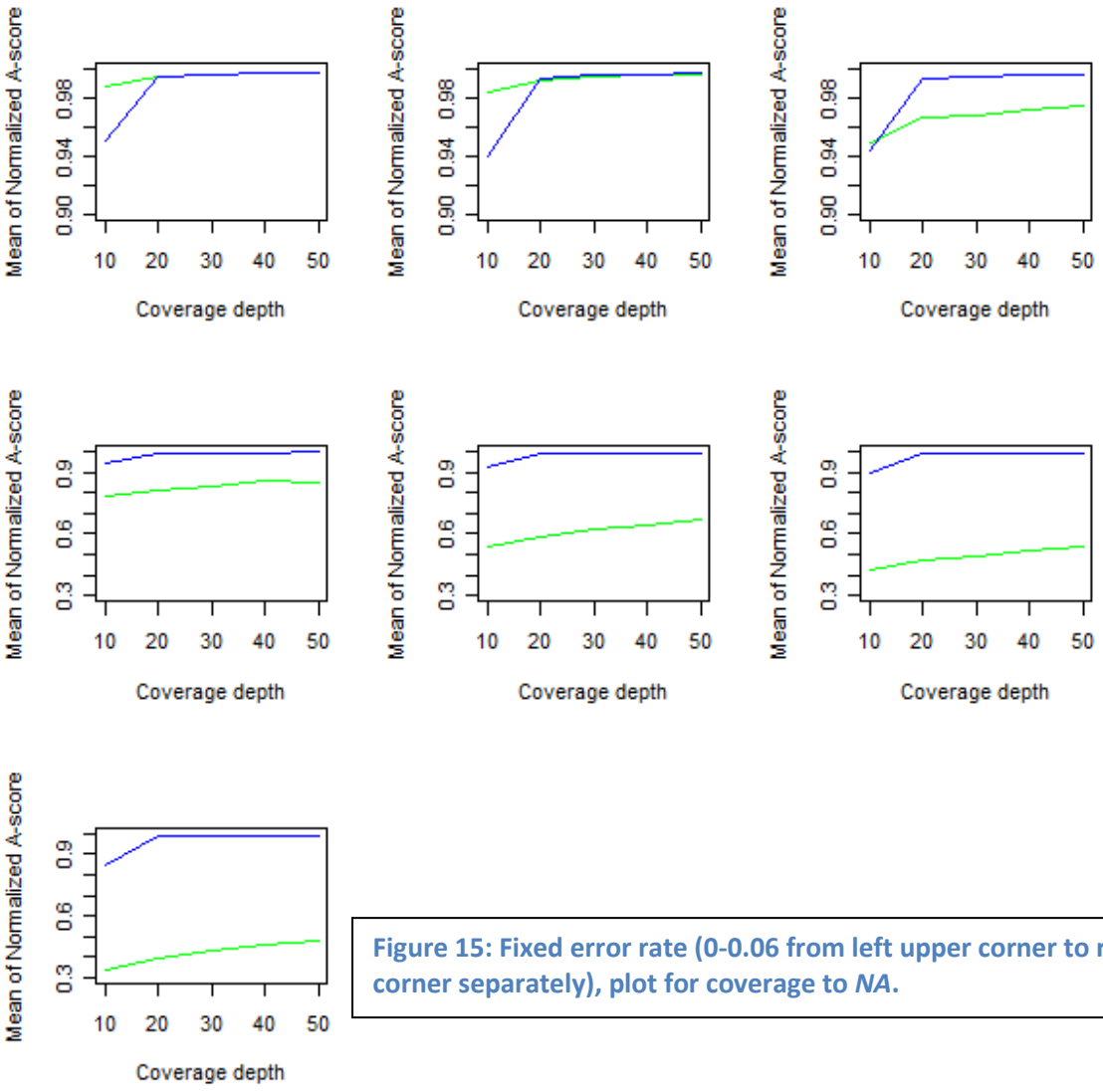


Figure 15: Fixed error rate (0-0.06 from left upper corner to right corner separately), plot for coverage to *NA*.

- Figure 16:

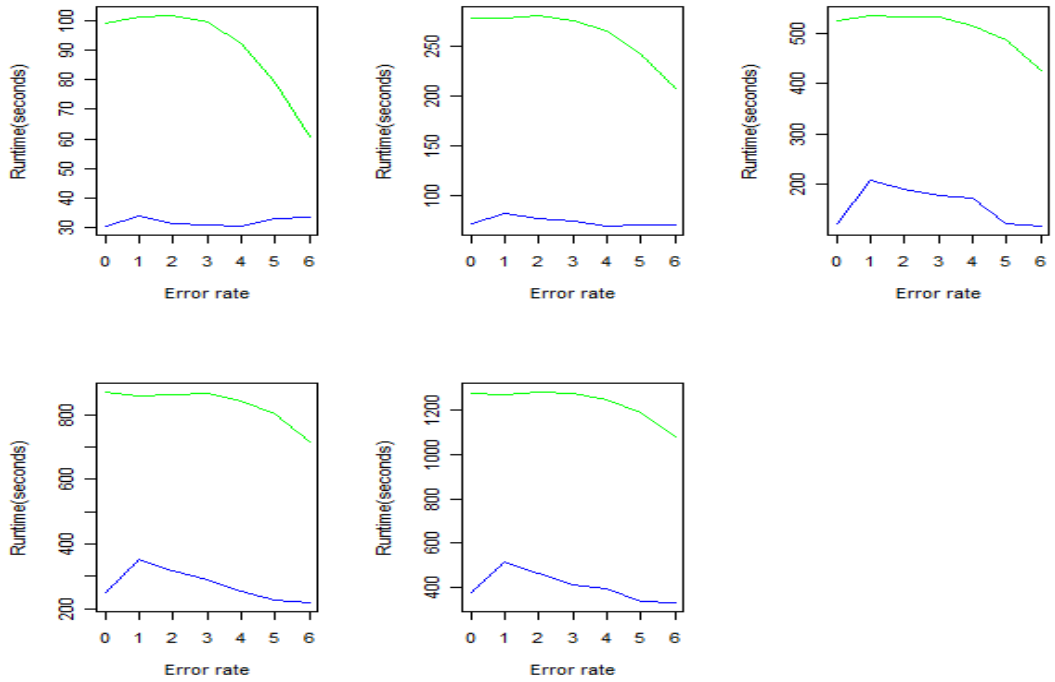


Figure 16: Fixed coverage depth (10-50 from left upper corner to right corner separately), plot for error rate to *T*.

- Figure 17:

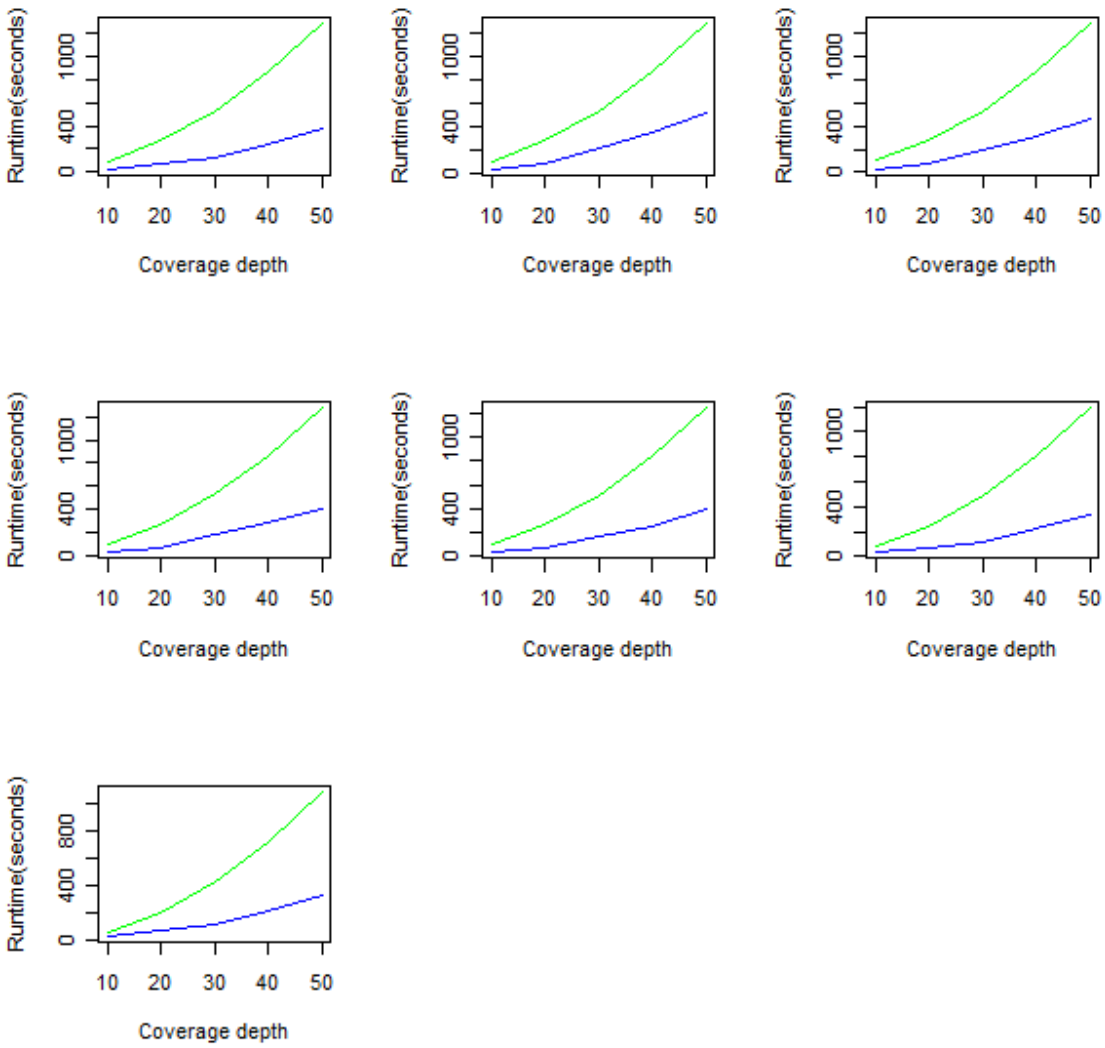


Figure 17: Fixed error rate (0-0.06 from left upper corner to right corner separately), plot for coverage to *T*.

Part III: Figures for 454 sequencing test

- Figure 18:

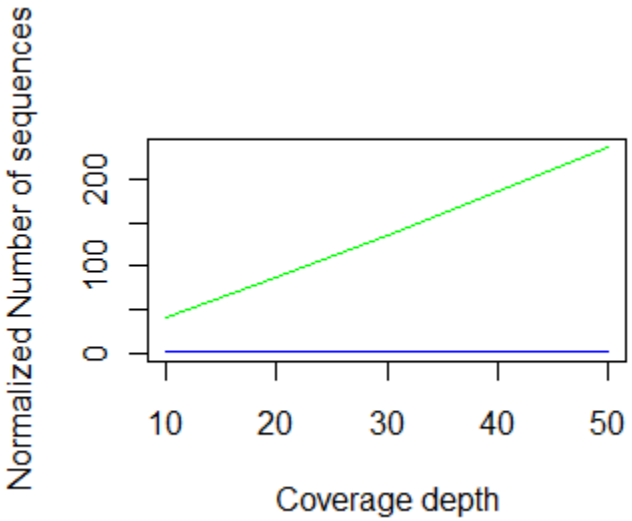


Figure 18: Plot for coverage to *NN*

- Figure 19:

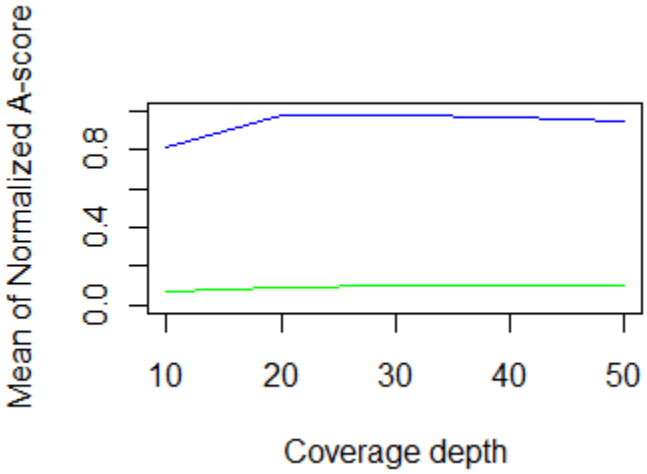


Figure 19: Plot for coverage to *NA*

- Figure 20:

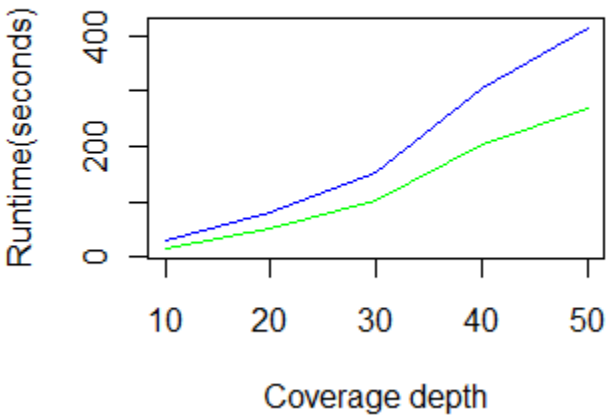


Figure 20: Plot for coverage to *T*

Part IV: Figures for Illumina sequencing test

- Figure 21:

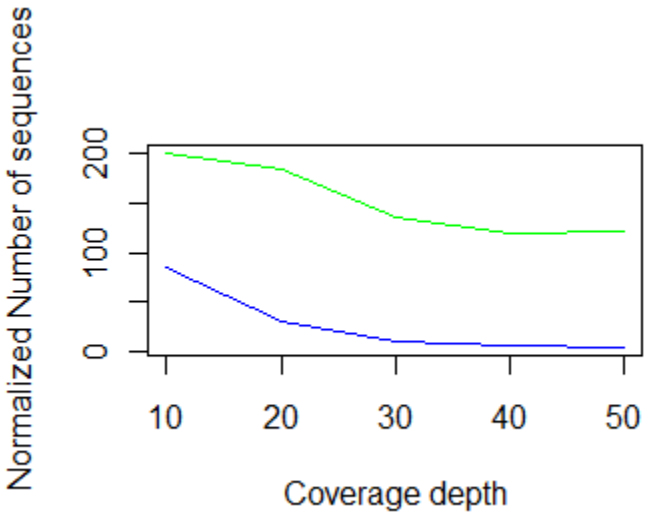


Figure 21: Plot for coverage to *NN*

- Figure 22:

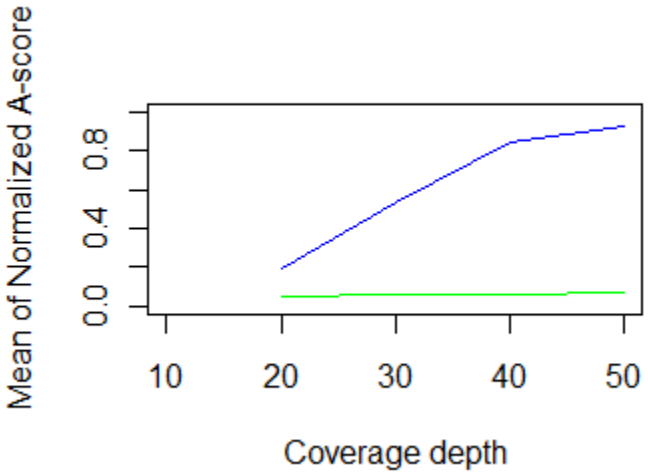


Figure 22: Plot for coverage to *NA*

- Figure 23:

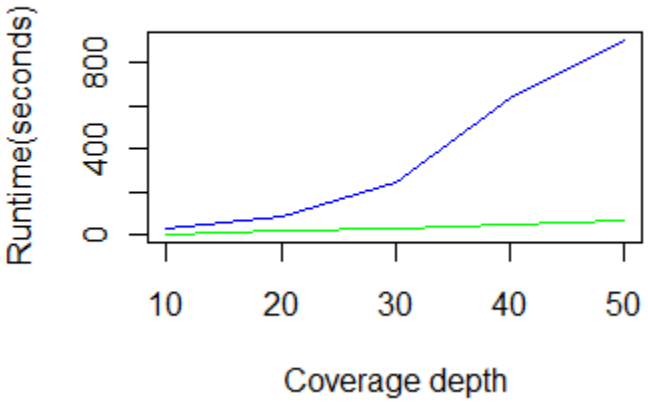


Figure 23: Plot for coverage to *T*