2008

DNA Fragment Assembly Algorithms: Toward a Solution for Long Repeats

Ching Li

San Jose State University

DNA FRAGMENT ASSEMBLY ALGORITHMS:
TOWARDS A SOLUTION FOR LONG REPEATS

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Ching Chia Li

May 2008

ABSTRACT

DNA FRAGMENT ASSEMBLY ALGORITHMS:
TOWARDS A SOLUTION FOR LONG REPEATS

by Ching Chia Li

In this work, we describe our efforts to seek optimal solutions for the DNA Fragment Assembly Problem in terms of assembly accuracy and runtime efficiency. The main obstacles for the DNA Fragment Assembly are analyzed. After reviewing various advanced algorithms adopted by some assemblers in the bioinformatics industry, this work explores the feasibility of assembling fragments for a target sequence containing perfect long repeats, which is deemed theoretically impossible without tedious finishing reaction experiments. Innovative algorithms incorporating statistical analysis proposed in this work make the restoration of DNA sequences containing long perfect repeats an attainable goal.

Contents

1. Introduction

The Human Genome Project aims to identify the exact sequence of nucleotide base pairs for the entire human genome. The Human genome contains about three billion nucleotide base pairs; however, current technologies usually sequence DNA fragments shorter than 1000 bases [1]. Large DNA sequences are always cut into small fragments for analysis and then assembled together to restore the original sequence. Thus, the bioinformatics industry needs efficient algorithms for the precise assembly of long DNA sequences from DNA fragments that are shorter than 1000 bases.

2. Background

DNA sequences, determining protein synthesis of biological entities, are important for scientists to understand the functioning of various organisms. Long and complicated as they are, all DNA sequences consist of four nucleotides – A (adenine), T (thymine), C (cytosine), and G (guanine), which are termed "bases" [1]. In 1982, Frederick Sanger led a group of scientists to sequence the 48,000 base-pairs long genome of a virus, *Bacteriophage lambda*, utilizing the DNA whole genome shotgun sequencing method [2]. Since then, the DNA whole genome shotgun sequencing method continues to evolve in terms of scalability, accuracy, and robustness. In 2001, the initial human genome sequencing of three and a half billion base-pairs was accomplished by this method. Improvements in DNA fragment assembly algorithms contribute significantly to the success of the shotgun sequencing method.

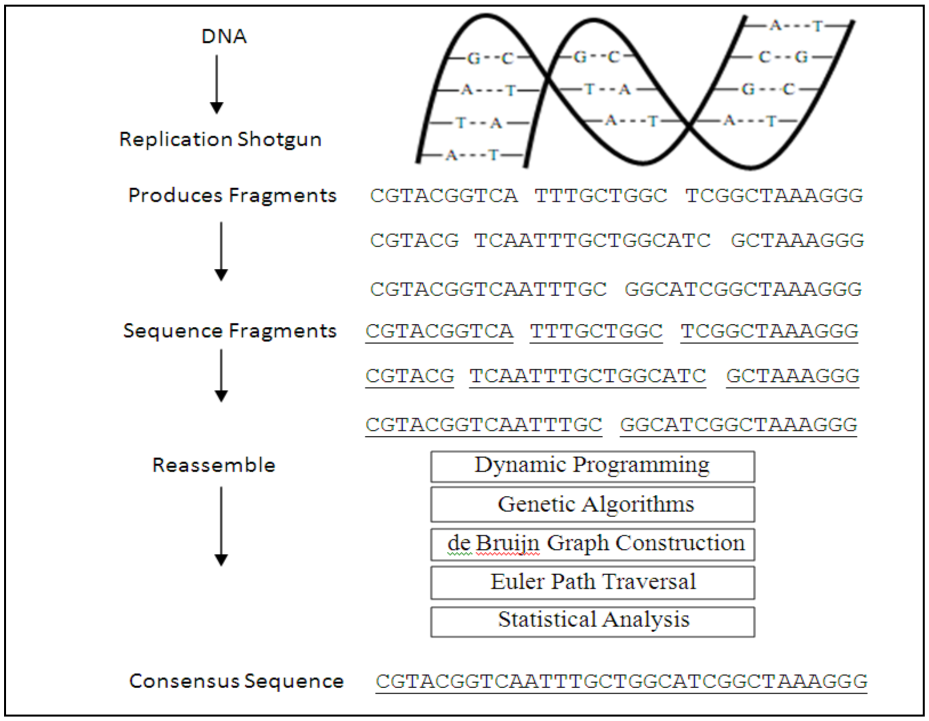
DNA
Replication Shotgun
Produces Fragments
CGTACGGTCA TTTGCTGGC TCGGCTAAAGGG
CGTACG TCAATTTGCTGGCATC GCTAAAGGG
CGTACGGTCAATTTGC GGCATCGGCTAAAGGG
Sequence Fragments
CGTACGGTCA TTTGCTGGC TCGGCTAAAGGG
CGTACG TCAATTTGCTGGCATC GCTAAAGGG
CGTACGGTCAATTTGC GGCATCGGCTAAAGGG
Reassemble
Dynamic Programming
Genetic Algorithms
de Bruijn Graph Construction
Euler Path Traversal
Statistical Analysis
Consensus Sequence
CGTACGGTCAATTTGCTGGCATCGGCTAAAGGG

Figure 1: DNA Sequencing Process

Though there are variations in the actual implementation of shotgun sequencing, they all follow a similar procedure:

1. Target sequence cloning – multiple copies of a DNA sequence under analysis are created.
2. For DNA fragment creation, each copy of the target sequence is fractured randomly with sonication or nebulation; fragments that are too long or too short are removed due to DNA sequence analysis performance concerns; statistically, fragment length has a normal distribution of about 10% variance after screening.
3. DNA sequence analysis on fragments, where fragments are inserted into engineered viruses to form vectors and a sequencing reaction is

performed in the vectors to produce a fragment read ranging from 300 to 900 bases long.

4. DNA fragment assembly – computational algorithms and expert knowledge are applied to put pieces of fragments back to a consensus sequence [3].

The more efficient and accurate the computational algorithm, the less intervention is required by biologists for DNA fragment assembly, and consequently the more efficient the DNA fragment assembly can be. Thus, the continuous improvement of DNA fragment assembly algorithm is crucial for biologists to study large scale DNA sequences efficiently.

3. Problem Definition

DNA fragment assembly reconstructs the original DNA sequence from a large number of fragments that are several hundred bases long. To accomplish this goal, all DNA fragment assembly algorithms need to overcome the following challenges:

Unknown orientation

Given the dual helix structure of DNA, each fragment can come from either strand of the helix structure. Thus, as we determine the fragment layout, we need to consider the reverse complement of each fragment, which essentially doubles our assembling efforts. A (adenine) complements T (thymine), while C (cytosine) complements G (guanine), and vice versa. To get the reverse complement of a fragment, we first reverse the fragment sequence; for example, the reverse of fragment ATGCTA is

ATCGTA; then taking the complement for each nucleotide base in the sequence, we have TAGCAT [1]. Essentially, the DNA fragment assembly results in two DNA base sequences complementing each other, each stands for one strand of the original DNA sequence.

Read	Orientation	Assembly
ATGCTA	←	--- TAGCAT ------------------
CATTGCC	→	--------- CATTGCC ----------
AATGC	←	--------GCATT ----------------
TGCCGTAG	→	----------------TGCCGTAG---

Figure 2: Calculating reverse complement of DNA fragments

Base-calling errors or sequencing errors

The technical constraint of analyzing less than one kbps sequences at a time is actually due to various read errors, since most sequence results longer than one kbps are filled with errors and therefore have to be discarded. Due to a complicated sequencing process, the DNA fragments are contaminated with base errors: Dideoxynucleotide (ddNTPs) are used to randomly fracture long DNA chains; however, the fluorescent signal, which tags ddNTPs, is weakened by the geometric distribution of concentration; in addition, molecules diffuse in the gel as they are read – longer fragment reads cause more molecules in the fragment diffuses; thus data quality at the end of fragments is usually inferior, while DNA polymerase (DNA copying enzyme affected by temperature) and sequencing reactions may hide some low quality data in the middle of high quality regions. The assumption of a uniform moving speed when reading a DNA sequence is also error prone, because various DNA strands often move on the gel at different speeds. Contamination and undiscovered vectors are two additional common factors causing errors [4].

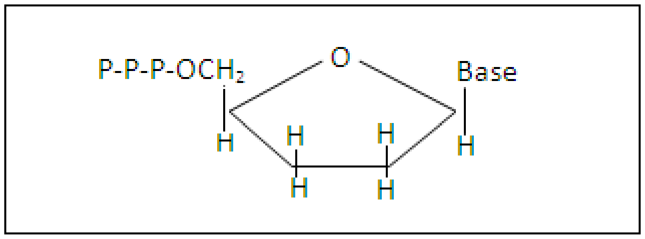

Figure 3: Chemical structure of ddNTP used to fracture DNA sequence

There are four different types of sequencing errors – Substitution (one base is reported as another base), Deletion (bases are not reported at all), Insertion (irrelevant bases are reported), and Ambiguity (uncertainty about the exact base). Table 1 lists the IUPAC Ambiguity codes for DNA sequence analysis [5].

IUPAC Code	Meaning	Complement
A	A	T
C	C	G
G	G	C
T/U	T	A
M	A or C	K
R	A or G	Y
W	A or T	W
S	C or G	S
Y	C or T	R
K	G or T	M
V	A or C or G	B
H	A or C or T	D
D	A or G or T	H
B	C or G or T	V
N	G or A or T or C	N

Table 1: IUPAC code meanings and complements

Figure 4 demonstrates how errors hinder fragment assembly.

```
Genome sequence:
         Repeat        Repeat
CTTCGCGTCATCATCACTTGAGTCATCATCACCTCGGA
Sequence reads in the correct layout:
 CTTCGCGTCATCATCA
        TCATCATCACTTGA
                 CTTGAGTCATCATCA
                       TCATCATCACCTCGGA
Fragments including some sequencing errors:
 CTTCGCGTCATCATCA
        TCATCATCAC*TTG*A
 CTT*GAGTCATCATCA
        TCATCATCACCTCGGA
```

Figure 4: Assembly errors caused fragment errors

Repeated regions

DNA sequences may contain many repeats. There are identical repeats as well as repeats with only slight differences. Repeats are difficult to resolve because there are multiple ways of joining related fragments together. Figure 5 provides a simple illustration of how repeats can cause assembly errors:

```
Genome sequence:
               Repeat        Repeat
      CTTCGCGTCATCATCATCACTTGAGTCATCATCATCACCTCGGA

Sequence reads in the correct layout:
      CTTCGCGTCATCATCA
                TCATCATCACTTGA
                         CTTGAGTCATCATCA
                                  TCATCATCACCTCGGA
Wrong layout:
      CTTCGCGTCATCATCA
             TCATCATCACTTGA
                      CTTGAGTCATCATCA
                            TCATCATCACCTCGGA
```

Figure 5: Assemble errors caused by repeats

The complexity of repeats actually goes much further. The length of repeats varies greatly and can be interspersed in numerous genomic locations or linked closely together. For instance, a trypsinogen gene has a five-fold repeat as long as four kbp with 3-5% variations among each fold. Three folds of the repeat locate so closely together that they confuse assembly algorithms with potential errors. Given the technology constraint of sequencing fragments shorter than one kbp, assembling long perfect repeats is deemed unsolvable [6]. The second half of our work is dedicated to developing algorithms that incorporate statistical analysis to put together the correct assembly for fragments containing long repeats.

Incomplete coverage

Given a target sequence of length L and N fragments of average length r, the genome coverage $C = N \cdot r/L$. There is a tradeoff between high coverage to ensure original DNA sequence cover probability and the computational complexity of the fragment assembler. Though no high coverage ensures the complete covering of target sequence due to the random fracturing process, coverage of eight to ten are preferred in common practice. For example, to get 10X coverage in a sequence of length 125 kb, we need 2,604 random fragments read with an average length of 480 bases:
$2604 \cdot 480/\ 125{,}000 = 10$. Because of the double helix DNA structure, we need to consider the reverse complement of each fragment [6].

Random creation of DNA fragments can lead to the situation where the coverage is insufficient to assemble all fragments to a consensus sequence and instead result in several long fragments. In Figure 6, judgments have to be made to determine the assembly orders of two contigs (long partially assembled DNA fragments).

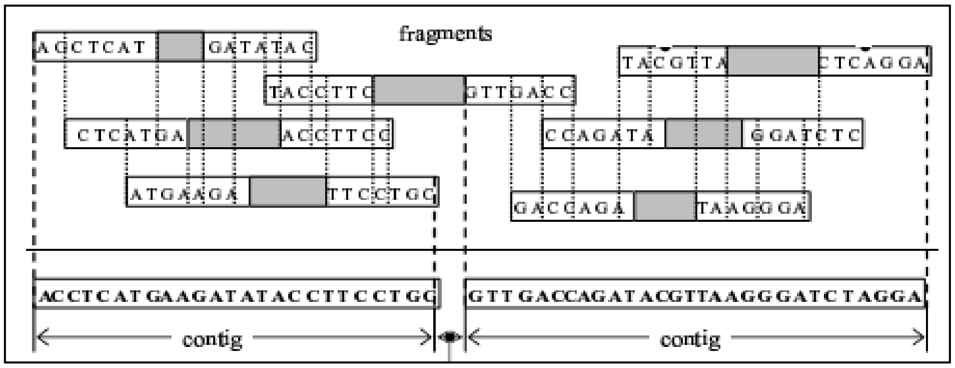

Figure 6: Failure to restore the target sequence due to no fragment coverage on some [22]

In summary, we define the DNA fragment assembly problem as the following: given a collection of fragment reads $F=\{f_i\}^R_{i=1}$ that are sequences over $\sum=\{A,C,G,T\}$, find the optimal superstring S, such that each f_i or its reverse complement, after a minimum number of mutations (insertion, substitution, or deletion of some nucleotide bases), is a substring of S. There can be multiple optimal superstrings for a unique collection of fragments.

4. Solution

In the past decade, a number of excellent DNA fragment assemblers emerged applying various algorithms. Some of the most well-known assemblers are Phrap[7], TIGR[8], CAP3[9], and EULER[10]. There are weaknesses even for the best assemblers – failing to handle repeats longer than fragment reads, generating too many contigs, assembling results shorter than the target sequence, and slow assembly speed. Aiming at improving these weaknesses, we have explored three types of algorithms – traditional "overlap–layout–consensus" algorithm, Euler algorithms, and genetic algorithms to solve the DNA Fragment Assembly problem.

4.1 Traditional Algorithms

Most DNA Fragment Assembly algorithms have three key modules: The Overlap module measures the degree of overlapping among fragments; The Layout module determines the blueprint to join fragments one after another according to the overlapping degrees among fragments; The Consensus module forms the consensus sequence according to the layout blueprint [11].

4.1.1 Overlap Measurement

Applying the traditional "overlap–layout–consensus" algorithm, we will first measure the feasibility of assembling every possible pair of the fragments with dynamic programming. There are four types of overlaps to consider, as illustrated in Figure 7.

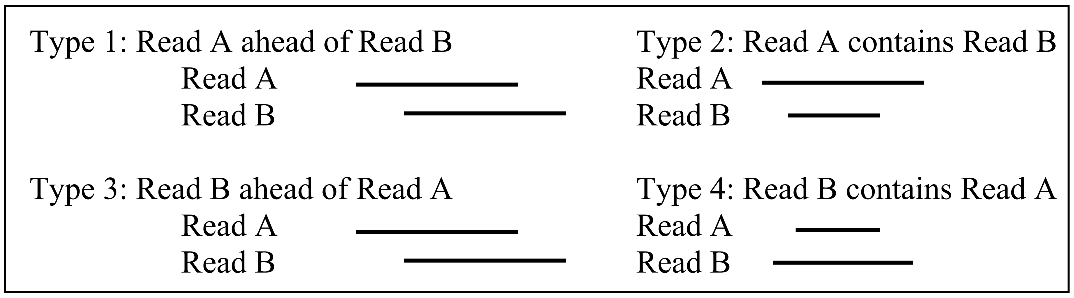
Type 1: Read A ahead of Read B
Read A
Read B
Type 2: Read A contains Read B
Read A
Read B
Type 3: Read B ahead of Read A
Read A
Read B
Type 4: Read B contains Read A
Read A
Read B

Figure 7: Four types of fragment overlapping

A commonly used dynamic programming algorithm for overlap pattern matching is String Alignment. String Alignment computes the similarity of two strings according to a predefined "alignment" function that provides a positive score on match but negative scores on insertion, deletion, and substitution. The final score for aligning two strings is deduced by gradually increasing the prefixes of the two strings and computing the scores of prefixes step by step. Scores in each step are reused in the next immediate step, so a matrix with the width of one string's length plus one, and the height of the other string's length plus one is required to hold the alignment scores of all prefixes of the two strings. A high alignment score indicates close similarity of two strings. The algorithm's complexity is O(nm), where n and m are the length of two strings respectively. In addition, it consumes O(nm) memory space due to the matrix caching prefixes alignment scores [12]. For k fragments inputs with average size m, the complexity for overlap pattern matching is $O(k^2m^2)$. Figure 8 illustrates the steps and functions used to align two strings ACGTCGTC and TCGTCTT.

i \ j	-	T	C	G	T	C	T	T
-	0	-2	-4	-6	-8	-10	-12	-14
A	-2	-1	-3	-5	-7	-9	-11	-13
C	-4	-3	0	-2	-4	-6	-8	-10
G	-6	-5	-2	1	-1	-3	-5	-7
T	-8	-5	-4	-1	2	0	-2	-4
C	-10	-7	-4	-3	0	3	1	-1
G	-12	-9	-6	-3	-2	1	2	0
T	-14	-11	-8	-5	-2	-1	2	3
C	-16	-13	-10	-7	-4	-1	0	1

Aligning S_1 = ACGTCGTC and S_2 = TCGTCTT

Dynamic string alignment $A(i, j) = \max \begin{cases} A(i-1, j-1) + v(S_1[i], S_2[j]) \\ A(i-1, j) + v(S_1[i], -) \\ A(i, j-1) + v(-, S_2[j]) \end{cases}$

The score function may be $v(S_1[i], S_2[j]) = \begin{cases} +1 & if \quad if\ S_1[i] = S_2[j] \\ -1 & if \quad if\ S_1[i] \neq S_2[j] \\ -2 & if \quad S_1[i] = -\ or\ S_2[j] = - \end{cases}$

We obtain:

```
      T C G T C – T T
      A C G T C G T C
Score -1+1+1+1+1-2+1-1=+1
```

Figure 8: A dynamic programming example to align two strings

Based on the dynamic programming algorithm for string alignment, a variety of improvements can be made: TIGR assembler computes the k-tuples in common between each pair of fragments; Myers combines suffix and indexes in sequence database for fast retrieval of similar fragments; Phrap and ARACHNE [13] use various flavors of common subsequence identification algorithms to rule out obviously unmatched fragment pairs before applying time consuming dynamic algorithms for detail alignments. AMASS represents fragments with multiple sub-string patterns called probes and performs pattern matching on probes rather than on the whole fragment to accomplish superior speed [14].

4.1.2 Layout Determination

According to the measurement result, we determine the basic layout of the fragments. This is the most challenging step because it is hindered by issues of errors, repeats, and insufficient coverage. Numerous creative methodologies, ranging from greedy algorithms to graph theory algorithm, have been applied to seek optimal solutions at this step.

The DNA Fragment Assembly problem can be reduced to Shortest Common Superstring Problem (SCS), in which case we attempt to find the shortest DNA sequence that contains all the DNA fragments. Gallant et al. proved that SCS is NP-complete [15]. In other words, we can only apply some heuristic methods to find a close to optimal solution in an acceptable amount of time. Greedy algorithm was firstly introduced by Staden in 1979 to iteratively assemble fragments with maximum overlaps to one DNA sequence [16]. The worst result for superstring computation is about twice as long as optimal superstring [17].

In reality, the shortest super string for fragment inputs is not the target DNA sequence, which we are looking for due to read errors, repeat regions, and orientation issues. A better model for DNA Fragment Assembly might be Sequence Reconstruction: for a set of fragments $f_i \in F$ with error rate $\varepsilon \in [0,1]$, find the shortest superstring S satisfying the condition – $\min\{d(S_{sub}, f_i), d(S_{sub}, f_i')\} \leq \varepsilon\, |S_{sub}|$, where fragment f_i' is the reverse complement of fragment f_i and $d(S_{sub}, f_i)$ is the minimum edit distance between subsequence of S and fragment f_i computed with dynamic programming similar to the string alignment algorithm. Sequence Reconstruction is also proven to be NP-complete [1]. A robust approach to tackle NP-Complete problems is genetic programming, which we elaborate in a later section.

In addition to string processing algorithms, graph theory might be an alternative to assist DNA Fragment Assembly. We can model DNA Fragment Assembly with weighted graphs where each vertex stands for a fragment and each edge stands for overlap between the two fragments. The Hamiltonian path that traverses each vertex once provides us with important insight for assembling the fragments. The graph of Figure 9 is a simplified illustration on reducing Fragment Assembly to the Traveling Salesman problem: for fragment set V = {GTG, TGA, GAT, ATG, TGC, GCT, CTG, TGG} finding the path sequence that visits each vertex once yields S=GTGATGCTGG, a minimum superstring for the fragment set. Although the Traveling Salesman Problem is NP-Complete [1], due to its long history in math and computer science there are many studies finding close to optimal solutions; thus, a lot of techniques tackling TSP can be revised to apply to DNA Fragment Assembly. Euler algorithms are innovative approaches that convert DNA Fragment Assembly to Euler Path Finding on a de Bruijn graph [9]. We have dedicated most of the research project to applying Euler graph theory on DNA Fragment Assembly.

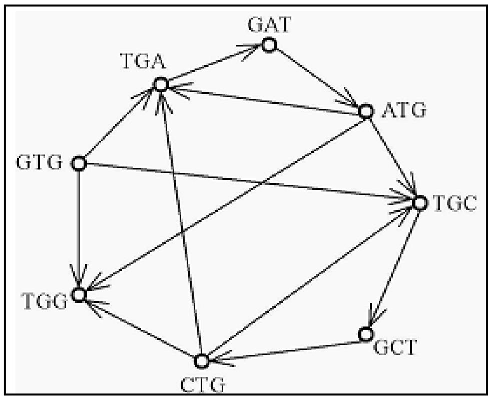

Figure 9: Hamiltonian path solution for DNA Fragment Assembly Problem [1]

4.1.3 Consensus Sequence Construction

Creating a consensus sequence is the final step. Resolving repeats intermixed with errors is the main issue challenging assembly algorithms at this stage. Phrap generates consensus sequence in a greedy approach according to its proprietary LLR-score order [7]. Phrap incorporates error probability to fragment alignment dynamic programming. In practice, errors in fragments are independent from each other, implying that fragments do not have errors at the same position of a sequence. In other words, given sufficient coverage, most fragment errors can be corrected by the majority rule. Celera Assembler masks repeats to avoid confusion and leaves repeats unassembled. Some repeats are already known from experience [1]. Those repeats are assembled based on the former understanding of repeats. TIGR, CAP3, and ARACHNE assemblers compute distance constraints between two ends of fragments by using the majority rule to assist pairwise ordering. Most assemblers cannot reconstruct fragments back to one unique consensus sequence at the end for sophisticated genome sequencing projects;

instead, several long contigs are returned for users to do the rest of the finishing work. Even for fragment inputs free of errors, most assemblers still make assembling errors: Phrap, CAP3, and TIGR assemblers make five, four, and two errors respectively when assembling error-free *Neisseria meningitidis*[1] genome fragments [11].

[1] *Neisseria meningitidis* is a kind of bacterium playing a role in meningitis.

4.2 Euler Algorithms

EULER algorithms for DNA Fragment Assembly, developed by Pevzner et al., completely abandon the traditional overlap-layout-consensus methodology. EULER algorithms are innovative in the sense that they cut the existing DNA fragments into even smaller pieces of regular size to transform the NP-Hard Fragment Layout issue to a polynomial time solvable Euler Path Discovery problem. Moreover, EULER algorithms surpass other DNA fragment assembly algorithms in error correction and repeat resolution – they can correct up to 97% of the errors and resolve all repeats that are not longer than fragment length. There are two main modules for Euler algorithms – Error Correction and Euler Superpath Resolution [18].

4.2.1 Error Statistics

Before we discuss error correction, we need to elaborate on the general error patterns in fragments to be assembled. Usually, the average error rate is known before fragment assembly and should be less than 10%. It is a common practice to discard fragments containing errors exceeding a certain error rate. Errors are independent of one another: different fragments covering the same range of a sequence have errors at different positions. For example, in Figure 10 three fragments cover AACTGCCTTAG while containing errors at different positions.

```
  CGTCAA?TGCCTTAGGCTA
ATCGTCAACTACCTTAG
       AACTGCC TAGGCTACA
```

Figure 10: Independent nature of fragment errors

Given average fragment length *m*, target sequence length *L*, and coverage *c*, there should be *cL/m* fragments in the fragment set. The possibility for one out of *L* positions to be randomly selected as the beginning of a fragment is *p* = *c/m*. For a position to be covered by a fragment, any of the inclusive *m* positions before the position can be selected as the starting position of the fragment. To have *x* coverage on a position, *x* of the *m* positions must be selected as the beginning of fragment. This infers a binomial distribution for the probability of the number of times that a position covered by fragment reads [2]:

$$\mathrm{P}\{\mathrm{C} = x\} = \binom{m}{x}(c/m)^x(1-c/m)^{m-x}$$

Equation 1 Possibility calculation for fragment coverage equal some number on a certain location

The possibility of a position covered by less than or equal to x fragments:

$$\mathrm{P}\{\mathrm{C} \le x\} = \sum_{k=0}^{x}\binom{m}{k}(c/m)^k(1-c/m)^{m-k}$$

Equation 2 Possibility calculation for fragment coverage no more than x on a target DNA sequence

For fragments with an average size larger than one hundred, we may approximate a binomial distribution using a normal distribution with expected value μ(*x*) = c and standard deviation $\sigma = \sqrt{m(c/m)(1-c/m)} = \sqrt{c(1-c/m)}$ according to the Central Limit Theorem. A key property of the normal distribution is that about 68% of the values should be in the range [μ-σ, μ+σ] and about 95% of the values should be in the range [μ-2σ, μ+2σ] according to Empirical Rule. For a normal distribution, the possibility of having a value 2σ less than the expected value is about 2.5%. Given a set of fragments with average length 800, coverage ten, the expected multiplicity (the number of occurrences in the fragment set) of a tuple is ten, and $\sigma = \sqrt{10(1-10/800)} \approx 3.4$. Thus, the possibility for a tuple to have multiplicity of three or less is about 2.5%. In other words, we are 97.5% confident that a tuple with multiplicity of three or less is caused by errors [19].

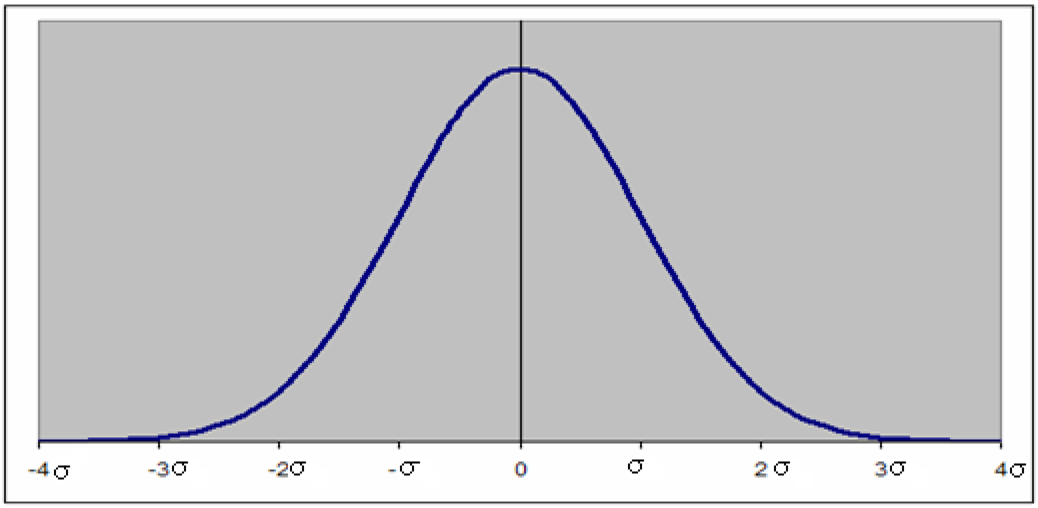

Figure 11: Normal distribution curve [19]

4.2.2 Error Correction

The Euler Algorithm exploits the fact that errors occur at different positions to perform error correction of the fragments. Euler Error Correction starts by chopping all fragments to much smaller tuples. For example, all possible 8-tuples of attcggctccgtgcttacatg is given by:

```
G_l = {
         attcggct
           ttcggctc
            tcggctcc
             cggctccg
               ggctccgt
                 gctccgtg
                   ctccgtgc
                     tccgtgct
                      ccgtgctt
                        cgtgctta
                          gtgcttac
                            tgcttaca
                             gcttacat
                               cttacatg
     }
```

Creation of tuples from fragments adopts a sliding window approach with the window width equal to the tuple size. The amount of tuples for a fragment set without errors can be associated linearly to the length of the target sequence (L) covered by the fragment set. Besides, there is an inverse relation between the proportion of repeats on the target sequence and the number of different tuples.

Due to high coverage, fragments overlap with each other. Normally each tuple appears in multiple fragments. We use multiplicity of a tuple to refer to the number of the tuple's occurrences in a fragment set. For a fragment set having coverage ten on the target sequence, the expected multiplicity of an ordinary tuple is ten. If there is no error, the multiplicity of an ordinary tuple is expected to be the same as the coverage. For tuples on repeat regions, the multiplicity can jump to two or more times of the coverage depending on the frequency of repeats. Due to the randomness of fragment creation, the possibility of a tuple with only multiplicity of two or less is very small.

We call the tuples with low multiplicity *weak* and the tuples with high multiplicity *solid*. Knowing that most weak tuples are caused by errors, we can then associate each weak tuple with one of the solid tuples and correct the errors accordingly. An error in a fragment usually causes l weak l-tuples and an additional l weak l-tuples in the reverse complement fragment. For an error located at d bases away from the fragment boundary, where $d < l$, there will be $2d$ weak tuples created by the error [20]. Figure 12 illustrates how an error results in weak tuples generated by the sliding window approach.

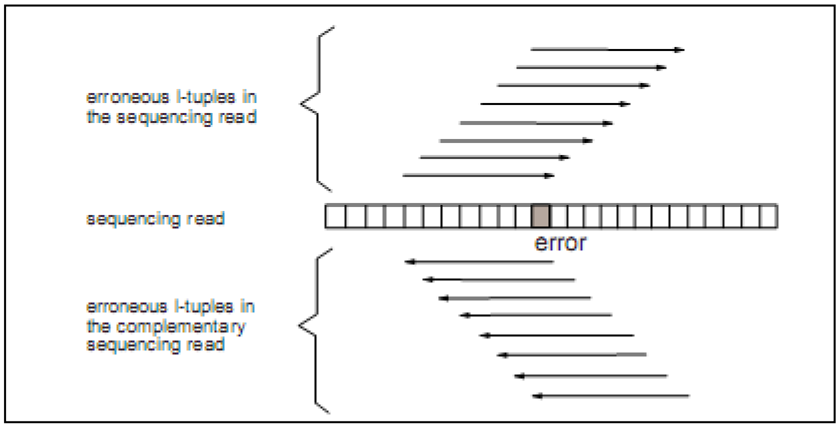
erroneous l-tuples in the sequencing read
sequencing read
error
erroneous l-tuples in the complementary sequencing read

Figure 12: Weak tuples generated by an error on a fragment and its reverse complement [18]

We define the relationship between two tuples as *neighbors* if we can change one to another with one mutation. We call a tuple *orphan* if the tuple meets the following three conditions:

(i) Multiplicity smaller than a pre-set threshold. For our former example of a fragment set with average size 800 bases and coverage ten, we may set the threshold to three to ensure the 97.5% confidence on error detection.

(ii) The tuple has a unique neighbor.

(iii) The tuple's neighbor is solid. The process of error correction consists in changing an orphan to its unique solid neighbor.

The example below demonstrates the steps of substitution error correction. In a fragment set that has ten fragments covering the region ggctccgtgctt, one fragment has an error at the fourth position changing base t to c. The rest nine fragments that are correct in the region will generate solid tuples with a multiplicity of nine on the left, while the fragment with error at the fourth position might create weak tuples on the right. By mutating the orphans to their corresponding solid neighbor, we correct the error in the weak tuples as well as the fragment. The fragments that are correct in one region might have errors in other regions. On the other hand, the fragment that has error in one region might be correct in other region. Taking advantage of the independent nature of fragment errors, Euler assembler can correct errors by majority rule.

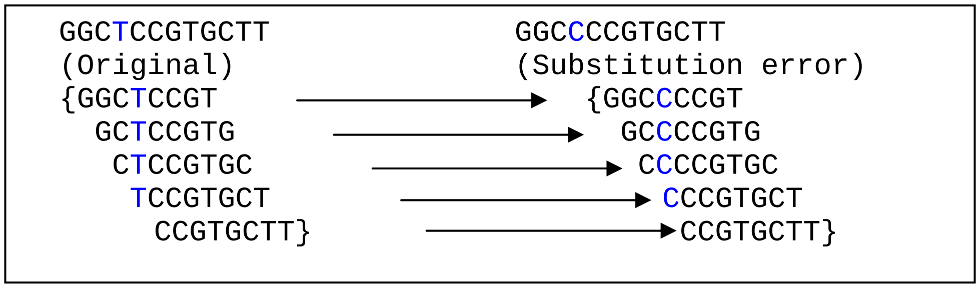

Figure 13: Associating weak tuples with their solid neighbor to correct a substitution error

Correcting insertion and deletion errors is slightly more complicated than correcting substitution errors. In Figure 14, the fragment on the right has an insertion error at the fifth position. Such an error causes a series of weak tuples without neighbors except the last one. Paying attention to this special pattern helps us detect and correct insertion errors.

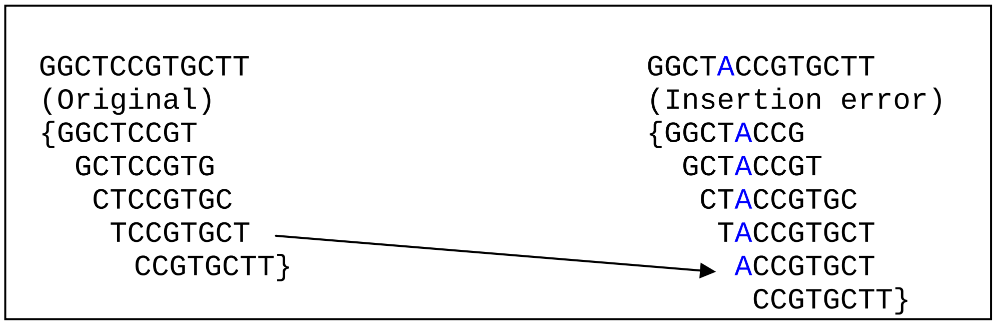

Figure 14: Associating a weak tuple with a solid neighbor to correct an insertion error

A similar method can be applied for correcting deletion error.

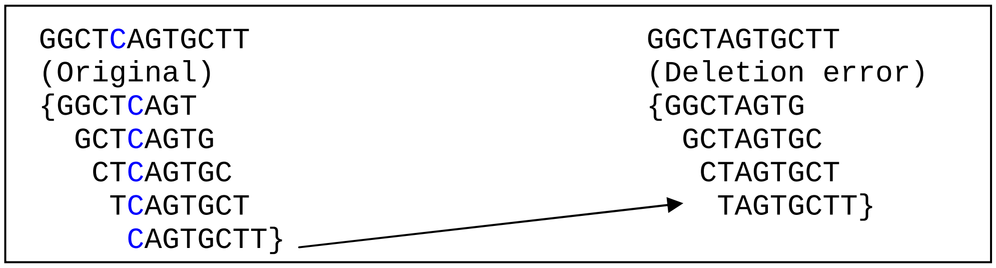

Figure 15: Associating a weak tuple with a solid neighbor to correct a deletion error

4.2.3 Correction Limitation

The selection of *l* value or tuple size depends on several factors: appearance of short local repeats, distance between errors, and runtime efficiency for neighbor discovery. If the tuple size falls close to the size of local repeats, we might encounter many weak tuples with more than one neighbor because one copy of repeats might differ slightly from another copy. This issue will confuse the assembly algorithm for the right way to correct the error. If the distance of two errors in one fragment is smaller than *l*, some weak tuples caused by the errors cannot be associated with a solid neighbor that is one mutation away; the complexity of screening a tuple's neighbor is O(*lG*) where *l* is the tuple size and *G* is the target sequence length. Given a large *G* for a complicated assembly project, a slight increment of tuple size causes the screening runtime to become significantly longer.

The Euler Error Correction methodology can incorrectly change correct fragments on the low coverage range of the target sequence. The Euler Assembler uses parameter Δ, defining the maximum number of errors in a fragment, as a threshold to prevent the Euler Error Correction removing the difference in repeats [20]. However, this can cause the situation in Figure 16 to be overlooked. In Figure 16, a fragment covering the key information connecting two closely spaced contigs is prone to be eliminated in insertion error correction.

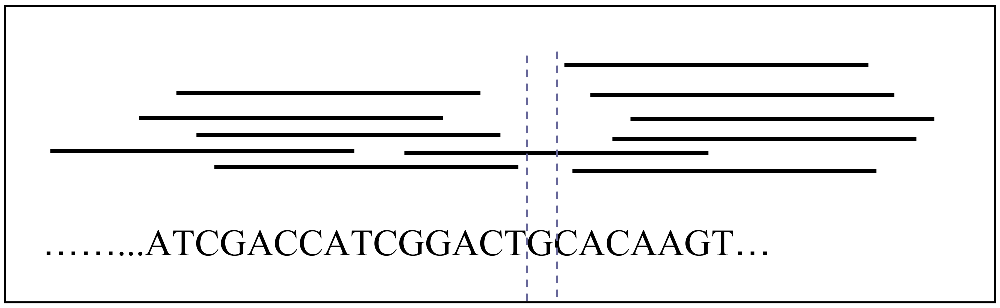

Figure 16: Low coverage on a position due to random fragment generation

The possibility of observing only one coverage at a base position for a fragment set with average length five hundred and coverage eight, is

$P\{C=1\}=\binom{500}{1}(8/500)^{1}(1-8/500)^{500-1} \approx 2.25 \cdot 10^{-3}$. The possibility of observing *x* bases with one coverage consecutively is $(2.25 \cdot 10^{-3})^{x}$, which decreases exponentially as x increases. Thus, most of the 2% correct fragment reads with low coverage are on individual base positions situated randomly across the target sequence. The threshold parameter Δ can hardly protect them from false Error Correction. The negative effect caused by the erroneous correction is that more contigs appear at the end of assembly.

In spite of the drawbacks of introducing small amount of new errors, Euler Error Correction is verified to be a superior method for error elimination in practical sequencing projects. In the case of *Neisseria meningitidis* fragment assembly, 234,410 errors were corrected with the side effect of 1,452 new errors. Differentiating tuples of multiplicity less than three as orphans is effective for error detection based on experiments [18].

4.2.4 Euler Superpath

Given a set of *l*-tuples $S_l = \{s_1,...,s_n\}$, we can construct a de Bruijn graph by using each tuple in set S_l as a directed edge. This edge runs from the first (*l*-1) tuple of the *l*-tuple as source vertex to the last *l*-1 tuple as destination vertex. More precisely, for each *l*-tuple as an edge in the de Bruijn graph, we define two *l*-1 tuples as vertices, the first *l*-1 nucleotide string as the source and the last *l*-1 nucleotide string as the destination. For sequence ATGCTTGCGTGCA, if the edges are 3-tuples, the vertices will be all the 2-tuples. Edge set S_l={ATG, TGC, GCT, CTT, TTG, GCG, CGT, GTG, GCA}. Vertex set S_{l-1}={AT,TG,GC,CT,TT,CG,GT,CA}. Consequently, we have the de Bruijn graph as illustrated by Figure 17.

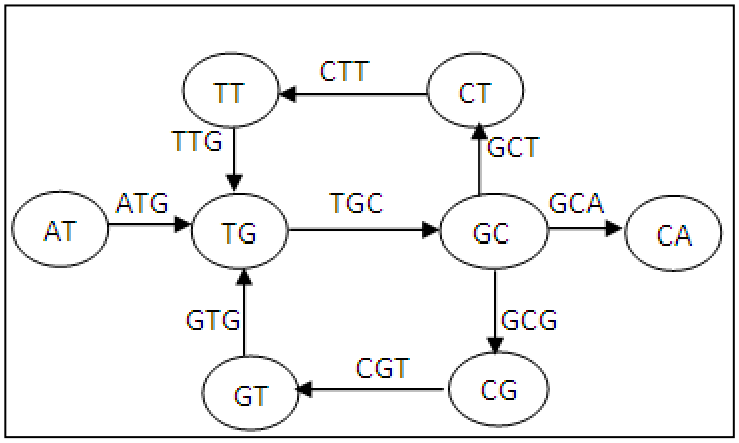

Figure 17: de Bruijn graph of the DNA sequence ATGCTTGCGTGCA

The problem of finding the consensus sequence for DNA fragment assembly is converted to the problem of looking for an Euler path that traverses the edges in a de Bruijn graph. Euler paths can be found in polynomial time of the number of edges. However, there is one more condition for the Euler path of consensus sequence to satisfy: the Euler path must contain all the fragment reads as subpaths. Such an Euler path is called Euler Superpath. The Euler Assembler developed by Pezner et al. performs graph system transformation to achieve the goal of finding the Euler Superpath according to the rule that two graph systems, identified by a graph and a path set, are equivalent if there is a one to one correspondence between graph G and path set P of the first system, and graph G' and path set P' of the second system. Through a series of transformations $(G,P) \rightarrow (G_1,P_1) \rightarrow \ldots \rightarrow (G_k,P_k)$, a new graph system is achieved where every edge in the graph G_k is a path in the path set P_k. As a result, finding the Euler path in the last graph system is the same as finding the Euler Superpath in the original de Bruijn graph system [18]. The following rules are applied to ensure equivalent transformations.

Direct Transformation: Let x = (v_{in}, v_{mid}) and y = (v_{mid}, v_{out}) be two adjacent edges in graph G and let $P_{x,y}$ be the set of all paths that include edges x and y. A new edge z =(v_{in},v_{out}) can be used to replace edges x and y in the graph G as well as the path set P, resulting in an equivalent system graph G_1 and path set P_1. In Figure 18, $P_{\rightarrow x}$ stands for the set of all paths that end at edge x; $P_{y\rightarrow}$ stands for the set of all paths that start with edge y; $P_{x,y}$ stands for the set of all paths that traverse through edge x and edge y.

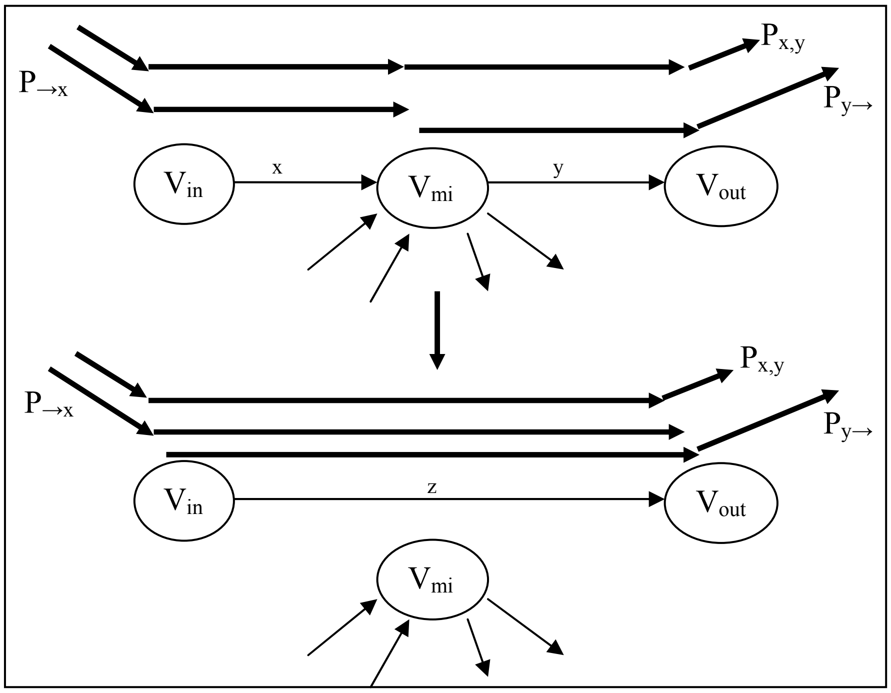

Figure 18: Replacing edges x and y with z by Direct Transformation

Branch Transformation:

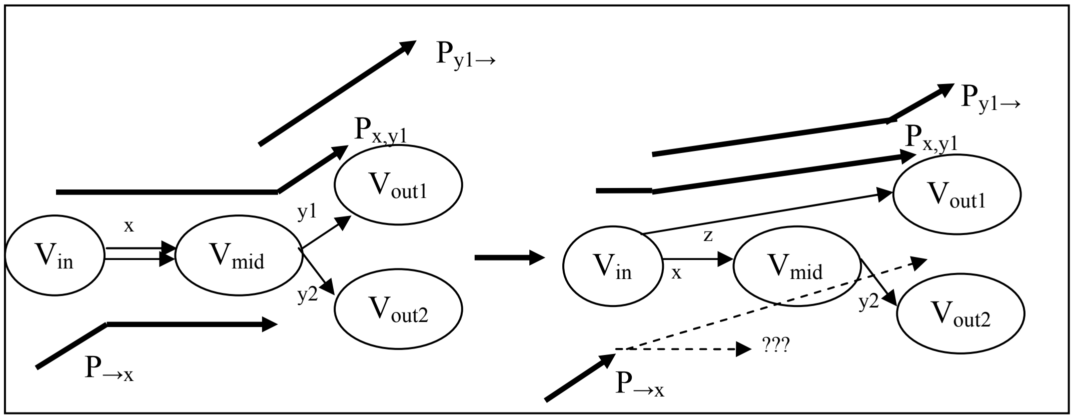

Figure 19: Path subset consistency to determine path replacement by Branch Transformation

When there is one incoming edge x but two outgoing edges y_1 and y_2 from v_{mid} to v_{out1} and $v_{out2,}$ we cannot replace x in every path ending at x with $z = (v_{in}, v_{out1})$ as an equivalent transformation. Instead, we must first define path subset $P_{x,y1}$ as all paths containing edge x and y_1 and path subset $P_{x,y2}$ as all paths containing edge x and y_2. Whether to replace x with z or not for a Path $p_{\to x}$ ending at x depends on whether Path $p_{\to x}$ is consistent with subset $P_{x,y1}$ or subset $P_{x,y2}$. Two paths are consistent with one another if they can be joined together without generating a branch. Path 2 is consistent with Path 3 but inconsistent with Path 1 due to the branch at vertex v in Figure 20.

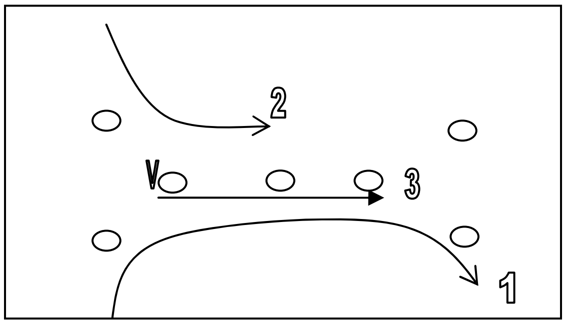

Figure 20: Path 2 consistent with Path 3 but inconsistent with Path 1

A path $p_{\rightarrow x}$ is consistent with path set $P_{x,y1}$ if $p_{\rightarrow x}$ is consistent with every path in $P_{x,y1}$ and then we can replace edge x in path $p_{\rightarrow x}$ with z. There are three possible results for the consistency check on two branches:

1. $p_{\rightarrow x}$ consistent with either $P_{x,y1}$ or $P_{x,y2}$
2. $p_{\rightarrow x}$ consistent with both $P_{x,y1}$ and $P_{x,y2}$
3. $p_{\rightarrow x}$ consistent with neither $P_{x,y1}$ nor $P_{x,y2}$

Result 1 allows us to relate $p_{\rightarrow x}$ to either $P_{x,y1}$ or $P_{x,y2}$ for an equivalent transformation. Result 2 indicates that path $p_{\rightarrow x}$ does not provide us any valuable information for assembling unless we can extend $p_{\rightarrow x}$ with another path(s) so that the new path can be related to either $P_{x,y1}$ or $P_{x,y2}$. Result 3 indicates that there is an error in $P_{\rightarrow x}$ that should be corrected.

In Figure 21, Path 2 is too short to tell us anything valuable since it is consistent with both Path 1 and Path 3, unless we can merge Path 4 with Path 2 to achieve the green dash Path 5 that is consistent with only Path 3.

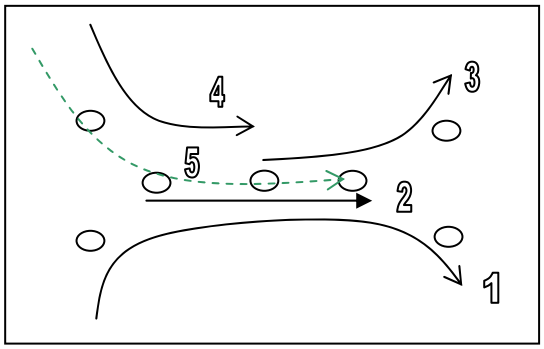

Figure 21: Associating Path 2, 3, and 4 to determine path consistency

In Figure 22, due to an insertion error in fragment read #3, Path3 representing the fragment cannot be related to either Path 1 or Path 2.

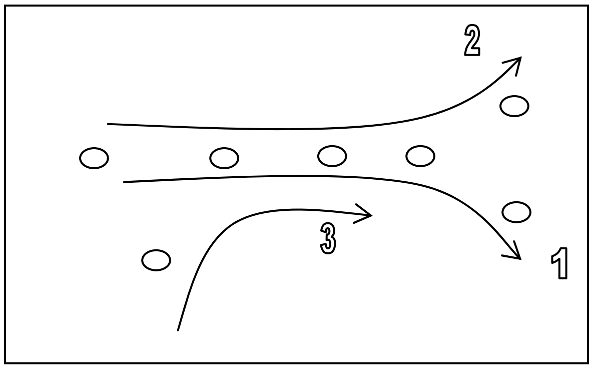

Figure 22: Fragment error leads to inconsistent Path 3

Compared to traditional pairwise overlap method, path system transformation is a powerful method to resolve repeats for fragment assembly: for a target sequence AGTTATCGCGCGAACTAAGGCC covered by three fragments ATCGCGCGAA, AGTTATCGCG, CGCGAACTAAGGCC, the traditional method might assemble AGTTATCGCG and CGCGAACTAAGGCC first with a greedy approach to get AGTTATCGCGAACTAAGGCC and we lose the subsequence ATCGCGCGAA which contains three occurrences (and not two) of CG. Alternatively, the initial graph system for a de Bruijn graph with 5-tuples edge and three fragment reads can be illustrated by Figure 23:

$$\begin{tikzcd}
\text{AGTT} \arrow[r] & \text{GTTA} \arrow[r] & \text{TTAT} \arrow[r] & \text{TATC} \arrow[r] & \text{ATCG} \arrow[r] & \text{TCGC} \arrow[r] & \text{CGCG} \arrow[r, shift left] \arrow[dll] & \text{GCGC} \arrow[l, shift left] & & & & & \\
 & & & & \text{GCGA} \arrow[r] & \text{CGAA} \arrow[r] & \text{GAAC} \arrow[r] & \text{AACT} \arrow[r] & \text{ACTA} \arrow[r] & \text{CTAA} \arrow[r] & \text{TAAG} \arrow[r] & \text{AAGG} \arrow[r] & \text{AGGC}
\end{tikzcd}$$

Figure 23: de Bruijn graph generated by the three fragments containing repeats

Most transformations are straightforward until there is a branch selection in front of CGCG where it can move forward to GCGA or GCGC. We have an equivalent graph system given in Figure 24:

$$\begin{tikzcd}
\text{AGTTATCGCG} \arrow[r, shift left] \arrow[d] & \text{GCGC} \arrow[l, shift left] & & & & & & & & \\
\text{GCGA} \arrow[r] & \text{CGAA} \arrow[r] & \text{GAAC} \arrow[r] & \text{AACT} \arrow[r] & \text{ACTA} \arrow[r] & \text{CTAA} \arrow[r] & \text{TAAG} \arrow[r] & \text{AAGG} \arrow[r] & \text{AGGC} \arrow[r] & \text{GGCC}
\end{tikzcd}$$

Figure 24: Direct Transformation before encountering branches at the repeat location

In addition, the path AGTTATCGCG is consistent with both ATCGCGCGAA and CGCGAACTAAGGCC, until we merge ATCGCGCGAA and CGCGAACTAAGGCC to be ATCGCGCGAACTAAGGCC to resolve the double edge between de Bruijn vertices CGCG and GCGC:

AGTTATCGCG→ATCGCGCGAACTAAGGCC

Figure 25: Delayed Branch Transformation resolves the repeats

Through this process, we get the final sequence: AGTTATCGCGCGAACTAAGGCC

4.3 Proposing Improvements on Euler Algorithms

In our approach, unlike the Euler Assembler that starts with error correction, our algorithm performs error correction on the fly when needed during the traversal of the graph. In other words, we postpone error correction and perform it only in need to achieve better runtime efficiency. Our algorithm also tackles very long repeats, which is achieved by incorporating statistical analysis in the Euler path traversals.

4.3.1 Traversal Approach for Euler Superpath

Given a set of fragments, we would like to reconstruct the target DNA. As is done in the Euler Assembler, we construct all 20-tuples from all the fragments. Conceptually, we have a graph whose edges are the 20-tuple sequences and the vertices are the 19-tuple sequences.

Instead of using the graph system transformation approach to discover the Euler Superpath, we traverse the de Bruijn graph according to fragment reads so that the traversed path contains all fragment reads as subpaths. The steps to discover the Euler Superpath are as follows:

1. Start traversal from a vertex v that does not have a predecessor; a regular expression checks edges to find a list of such vertices. Each vertex with no predecessor represents the beginning of a contig or an island in the de Bruijn graph. Traversal continues until encountering a vertex that has more than one outgoing edges available (branches) or a vertex that has no way out (end of the contig). Due to repeats, an edge can be traversed multiple times.
2. There can be at most four choices in front of a branch vertex – A, T, C, and G. When we encounter branches, we retrieve the fragment path subsets for each branch, which consist of all fragments for each of the choices. We relate our traversed path to branch path subsets and select the option whose path subset is consistent with our traversed path.

   For example, given three fragments for coverage one without error:

   (1) AAGACGTAGA
   (2) CTGACA
   (3) CGTAGACT
   We can construct the de Bruijn graph shown Figure 26.

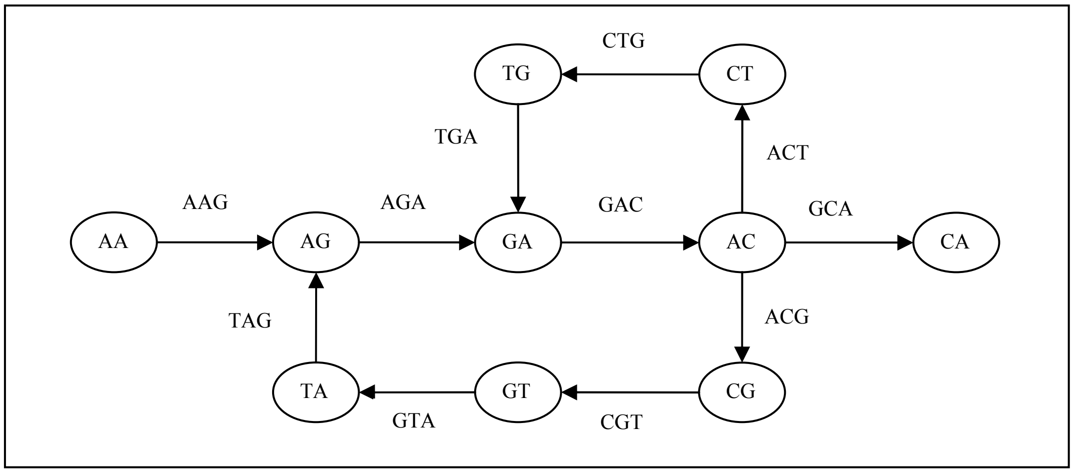

Figure 26: de Bruijn graph for target sequence AAGACGTAGACTGACA

We start the traversal with vertex AA and reach vertex AC, which corresponds to sequence AAGAC. From vertex AC we have three choices. At this point we compare AAGAC to all three fragments. Because AAGAC is the prefix of the first fragment, which is the one we choose. The next character in the first fragment is G, consequently, the traversal now visits vertex CG. Traversal continues until we reach the ending node CA and we get the target sequence AAGACGTAGACTGACA.

3. Allowing traversal of an edge more than once can make the traversal end up in an infinite circle. An ending circle in de Bruijn graph represents an ending repeat for a sequence. To find out how many times we want to traverse the circle, we resort to the fragment that contains the largest amount of repeats, or better still the statistical analysis approach discussed later in this paper.

As we traverse the graph of Figure 27 and encounter AG for the second time, we know it might be an ending cycle. Then we query all the fragments containing the AG node and compare the fragments with existing traversed paths to get the longest extension from the fragments. For example, we have AAGACGTA and if the longest fragment extension for AG repeat is GACGTAAGACGTAAGACGTA; by comparing our traversed path with the fragment we know the ending cycle must be traversed one more time.

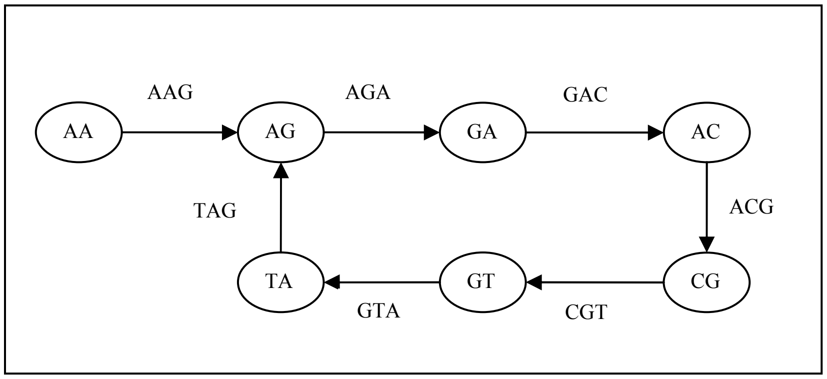

Figure 27: Same de Bruijn graph for sequences GACGTAAGACGTA and GACGTAAGACGTAAGACGTA

4.3.2 Statistical Analysis for Perfect Long Repeat Assembly

Perfect long repeats are identical repeats longer than maximum fragment reads (~1000 b). Using the traditional fragment overlapping approach or the Euler graph system transformation for the Euler Superpath cannot restore them because all copies of repeats will be collapsed to one.

Statistical analysis can provide us valuable insight to restore the target sequence containing perfect long repeats. In the error correction section, we calculated that the possibility for a tuple to have a multiplicity of three or less is about 2.5%, given a set of fragments with average length 800 and coverage ten. Because the normal distribution for binomial statistic approximation is a symmetrical bell shape, we can deduce that under the same circumstances, the probability for a normal tuple to have a multiplicity of eighteen or more, which is 2σ larger than the expected value, is also about 2.5%. In other words, there is a 97.5% probability that the tuple is a two-fold repeat. For a tuple on a two-fold repeat, the expected multiplicity is twenty since the coverage is twenty due to the repeat and standard deviation $\sigma = \sqrt{20(1-20/800)} \approx 4.5$. Therefore for a tuple of multiplicity over twenty five, we are 84% sure that it is a three-fold repeat [19]. Applying the same rule, tuples with multiplicity over thirty six and forty seven are likely to be a four-fold and five-fold repeats respectively. Based on the statistics, we tag each edge in the de Bruijn graph with a "best traversal amount". It would be best for our Euler Superpath to traverse an edge as many times as the "best traversal amount" tagged to the edge. There is a two-fold long perfect repeat intermixed with short repeats in the following target sequence: AA GACGTAGACT GACGTAGACT GACA. Given sufficiently random fragment coverage, we should have the de Bruijn table augmented with a "best traversal amount" for each edge as shown in Figure 28:

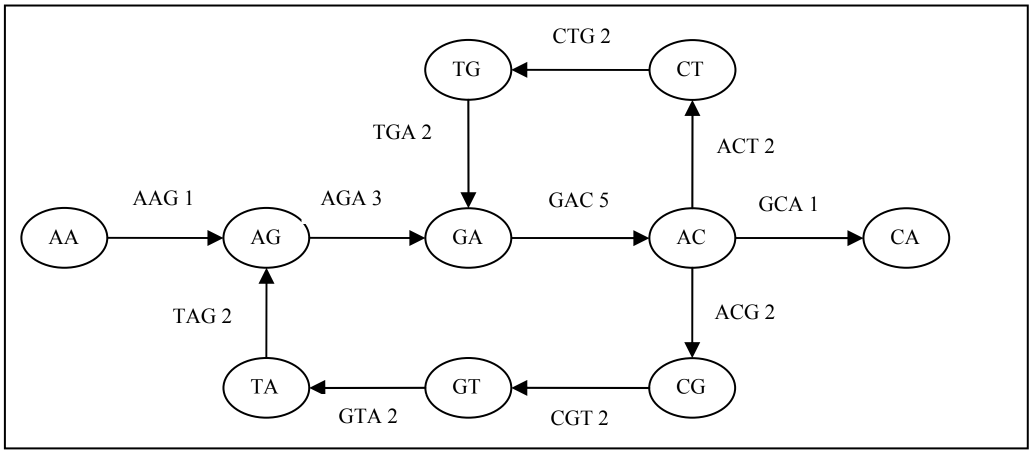

Figure 28: de Bruijn graph supplemented with "best traversal amount" by statistical analysis

Our Euler Assembler can traverse the de Bruijn graph containing long perfect repeats by applying the following steps:

1. Start traversal from a vertex v that does not have a predecessor and count the number of traversals on each edge
2. Resolve branch confusion according to consistency among traversed path and fragment subset on each branch. Long perfect repeats have exactly the same long repeats, so there should be no change on fragment subset consistency.
3. If there is more than one choice consistent with existing traversed path at a branch due to the confusion of perfect long repeat, randomly select one choice to continue traversal.
4. Stop the traversal at a node that has no edge out or the outgoing edges have been traversed "best traversal amount" of times.
5. For edges not traversed by sufficient amount of times according to "best traversal amount", backtrack to the nearest branch and redo branch selection to traverse those edges to meet "best traversal amount" requirement.
6. Resolving ending perfect long repeat is straightforward – traversing edges in the cycle up to "best traversal amount" of times. The target sequence for the de Bruijn graph in Figure 29 is AAGACGTAGACGTAGAC for a two and a half folds perfect long ending repeat.

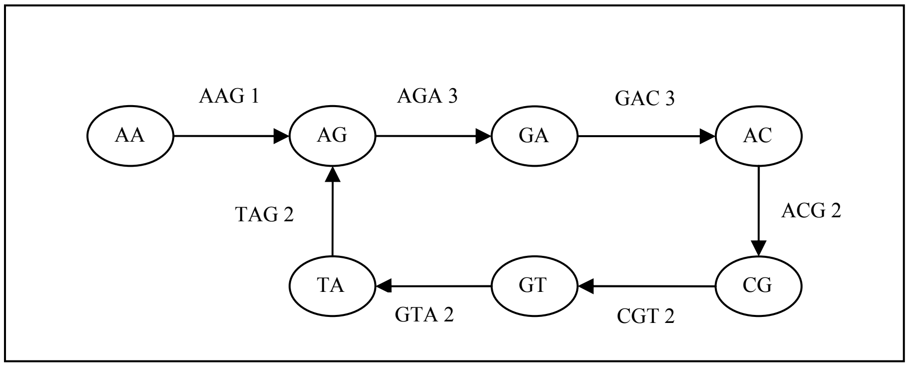

Figure 29: de Bruijn with statistical data uniquely determine target sequence

For the example in Figure 28, if the traversal by mistake goes by AAGACA, our algorithm will backtrack to the last branch, at AAGAC, to redo the decision, because a few edges are unvisited. The existing traversal on the last node CA does not need to be abandoned, because the traversal still needs to go to CA after satisfying the "best traversal amount" for some other edges. As a result, we can still get the following target sequence containing perfect long repeat.

AA GAC|GTAGACT GACGTAGACT GAC|A[2]

The target DNA sequence, AAGACGTAGACTGACAGACTGACC has more complicated long repeats with slight differences among them, so the order of long repeat matters. The fragment reads do not give us any clues on which one goes first, because each of the repeats is longer than any fragment. In the Genetic Algorithm section, we explore the problem of deciding the order of long repeats with slight

[2] Note: GACGTAGACT is long repeat not covered by any fragment, but GACGTA ahead of GACT should hopefully be covered by some fragments.

differences, which is a common challenge for assembling target sequence containing ALU repeats [8].

4.3.3 Postponed Error Correction

An important reason to perform error correction before assembly is that errors will cause large quantity of false tuples that are translated to edges in a de Bruijn graph. These false edges interfere with the Euler Superpath discovery at the assembly step. For a sequencing project of 20% repeat rate, 4% error rate, using tuple size of twenty, the quantity of solid and weak tuples are comparable to each other. Consequently, finding an Euler path for the de Bruijn graph will be a serious problem because of the large amount of false edges.

However, error correction can introduce new errors as we have discussed previously. Also, the process of finding a unique neighbor for an orphan is time consuming. Despite of the up to 97% success rate correcting error, we suggest postponing the error correction process until it is necessary. We can drop all weak tuples from the de Bruijn graph to attain a graph that is as clear as the graph after error correction. Essentially for de Bruijn graph, Euler Error Correction removes all weak tuples and increases the multiplicity of relevant solid tuples by one. For solid tuples, increasing multiplicity by one does not make sense. For graph system transformation, the resolution of each edge requires fragment set consistency check. This is another reason to have an almost error free fragment set, because fragments need to be retrieved frequently at every transformation. On the other hand, with the new traversal approach, the only time for fragments referencing is to decide which branch to continue traversal. We can safely ignore the potential errors implied by low multiplicity until the traversal encounters two or more branches. Solving errors at location near branches are safer, because branches imply repeats. In Figure 17, edge TGC is in front of three branches, indicating TGC will be repeated three times. The fragment coverage at repeat region is twice or more than the average coverage, though we still need threshold parameter Δ to retain the difference with repeat with low coverage. Statistically, 2.5% of the weak tuples are from the low coverage region due to random fragment generation. By limiting our error correction only to fragments necessary for traversal, we protect fragments of low coverage

4.4 Existing GA for DNA Fragment Assembly

Genetic algorithms are heuristic techniques that can be used to tackle the DNA Fragment Assembly problem. General steps applying genetic algorithm are as following:

1. The algorithm randomly generates a pool of solutions.
2. It screens for superior solutions with a fitness function.
3. Mutation and crossover operations are performed on good solutions to create next generation solutions.

It is believed that solutions evolve better for the DNA Fragment Assembly problem from one generation to the next. Having a random initial population, an appropriate fitness function, and suitable mutation and crossover operations allow the genetic algorithm to converge to good solutions for the DNA Fragment Assembly problem.

Each fragment is represented by a number or ID. A solution for fragment assembly is represented by a permutation of the fragment number. The fitness function in Equation 3 was used for most genetic algorithms for DNA fragment assembly [21]:

$$F1(I) = \sum_{i=0}^{n-2} w_{f[i],f[i+1]} \quad \text{(Equation 3)}$$

where $w_{f[i],f[i+1]}$ is the pairwise overlap strength of fragment i and i+1. Overlap strength can be computed with dynamic programming for minimum edit distance, string alignment, or shortest common superstring of the two fragments. The obvious problems for the fitness function are errors and repeats. Moreover, a pair with best overlapping scores might not be a pair contributing to the assembly most: for example fragment pair

ATTGCTCGCT and *TGCTCGCTAA* scores better than fragment pair *ATTGCTCGCT* and *TCGCTAACCGTA,* but the former pair indeed is closer to the optimal fragment assembly solution. Due to its efficiency and adequacy, this fitness function continues to be used. It takes O(n) time to evaluate each individual solution for fragment set of size n [22].

In Equation 4, the fitness function extends the previous fitness function by adding a penalty to solutions that separate overlapping fragments distantly. The cost of the addition increases the complexity to O(n^2) for each solution evaluation, while it still fails to address the fundamental loopholes mentioned previously.

$$F2(I) = \sum\sum |i-j| * w_{f[i],f[j]} \quad \text{(Equation 4)}$$

Two types of operations are used to evolve solutions from one generation to another – Crossover and Mutation.

For crossover, the genetic algorithm exchanges small portions between two solutions to encourage good partial solutions to flourish in different individuals. The example in Figure 30 illustrates order crossover for two assembly solutions at seventh position. The first two positions are changed correspondingly since Fragment 7 is moved to the crossover section; Fragment 1 is pushed up and Fragment 5 is added to the front.

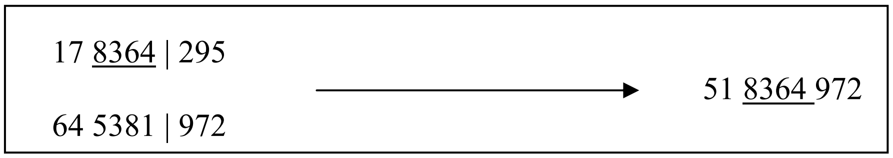

Figure 30: Crossover for two solutions of DNA fragment assembly

Edge-recombination crossover better suits DNA Fragment assembly by preserving valuable adjacencies. Given the same two parent solutions above, we can deduce an adjacency list for edge-recombination:

Fragment	Adjacent Fragments
1	7,8,9
2	4,9,7
3	8,6,5,8
4	6,2,6,5
5	9,4,3
6	3,4,4
7	1,8,9,2
8	7,3,3,1
9	2,5,1,7

Figure 31: Edge-recombination crossover to preserve fragment adjacency

The recombination starts from the Fragment 1, and then takes Fragment 8 due to its shared adjacency Fragment 3. Following Fragment 3 is Fragment 6 for shared adjacency Fragment 4. Fragment 2 is the next to select because it has more unselected adjacency. Applying the same rule the rest of the solution is Fragment 9, Fragment 7, and lastly Fragment 5. 18364975 is the resulting edge-recombination crossover child. For fragments with equivalent qualification during crossover process, arbitrary selection can be made. To explore nearby search space, the mutation performs elemental changes on an individual solution. Some previously eliminated solutions could be restored to the solution pool to contribute to further solution generation. Three kinds of mutations can be applied for fragment assembly solutions:

1. Swapping fragment number at two random positions in a solution. For example, swapping the first and the last fragments for solution 18364975 produces 58364971.
2. Swapping two adjacent fragments in a solution to achieve better fitness score.
3. Randomly selecting two fragments and moving one next to another for a better fitness score [23].

4.5 New GA for Fragment Coverage Restoration

We would like to propose a new genetic algorithm aiming at restoring all *m* layers of fragments given a fragment set with coverage *m*. Our genetic algorithm completely abandons the overlapping method to connect two or more fragments together. Instead, we think fragments should be concatenated to one another, because during random fragment creation multiple copies of the target sequence are randomly cut to fragments without any overlapping among fragments from the same copy.

4.5.1 Solution Generation

For a fragment set of size *n* and coverage *m*, a quick solution is to retrieve *n/m* fragments from the fragment set and concatenate the fragments in a certain order to form a superstring. Given a fragment set, we should be able to know the total nucleotide bases in it as well as the coverage. We can estimate the target sequence length by dividing the total nucleotide bases by the coverage. The length of a proposed sequence solution should have less than 10% difference from the estimated length.

The solution generation does not need to be random: we can start with a long fragment because we want to select a good successor and predecessor. For a fragment set with high coverage especially at repeat region, we might frequently have to decide which fragment to choose as next successor or predecessor from several candidates while only one is from the same layer as the current fragment. Also, we need to take into consideration that some nucleotide bases might be missing.

```
       ATCGGACTGACACACACAGCCTTAGGACTCG
CGGTCAGATCGGACTGACACA
(current fragment)   CACAGCCTTAGGACTCG (alternatives)
                     CAGCCTTAGGACTCG
                     CACACAGCCTTAGGACTCG
                           GCCTTAGGACTCG
```

Figure 32: Solution generations by hints from overlapping fragments

4.5.2 Fitness Function

The fitness function lays fragments on a specific position of the proposed solution to restore all layers of coverage. Repeats covered by fragments read will be resolved during fragment layout determination, because only one layout is correct to restore all fragments layers covering the repeat region. The complexity is still O(n) for individual solution fitness evaluation because all it does is to find the suitable position on the solution superstring according to pattern matching between solid tuple representations for fragments and a proposed sequence solution. The fitness function is capable of dealing with most errors and repeats. Meanwhile, the fitness function rewards good partial solutions to approach fragment assembly solution quickly.

Several merits of the innovative fitness function deserve further elaboration: the goal of the fitness function is to restore the original *m* coverage; laying out a fragment according to a proposed solution is to find a location to place the fragment on the proposed solution. If the proposed solution is correct, all fragments are placed in the correct location on the solution. Consequently, the fitness function must be able to restore all *m* layers of fragments covering the target sequence. For each of the *m* layers, there is no overlapping at all; thus, during restoration of coverage, no fragment are overlapped –

we only permutate the fragments to form layers of coverage. The same rule applies to the creation of the solution superstring; the new fitness function here is even capable of solving long perfect repeats.

4.5.3 Operations

Any operation must maintain the integrity of fragments. Because each solution is only using a subset of all available fragments, two solutions for crossover might be totally different from one another, which makes exchanging small portions among solutions straightforward.

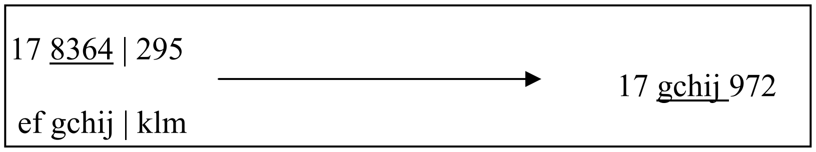

Figure 33: Crossover on solutions not sharing fragments

For two solutions sharing the same fragments affected by the crossover, the affected position(s) should perform the crossover too.

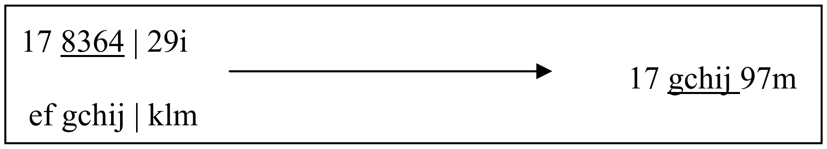

Figure 34: Crossover on solutions sharing fragments

Some regions of the solution might be capable of restoring all *m* layers of fragment, so the partial solution of those regions should certainly be retained with Edge-recombination crossover. During fitness measurement, we can use the adjacency between two fragments according to the number of fragment layers that the fitness function can restore on top of the region.

Six kinds of mutations can be applied for fragment assembly solutions:

1. Swapping fragments at two random positions in a solution.
2. Swapping two adjacent fragments in a solution to achieve better layer restoration.
3. Randomly selecting two fragments and moving one next to the other for a better fitness score.
4. Replacing a certain fragment in the solution with another fragment in the fragment set.
5. Replacing a certain fragment with its reverse complement.

The frequency of performing crossover and mutation are controlled by the crossover rate and the mutation rate, respectively. Setting a high rate for these operations might overlook good solutions distancing the final regression of genetic algorithm away from optimal solution. On the contrary, low operation rates slow down the progress of a genetic algorithm towards optimal solution leading to runtime inefficiency. A wise design for genetic algorithm is setting variable operation rates – a high rate at the beginning when solutions are far from being optimal, and a low rate at the end to fine tune final solutions.

4.6 Combining Euler and Genetic Algorithms

Using the genetic algorithm alone to figure out the blueprint of target sequence to restore all layers of coverage can be time consuming. On the other hand, Euler algorithms assemble most of the fragments correctly in polynomial time, though they cannot determine the order of long perfect repeats with slight differences. We can base our genetic algorithm's coverage restoration on de Bruijn graph. Combining these two algorithms can help us achieve more efficient runtime and assembly results that are closer to optimal.

Our Euler Genetic Hybrid algorithm starts from generating a de Bruijn graph with all solid tuples from the fragment set including original fragments and their reverse complements. Then, statistical analysis is performed on tuple multiplicity to drop the tuples with low multiplicity. Next, path traversal starts from the head of each contig, which is a head tuple of a fragment with no predecessor in the de Bruijn graph. Coverage restoration begins where Euler Path Traversal encounters branches or more than one option to continue. Traversal or the assembly process terminates when the expected length is met or no more fragments can be used for coverage restoration. In summary, our algorithm has the following modules:

1. de Bruijn graph generation
2. Target sequence length estimation
3. Euler Path traversal on solid de Bruijn tuples

4. Coverage Restoration

5. Termination

Coverage restoration can help us decide the order of long perfect repeats containing slight differences, because only one order of the long repeat can ensure fragments adjacency match for all layers. If we change the order of long repeat with slight difference in Figure 35, the fragments at the end will have to switch position with fragments at the front as well, which breaks the fragment adjacency among layers.

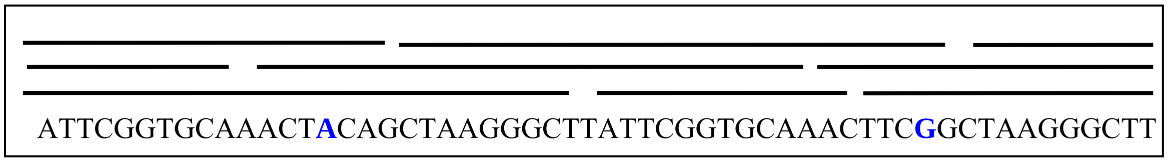

Figure 30: Determining the order of long repeats by adjacency

5. Fragment Assembler Design and Implementation

5.1 An Illustrative Example

Given an original sequence,

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaaatacta
```

Figure 36 shows the fragments to assemble with coverage of five and fragment IDs starting from 0 according to the order we read from file:

Fragment	FragmentID
gctagctgcaagtcagttactgagttaagtta	0
ttatttagttaatactttaacaatattat	1
tacggtatttaaaaaatacta	2
gctagctgcaagtcagttaactgagttaagttagtattta	3
gttaatacttttaacaatattattaaggtattttaaaaaatacta	4
gctagctgcagtcagttaactgagttaa	5
gttattatttagttaattactttta	6
acaatattattaaggtatttaaaaatacta	7
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt	8
ttaacaatattattaaggtatttaaaaaatacta	9
gctagctgcaag	10
tcagttaactgagttaagttattatttagttaatacttttaacaatattattaa	11
ggtatttaaaaaatacta	12

Figure 36: Fragment set for Assembly

Fragments are read into the Fragment table with an ID as the primary key and the fragments themselves as the index. Our assembly program chops the fragments into tuples of length 15 with a sliding window approach as shown in Figure 37:

```
gctagctgcaagtcagttactgagttaagtta    ->

gctagctgcaagtca, ctagctgcaagtcag, tagctgcaagtcagt, agctgcaagtcagtt, gctgcaagtcagtta,

ctgcaagtcagttac, tgcaagtcagttact, gcaagtcagttactg, caagtcagttactga, aagtcagttactgag,

agtcagttactgagt, gtcagttactgagtt,

tcagttactgagtta, cagttactgagttaa, agttactgagttaag, gttactgagttaagt, ttactgagttaagtt,

tactgagttaagtta
```

Figure 37: Retrieving tuples from a fragment with the sliding window approach

All the tuples are input to a MySQL database with the following fields:

- multiplicity (the number of occurrences in the fragment set)
- the fragment IDs of the fragments containing the tuple
- TupleID starting from 0 according to the order each tuple is read
- PredecessorIDs as the immediate tuple ahead of the current tuple
- SuccessorIDs as the immediate tuple behind the current tuple

TupleID	Tuple	Multiplicity	FragmentIDs	PredecessorIDs	SuccessorIDs
0	gctagctgcaagtca	3	0,3,8		1,1,122
1	ctagctgcaagtcag	2	0,3	0,0	2,2
2	tagctgcaagtcagt	2	0,3	1,1	3,3
3	agctgcaagtcagtt	2	0,3	2,2	4,4
4	gctgcaagtcagtta	2	0,3	3,3	5,40
5	ctgcaagtcagttac	1	0	4	6
6	tgcaagtcagttact	1	0	5	7
7	gcaagtcagttactg	1	0	6	8
8	caagtcagttactga	1	0	7	9
9	aagtcagttactgag	1	0	8	10
10	agtcagttactgagt	1	0	9	11
11	gtcagttactgagtt	1	0	10	12
12	tcagttactgagtta	1	0	11	13

TupleID	Tuple	Multiplicity	FragmentIDs	PredecessorIDs	SuccessorIDs
13	cagttactgagttaa	1	0	12	14
14	agttactgagttaag	1	0	13	15
15	gttactgagttaagt	1	0	14	16
16	ttactgagttaagtt	1	0	15	17
17	tactgagttaagtta	1	0	16	
18	ttatttagttaatac	3	1,8,11	,147,147	19,19,19
19	tatttagttaatact	3	1,8,11	18,18,18	20,20,20
20	atttagttaatactt	3	1,8,11	19,19,19	21,21
21	tttagttaatacttt	2	1,11	20,20	22,150
22	ttagttaatacttta	1	1	21	23
23	tagttaatactttaa	1	1	22	24
24	agttaatactttaac	1	1	23	25
25	gttaatactttaaca	1	1	24	26
26	ttaatactttaacaa	1	1	25	27
27	taatactttaacaat	1	1	26	28
28	aatactttaacaata	1	1	27	29
29	atactttaacaatat	1	1	28	30
30	tactttaacaatatt	1	1	29	31
31	actttaacaatatta	1	1	30	32
32	ctttaacaatattat	1	1	31	
33	tacggtatttaaaaa	1	2		34
34	acggtatttaaaaaa	1	2	33	35
35	cggtatttaaaaaat	1	2	34	36
36	ggtatttaaaaaata	3	2,9,12	35,149	37,37,37
37	gtatttaaaaaatac	3	2,9,12	36,36,36	38,38,38
38	tatttaaaaaatact	3	2,9,12	37,37,37	39,39,39
39	atttaaaaaatacta	3	2,9,12	38,38,38	
40	ctgcaagtcagttaa	1	3	4	41
41	tgcaagtcagttaac	1	3	40	42
42	gcaagtcagttaact	1	3	41	43
43	caagtcagttaactg	1	3	42	44
44	aagtcagttaactga	1	3	43	45

TupleID	Tuple	Multiplicity	FragmentIDs	PredecessorIDs	SuccessorIDs
45	agtcagttaactgag	2	3,5	44,100	46,46
46	gtcagttaactgagt	2	3,5	45,45	47,47
47	tcagttaactgagtt	3	3,5,11	46,46	48,48,48
48	cagttaactgagtta	3	3,5,11	47,47,47	49,49,49
49	agttaactgagttaa	3	3,5,11	48,48,48	50,50
50	gttaactgagttaag	2	3,11	49,49	51,51
51	ttaactgagttaagt	3	3,8,11	50,136,50	52,52,52
52	taactgagttaagtt	3	3,8,11	51,51,51	53,53,53
53	aactgagttaagtta	3	3,8,11	52,52,52	54,137,137
54	actgagttaagttag	1	3	53	55
55	ctgagttaagttagt	1	3	54	56
56	tgagttaagttagta	1	3	55	57
57	gagttaagttagtat	1	3	56	58
58	agttaagttagtatt	1	3	57	59
59	gttaagttagtattt	1	3	58	60
60	ttaagttagtattta	1	3	59	
61	gttaatacttttaac	2	4,11	,152	62,62
62	ttaatacttttaaca	2	4,11	61,61	63,63
63	taatacttttaacaa	2	4,11	62,62	64,64
64	aatacttttaacaat	2	4,11	63,63	65,65
65	atacttttaacaata	2	4,11	64,64	66,66
66	tacttttaacaatat	2	4,11	65,65	67,67
67	acttttaacaatatt	2	4,11	66,66	68,68
68	cttttaacaatatta	2	4,11	67,67	69,69
69	ttttaacaatattat	2	4,11	68,68	70,70
70	tttaacaatattatt	2	4,11	69,69	71,71
71	ttaacaatattatta	3	4,9,11	70,70	72,72,72
72	taacaatattattaa	3	4,9,11	71,71,71	73,73
73	aacaatattattaag	2	4,9	72,72	74,74
74	acaatattattaagg	3	4,7,9	73,73	75,75,75
75	caatattattaaggt	3	4,7,9	74,74,74	76,76,76
76	aatattattaaggta	3	4,7,9	75,75,75	77,77,77

TupleID	Tuple	Multiplicity	FragmentIDs	PredecessorIDs	SuccessorIDs
77	atattattaaggtat	3	4,7,9	76,76,76	78,78,78
78	tattattaaggtatt	3	4,7,9	77,77,77	79,79,79
79	attattaaggtattt	3	4,7,9	78,78,78	80,112,112
80	ttattaaggtatttt	1	4	79	81
81	tattaaggtatttta	1	4	80	82
82	attaaggtattttaa	1	4	81	83
83	ttaaggtattttaaa	1	4	82	84
84	taaggtattttaaaa	1	4	83	85
85	aaggtattttaaaaa	1	4	84	86
86	aggtattttaaaaaa	1	4	85	87
87	ggtattttaaaaaat	1	4	86	88
88	gtattttaaaaaata	1	4	87	89
89	tattttaaaaaatac	1	4	88	90
90	attttaaaaaatact	1	4	89	91
91	ttttaaaaaatacta	1	4	90	
92	gctagctgcagtcag	1	5		93
93	ctagctgcagtcagt	1	5	92	94
94	tagctgcagtcagtt	1	5	93	95
95	agctgcagtcagtta	1	5	94	96
96	gctgcagtcagttaa	1	5	95	97
97	ctgcagtcagttaac	1	5	96	98
98	tgcagtcagttaact	1	5	97	99
99	gcagtcagttaactg	1	5	98	100
100	cagtcagttaactga	1	5	99	45
101	gttattatttagtta	3	6,8,11	,146,146	102,102,102
102	ttattatttagttaa	3	6,8,11	101,101,101	103,103,103
103	tattatttagttaat	3	6,8,11	102,102,102	104,147,147
104	attatttagttaatt	1	6	103	105
105	ttatttagttaatta	1	6	104	106
106	tatttagttaattac	1	6	105	107
107	atttagttaattact	1	6	106	108
108	tttagttaattactt	1	6	107	109

109	ttagttaattacttt	1	6	108	110
110	tagttaattactttt	1	6	109	111
111	agttaattactttta	1	6	110	
112	ttattaaggtattta	2	7,9	79,79	113,113
113	tattaaggtatttaa	2	7,9	112,112	114,114
114	attaaggtatttaaa	2	7,9	113,113	115,115
115	ttaaggtatttaaaa	2	7,9	114,114	116,116
116	taaggtatttaaaaa	2	7,9	115,115	117,148
117	aaggtatttaaaaat	1	7	116	118
118	aggtatttaaaaata	1	7	117	119
119	ggtatttaaaaatac	1	7	118	120
120	gtatttaaaaatact	1	7	119	121
121	tatttaaaaatacta	1	7	120	
122	ctagctgcaagtcat	1	8	0	123
123	tagctgcaagtcatt	1	8	122	124
124	agctgcaagtcattt	1	8	123	125
125	gctgcaagtcattta	1	8	124	126
126	ctgcaagtcatttaa	1	8	125	127
127	tgcaagtcatttaac	1	8	126	128
128	gcaagtcatttaact	1	8	127	129
129	caagtcatttaactg	1	8	128	130
130	aagtcatttaactga	1	8	129	131
131	agtcatttaactgag	1	8	130	132
132	gtcatttaactgagt	1	8	131	133
133	tcatttaactgagtt	1	8	132	134
134	catttaactgagtta	1	8	133	135
135	atttaactgagttaa	1	8	134	136
136	tttaactgagttaag	1	8	135	51
137	actgagttaagttat	2	8,11	53,53	138,138
138	ctgagttaagttatt	2	8,11	137,137	139,139
139	tgagttaagttatta	2	8,11	138,138	140,140
140	gagttaagttattat	2	8,11	139,139	141,141
141	agttaagttattatt	2	8,11	140,140	142,142

TupleID	Tuple	Multiplicity	FragmentIDs	PredecessorIDs	SuccessorIDs
142	gttaagttattattt	2	8,11	141,141	143,143
143	ttaagttattattta	2	8,11	142,142	144,144
144	taagttattatttag	2	8,11	143,143	145,145
145	aagttattatttagt	2	8,11	144,144	146,146
146	agttattatttagtt	2	8,11	145,145	101,101
147	attatttagttaata	2	8,11	103,103	18,18
148	aaggtatttaaaaaa	1	9	116	149
149	aggtatttaaaaaat	1	9	148	36
150	ttagttaatactttt	1	11	21	151
151	tagttaatactttta	1	11	150	152
152	agttaatacttttaa	1	11	151	61

Figure 38: de Bruijn Tuples generated from the give fragment set

Before traversal starts, the assembler has a rough estimation of target sequence length by the following procedure:

1. Adding up the sum of all fragment lengths in the fragment table
2. Dividing the sum by two for reverse complement
3. Dividing the output of the previous step by coverage.

In the illustrative example, there is no reverse complement. The sum of all fragment lengths is 419, which is divided by the coverage five to get estimated sequence length eighty three. The terminating condition would be

1. Traversed path is longer than 105% of estimated length (eighty-six for our example)
2. Cannot resolve branch at a certain position
3. Traversed path longer than restored coverage at all layers. For coverage of five, there are five layers to restore.

Euler path traversal starts with the Tuple field that has no Predecessor field and a Multiplicity field larger than one as a solid tuple. In our example, traversal starts at Tuple 0 – `gctagctgcaagtca`. From the database, the assembler knows immediately the next tuple is either Tuple 1 or Tuple 122, so traversal encounters a branch to resolve. However, Tuple 122 has a multiplicity of one that indicates most likely it is caused by error. Traversal continues on Tuple 1 – `gctagctgcaagtcag`. Traversal is straightforward until the assembler arrives at Tuple 4, `gctagctgcaagtcagtta`, where the successor can be either Tuple 5 or Tuple 40, both with a multiplicity of one. At this location, coverage restoration starts.

For coverage restoration, the assembler begins with tuple `gctagctgcaagtca` to withdraw fragments – Fragment 0, Fragment 3, and Fragment 8 in the Fragment Table – containing the beginning tuple. The assembler applies the dynamic string alignment algorithm to align the existing traversed path with the three fragments as shown in Figure 39.

```
gctagctgcaagtcagtta                                    Traversed Path
gctagctgcaagtcagttactgagttaagtta                       Fragment 0 at Coverage Layer 1
gctagctgcaagtcagttaactgagttaagttagtattta               Fragment 3 at Coverage Layer 2
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt     Fragment 8 at Coverage Layer 3
```

Figure 39: Initial coverage restoration

By the majority rule, the assembler moves on to `ctgcaagtcagttaa` or Tuple 40 for the next tuple. The successor of Tuple 40 is Tuple 41, which is another weak tuple, so the assembler refers to the restored coverage layers to decide the next tuple. Again with the majority rule, Tuple 41 is the right choice. Traversal continues until Tuple 53, `gctagctgcaagtcagttaactgagtta,` where traversal jumps to Tuple 137 because Tuple 54 is weak. Continuing from Tuple 137 to Tuple 146, the path is `gctagctgcaagtcagttaactgagttaagttattatttagtt`. At tuple 147, traversal goes to Tuple 101, Tuple 102, Tuple 103, Tuple 147, Tuple 18, Tuple 19, Tuple 20, and Tuple 21 `gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt,` where the successor can be Tuple 22 or Tuple 150. Here coverage restoration starts again. Restoration starts from the first layer, beginning with tuple `gctagctgcaagtcagtta,` and three fragments available for alignment as shown in Figure 40:

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt        Traversed path Length 51
gctagctgcaagtcagttactgagttaagtta                           Fragment 0 Length 32
gctagctgcaagtcagttaactgagttaagttagtattta                   Fragment 3 Length 40
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt         Fragment 8 Length 50
```

Figure 40: Available fragments for restoration at the beginning of Layer 1

The assembler aligns each of these fragments with the traversed path. The traversed path is significantly longer than Fragment 0, so the assembler uses the beginning portion of the traversed path – the first thirty-eight nucleotides for alignment or length of Fragment 0 plus six. Because there are six more nucleotides in the traversed path, the actual alignment score for Fragment 0 should be six insertion-scores less than the alignment-score of the first thirty-eight nucleotides of the traversed path and Fragment 0. The six extra nucleotides are to tolerate some insertions for the fragment. The error rate of alignment is the sum of insertion (excluding the extra length), deletion for alignment, and the difference between fragment and traverse path divided by the fragment length. The assembler does not use a fragment for restoration at a layer position if the error rate is two times higher than the overall error rate in a fragment set. This logic is to prevent long fragments from achieving a high score by length in spite of errors.

```
gctagctgcaagtcagtta ctgagttaagtta                           Fragment 0
gctagctgcaagtcagttaactgagttaagttattatt                      beginning of traversed path
gctagctgcaagtcagttaactgagttaagttagtattta                    Fragment 3
gctagctgcaagtcagttaactgagttaagttattatttagttaata             beginning of traversed path
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt          Fragment 8
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt         traversed path
```

Figure 40: Aligning fragments with traversed path

Among the fragments that are suitable for coverage restoration at a layer position, the fragment with highest score is selected. The traversed path is most likely to be correct because the assembler only traverses solid tuples or resolves branches by majority rule. So, for Layer 1, Fragment 8 has the highest score. The fragment selection for coverage restoration is based on the indexed tuple. After aligning Fragment 8 at Layer 1, there is only one nucleotide left that is shorter than a tuple, so restoration stops at the fiftieth nucleotide for Layer 1. Note that the Fragment table has a Consumed field to record whether a fragment's position has been determined, with 1 for permanently determined and 2 for temporarily determined. Fragment 8 is marked Consumed=1, so that future restoration cannot use this fragment. For Layer 2, restoration stops after the alignment of Fragment 3. For Layer 3, after the alignment of Fragment 0 the restoration continues with Tuple 18 `ttatttagttaatac`.

From the de Bruijn table, Fragment 1, Fragment 8, and Fragment 11 contain Tuple 18. The assembler retrieves Fragment 1, Fragment 8, and Fragment 11 to align with the rest of the traversed path `ttatttagttaatacttt` at Layer 3.

```
           gctagctgcaagtcagtta ctgagttaagtta
           gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt
ttatttagttaatacttt                                           Remain of traversed path
ttatttagttaatactttaacaatattat                                Fragment 1
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt           Fragment 8
tcagttaactgagttaagttattatttagttaatacttttaacaatattattaa       Fragment 11
```

Figure 41: Aligning fragments with partial traversed path

For the alignment, the Fragment length is much longer than the rest of the path, so the assembler aligns the beginning of the Fragment with the rest of the traversed path. For Fragments 8 and Fragments 11, Tuple 18 is in the middle of the fragments. Fragments 8 and Fragments 11 cannot be used for Layer 3 restoration because by definition, a layer excludes overlap. Thus Fragment 1 is used for coverage restoration at Layer 3 following Fragment 0. Restoration at Layer 3 stops because the length of Layer 3 is longer than the traversed path. Fragment 1 is marked Consumed = 2 because it is temporarily aligned at the current position – as the traversed path extends later on, there can be other suitable fragments to align at the current position of Fragment 1 at Layer 3. The current restoration is shown in Figure 42:

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt             Traversed Path
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt              Layer 1
gctagctgcaagtcagttaactgagttaagttagtattta                        Layer 2
gctagctgcaagtcagtta ctgagttaagttattatttagttaatactttaacaatattat  Layer 3
```

Figure 42: Coverage restoration for the first three layers

There are five layers for a fragment set of coverage five. For the fourth layer, there are no more fragments containing first tuple `gctagctgcaagtca`, so the assembler tries the next tuple in the traversed path in a slide window approach until there are fragments with Consumed=0. At Tuple 45 or `agtcagttaactgag`, Fragment 5 is an available fragment for Layer 4. Restoration continues at Layer 4, on Tuple 101 or `gttattatttagtta`, with available Fragment 6 and Fragment 11. Fragment 6 is temporarily selected for Layer 4. Similarly for Layer 5 restoration, the restoration of all five layers is shown in Figure 43:

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt                    Traversed Path
gctagctgc agtcagttaactgagttaa                                          Layer 4
gttattatttagttaattactttta                                              Fragment 6
tcagttaactgagttaagttattatttagttaatacttttaacaatattattaa                 Fragment 11


gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt                    Traversed Path
gctagctgc agtcagttaactgagttaagttattatttagttaatactttta                  Layer 4


gctagctgcaagtcagttaactgagttaagttattatttagttaatacttt                    Traversed Path
gctagctgcaagtcatttaactgagttaagttattatttagttaatactt                     Layer 1
gctagctgcaagtcagttaactgagttaagttagtattta                               Layer 2
gctagctgcaagtcagtta ctgagttaagttattatttagttaatactttaacaatattat         Layer 3
gctagctgc agtcagttaactgagttaagttattatttagttaatactttta                  Layer 4
            tcagttaactgagttaagttattatttagttaatacttttaacaatattattaa     Layer 5
```

Figure 43: Coverage restoration for all five layers

From the restored layers, the assembler moves to Tuple 150 or **`ttagttaatactttt`**. Due to the weak multiplicity of Tuple 151 and Tuple 152, the assembler continues to rely on restoration for path traversal until Tuple 61.Traversal continues until Tuple 116, where the traversed path is

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaa
```

Restoration starts again. Each time restoration starts, the assembler sets fragments with Consumed = 2 back to 0 because those fragments are partially aligned with traversed path and might be adjusted to another position for better alignment with a longer traversed path. Figure 44 shows the beginning of restoration with temporarily aligned fragments in last restoration removed and a longer traversed path:

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaa
            Traversed Path

gctagctgcaagtcatttaactgagttaagttattatttagttaatactt                                 Layer 1
gctagctgcaagtcagttaactgagttaagttagtattta                                           Layer 2
gctagctgcaagtcagtta ctgagttaagtta                                                  Layer 3
gctagctgc agtcagttaactgagttaa                                                      Layer 4
                                                                                   Layer 5
```

Figure 44: Continue restoration with temporarily aligned fragment removed

Following the same logic, the restoration ends as shown in Figure 45.

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaa
            Traversed Path

gctagctgcaagtcatttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaaatacta
            Layer 1
gctagctgcaagtcagttaactgagttaagttagtatttagttaatacttttaacaatattattaaggtatttaaaaaatacta
            Layer 2
gctagctgcaagtcagtta ctgagttaagttattatttagttaatacttt aacaatattat
            Layer 3
gctagctgc agtcagttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaatacta
            Layer 4
             tcagttaactgagttaagttattatttagttaatacttttaacaatattattaa
            Layer 5
```

Figure 45: Coverage restoration for a longer traversed path

Again, by majority rule traversal continues on Tuple 148, Tuple 149, Tuple 36, Tuple 37, Tuple 38, and finally ends at Tuple 39.

```
gctagctgcaagtcagttaactgagttaagttattatttagttaatacttttaacaatattattaaggtatttaaaaaatacta
```

This is an exact match for the original sequence.

5.3 Complexity Analysis

We define *n* as the total amount of nucleotide bases in the fragment set. The runtime complexity for the insertion of fragments into the fragment table is *O(2In+Kn)*. For each fragment, the corresponding reverse complement is calculated for insertion as well. *K* is the time constant for reverse complement calculation. *I* is the time constant for database insertion. The creation of the de Bruijn table is *O(Mn),* where *M* is the time constant for seeking predecessors and successors of a tuple. Fragment table has fragment ID as the primary key. De Bruijn table has the TupleID as primary key and Tuple as index. Before insertion of a tuple, existence of the tuple is checked: if the tuple exists, update; otherwise insert. The dynamic fragment alignment algorithm has $O(L^2n)$, where L is the average fragment length. Thus the overall runtime complexity is O(n). The space requirement for the assembler is also O(n).

5.4 Program Architecture

All core operations are programmed in C to achieve good runtime efficiency and then encapsulated with C++ classes for an object-oriented design. The MySQL DBMS is used for data storage. The development platform is Visual C++ .NET Windows XP because of the debugging aid of runtime checking.

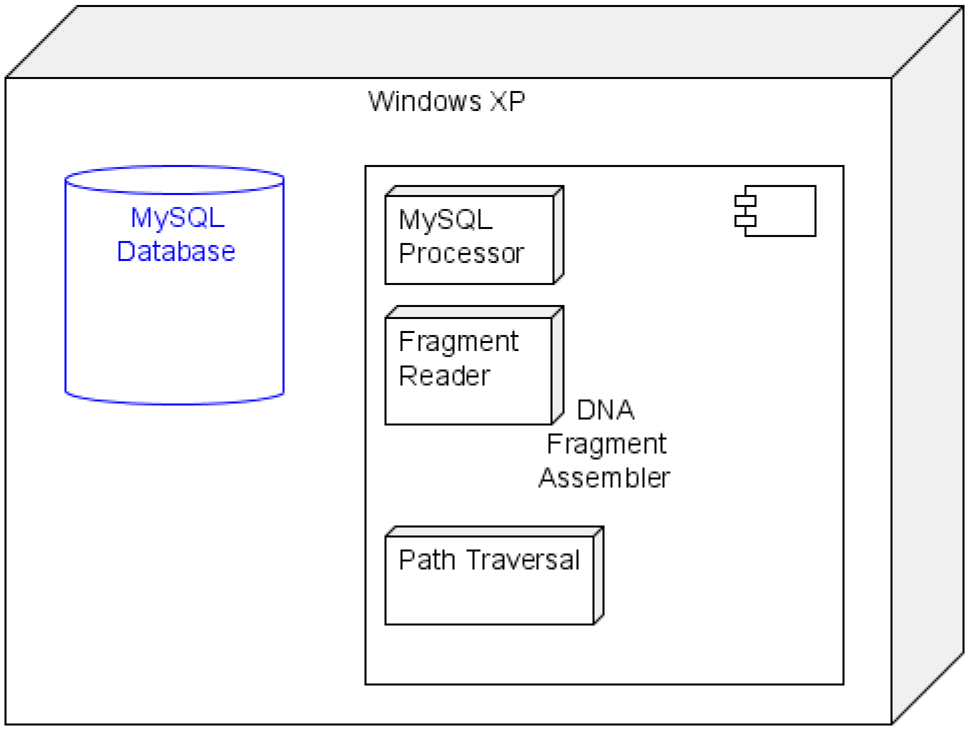

Figure 46: Deployment diagram for implemented assembler

There are three C++ classes for assembly algorithm implementation: MySQL_Processor, FragmentReader, and Restorer. The MySQL_Processor class encapsulates all operations required to communicate with the MySQL database. The FragmentReader class imports fragments from an input file, calculates the reverse complement of each fragment, and creates the fragment table and the de Bruijn table. The Restorer class traverses the Euler path and restores fragment coverage.

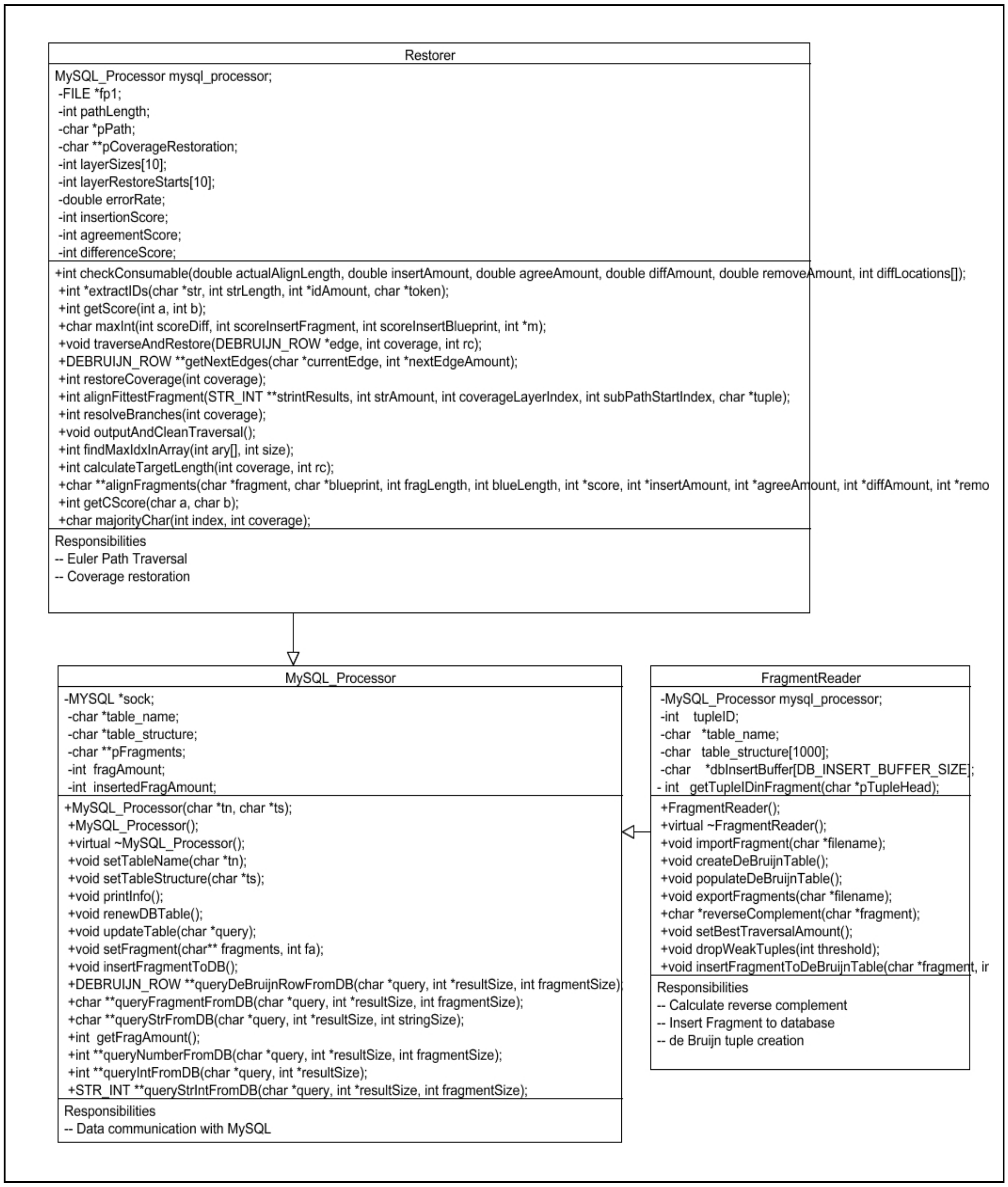
Restorer
MySQL_Processor mysql_processor;
-FILE *fp1;
-int pathLength;
-char *pPath;
-char **pCoverageRestoration;
-int layerSizes[10];
-int layerRestoreStarts[10];
-double errorRate;
-int insertionScore;
-int agreementScore;
-int differenceScore;
+int checkConsumable(double actualAlignLength, double insertAmount, double agreeAmount, double diffAmount, double removeAmount, int diffLocations[]);
+int *extractIDs(char *str, int strLength, int *idAmount, char *token);
+int getScore(int a, int b);
+char maxInt(int scoreDiff, int scoreInsertFragment, int scoreInsertBlueprint, int *m);
+void traverseAndRestore(DEBRUIJN_ROW *edge, int coverage, int rc);
+DEBRUIJN_ROW **getNextEdges(char *currentEdge, int *nextEdgeAmount);
+int restoreCoverage(int coverage);
+int alignFittestFragment(STR_INT **strintResults, int strAmount, int coverageLayerIndex, int subPathStartIndex, char *tuple);
+int resolveBranches(int coverage);
+void outputAndCleanTraversal();
+int findMaxIdxInArray(int ary[], int size);
+int calculateTargetLength(int coverage, int rc);
+char **alignFragments(char *fragment, char *blueprint, int fragLength, int blueLength, int *score, int *insertAmount, int *agreeAmount, int *diffAmount, int *remo
+int getCScore(char a, char b);
+char majorityChar(int index, int coverage);
Responsibilities
-- Euler Path Traversal
-- Coverage restoration
MySQL_Processor
-MYSQL *sock;
-char *table_name;
-char *table_structure;
-char **pFragments;
-int fragAmount;
-int insertedFragAmount;
+MySQL_Processor(char *tn, char *ts);
+MySQL_Processor();
+virtual ~MySQL_Processor();
+void setTableName(char *tn);
+void setTableStructure(char *ts);
+void printInfo();
+void renewDBTable();
+void updateTable(char *query);
+void setFragment(char** fragments, int fa);
+void insertFragmentToDB();
+DEBRUIJN_ROW **queryDeBruijnRowFromDB(char *query, int *resultSize, int fragmentSize);
+char **queryFragmentFromDB(char *query, int *resultSize, int fragmentSize);
+char **queryStrFromDB(char *query, int *resultSize, int stringSize);
+int getFragAmount();
+int **queryNumberFromDB(char *query, int *resultSize, int fragmentSize);
+int **queryIntFromDB(char *query, int *resultSize);
+STR_INT **queryStrIntFromDB(char *query, int *resultSize, int fragmentSize);
Responsibilities
-- Data communication with MySQL
FragmentReader
-MySQL_Processor mysql_processor;
-int tupleID;
-char *table_name;
-char table_structure[1000];
-char *dbInsertBuffer[DB_INSERT_BUFFER_SIZE];
- int getTupleIDinFragment(char *pTupleHead);
+FragmentReader();
+virtual ~FragmentReader();
+void importFragment(char *filename);
+void createDeBruijnTable();
+void populateDeBruijnTable();
+void exportFragments(char *filename);
+char *reverseComplement(char *fragment);
+void setBestTraversalAmount();
+void dropWeakTuples(int threshold);
+void insertFragmentToDeBruijnTable(char *fragment, ir
Responsibilities
-- Calculate reverse complement
-- Insert Fragment to database
-- de Bruijn tuple creation

Figure 47: Class diagram for implemented assembler

5.5 Database Schema

The Fragment Table has three columns – FragmentID, Fragment, and Consumed. Fragment ID starts with 0, fragments with even-numbered IDs are original fragments from an input file. A fragment with an odd-numbered ID is the reverse complement of the fragment with an even-numbered ID immediately preceding the odd number. For example, a fragment with ID 0 and a fragment with ID 1 are the reverse complements of each another. The Consumed column signals the state of a fragment in coverage restoration: 0 for available to use, 1 for permanent used for restoration, and 2 temporarily used for restoration.

FragmentID (Type:int) (Primary Key)	Fragment (Type: varchar) (Secondary Key)	Consumed (Type: int)
…	…	…

Table 2: Fragment Table structure

The deBruijn Table contains all solid tuples generated by fragments with the sliding window approach. An ID is associated with each Tuple for fragment encoding. The Multiplicity field records the number of occurrences of a tuple in the fragment set for statistical analysis. Tuples with Multiplicity less than a threshold are not traversed by the Euler path. The FragmentIDs column of a tuple concatenates all fragment IDs of the fragment that contains the tuple. FragmentIDs column speeds up fragment layout during coverage restoration –relevant fragments containing a tuple can be retrieved quickly. PredecessorIDs column concatenates Tuple IDs preceding a tuple. SuccessorIDs column concatenates TupleIDs following a tuple.

Tuple ID (Type:int, Secondary Key)	Tuple (Type: 20-character-string) (Primary Key)	Multiplicity (Type:int)	FragmentIDs (Type: varchar) (Foreign keys separated by ‘,’)	PredecessorIDs (Type: varchar)	SuccessorIDs (Type: varchar)
…	…	…	…	…	…

Table 3: deBruijn Table structure

Figure 52 illustrates the entity relationship between de Bruijn table and Fragment table.

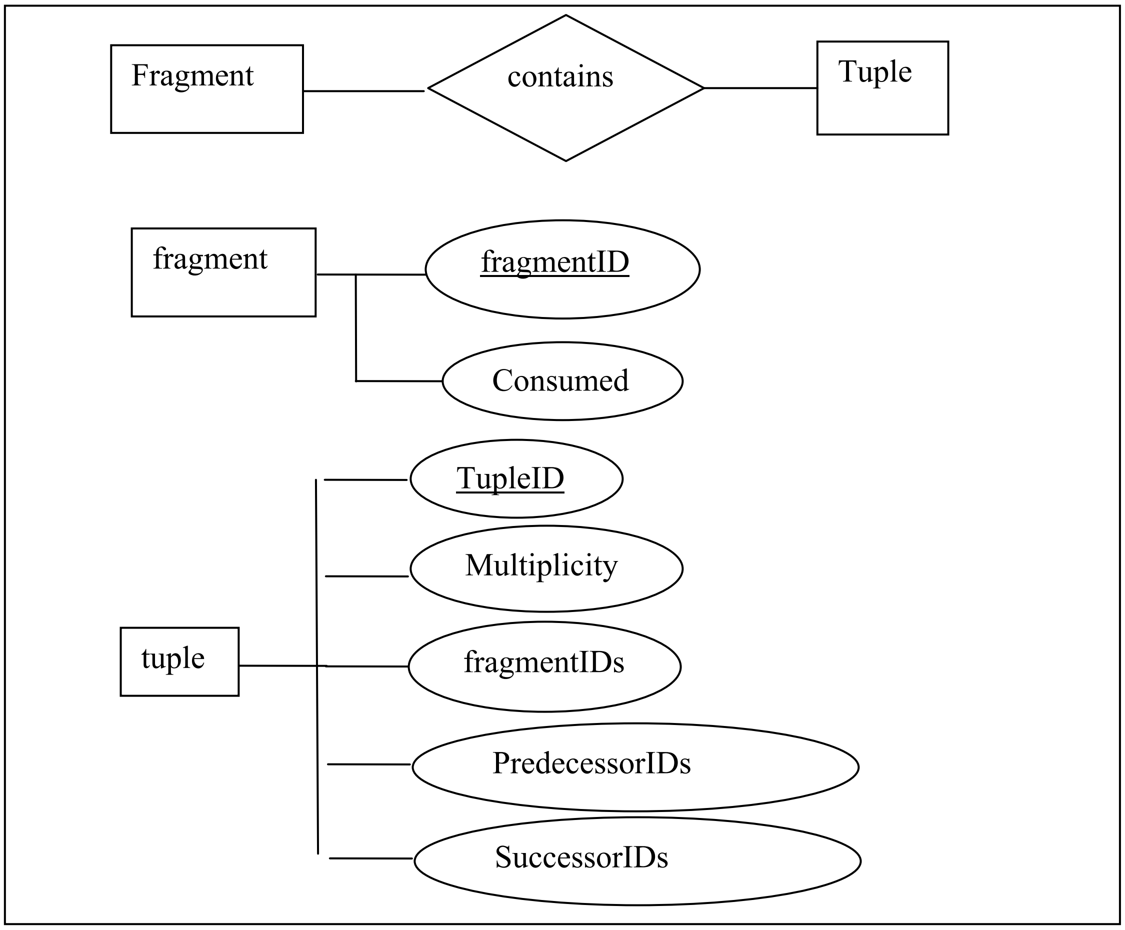

Figure 48: Entity-relationship models between deBruijnTable and FragmentTable

6. Test and Result

The assembly program has been tested with a section of the TIGR_GMG sequence modified that contains a two-fold long repeat. Each fold is about 1600 bases long with slight differences among the folds. The original sequence is 3275 bases long. The sequence is then mutated with GenFrag to have 1.98% error rate and the coverage of ten. The output of the assembly is a sequence 3264 nucleotide-bases long.

Only four mistakes were found in the output:

1. One insertion of twenty nucleotide bases
2. Two deletion of fifteen nucleotide bases
3. One deletion of one nucleotide base.

The test is performed on a computer equipped with Intel® Core 2 Duo CPU 2.00 GHz and 1.99 GB of RAM. It takes fifty minutes to load the 30K fragment set into the MySQL database. On the other hand, it takes only five minutes to finish Euler path traversal and Coverage Restoration for the DNA fragment assembly.

7. Conclusion

DNA Fragment Assembly is a key process for DNA sequencing. Due to current technical limitations, a long target DNA sequence is cloned into multiple copies. These copies need to be randomly fractured to fragments less than 1000 nucleotide bases in length. After analysis on individual fragments, all fragment reads need to be assembled together to rebuild the original target sequence. DNA Fragment Assembly algorithms have to overcome several challenges to correctly rebuild the original target sequence from fragments – DNA double helix structure, sequencing errors, repeats, and insufficient coverage. All existing DNA Fragment Assembly algorithms are hindered by these challenges. In particular, repeats longer than fragment lengths are nearly impossible to assemble correctly with current assembly algorithms. Tedious finishing reaction experiments have to be carried out to manually restore target DNA sequences at regions containing those long repeats. Hence, there is still considerable need for improvements of repeat resolution, error correction, and runtime efficiency on DNA Fragment Assembly.

Aiming at improving DNA Fragment Assembly performance in these areas, we propose a number of enhancements for the Euler Assembler developed by Pevzner et al.:

1. Traversal approach for Euler Superpath discovery,
2. Statistical Analysis for error and repeat detection,
3. Perfect long repeats fragment assembly.

In addition, we provide an innovative genetic algorithm to restore the coverage of fragments on target sequence. Our genetic algorithm forms solutions with a portion of fragments covering the target sequence as blueprints to restore the coverage of all fragments. Fast pattern matching techniques are applied to evaluate the fitness of a solution. The genetic algorithm determines the sequence order among copies of long repeats with slight differences, because only one order is correct to restore all layers of fragments covering the repeat region. We combine our enhanced Euler algorithms and the genetic algorithm to ensure runtime efficiency. This solution is close to optimal. Future research might address the issue of load time, platform independence, and scalability.

8. Future Research

The following steps might improve upon our work:

1. Improve loading time. Loading time is the bottleneck of our assembler, though the runtime complexity is O(n). The assembler communicates with the MySQL database through direct C API call. We can consider using Oracle SQL loader or writing SQL scripts to further speed up the process of fragment input and tuple initialization.
2. Linux instead of XP for better performance. Initially the assembler is designed for the Linux platform. However, the gcc compiler on the Linux platform does not have runtime memory infringement checking. There were a number of memory infringement bugs in the assembler causing assembly result inconsistency. Visual C++ .NET compiler was then used on Windows platform to debug the assembler for the memory infringement issue. The runtime memory checking feature of Visual C++ .NET compiler significantly slows down the execution of Visual C++ application. Due to time constraint, no final testing has been done on the Linux platform. Based on experience, on the Linux platform the assembler can load data to database two to three times faster than Windows platform.
3. Larger test data set. Scalability is important for DNA Fragment Assembly. In theory our assembler is highly scalable since the overall runtime complexity is O(n). The theoretical runtime analysis needs to be verified by thorough testing with large data set.

References:

[1] Tammi, M. T. (2003). The Principles of Shotgun Sequencing and Automated Fragment Assembly. Aug 11, 2007, http://web.cgb.ki.se/student/sfa.pdf

[2] The Wellcome Trust Sanger Institute. (Sep 21, 2005). Jan 12, 2008, http://www.sanger.ac.uk/Info/Intro/sanger.shtml

[3] Li, L., & Khuri, S. (2004, Jun.) A Comparison of DNA Fragment Assembly Algorithms. *Proceedings of the 2004 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, 329-335.

[4] Muluykov, Z., & Pevzner, P.A. (2002). EULER-PCR: finishing experiments for repeat resolution in DNA sequencing. *Pacific Symposium on Biocomputing*, 199-210.

[5] Smith, J. (Aug 2, 2007). *IUPAC Table*, Jan 13, 2008, http://droog.gs.washington.edu/parc/images/iupac.html

[6] Luque, G., & Alba, E. (2005). Metaheuristics for the DNA Fragment Assembly Problem. *International Journal of Computational Intelligence Research*, *1(2),* 98–108.

[7] Green, P. (1998). *Phrap Documentation,* Dec 12. 2007, http://www.phrap.org/phredphrapconsed.html

[8] Sutton, G. G. et al., (1995) TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects. *Genome Science and Technology, 1,* 9-19.

[9] Huang, X., & Madan, A. (1999). CAP3: A DNA sequence assembly program. *Genome Res.*, *9*, 868-877.

[10] Pevzner, P. A., Tang, H., & Waterman, M.S., (2001). A New Approach to Fragment Assembly in DNA Sequencing. *Proceedings of the 5th Annual International Conference on Computational Molecular Biology,* 256-267.

[11] Myers, G. (1999, May). Whole-Genome DNA Sequencing. *Computing in Science & Engineering*, 33 – 43.

[12] Khuri, S. (2007, Jan). *CS255 Design and Analysis of Algorithms Lecture note,* Computer Science Department San Jose State University

[13] Batzoglou, S., Jaffe, D. B., Stanley, K., Butler, J., & Lander, E. S. (2002). ARACHNE : a whole-genome shotgun assembler. *Genome Research, 12,* 177-189.

[14] Kim, S., & Segre, A. M., (1999). AMASS : A Structured Pattern Matching Approach to Shotgun Sequence Assembly. *Journal of Computational Biology, 6 (4)*

[15] Gallant, J., Maier, D., & Storer, J., (1980). On finding minimal length superstrings. *Journal of Computer and System Sciences*, *20*, 50–58.

[16] Staden, R. (1979). A strategy of DNA sequencing employing computer programs. *Nucleic Acids Res*., *6*, 2601–2610.

[17] Myers, E. (1995). A sublinear algorithm for approximate keyword matching. *Algorithmica*, *12 (4–5),* 345–374.

[18] Pevzner, P. A., Tang, H., & Waterman M. S. (2001). An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, *98,* 9748-9753.

[19] Dennis, D. W., William, M., & Richard L. S. (2002). *Mathematical Statistics with Applications 6th edition*, 348. Thomson.

[20] Pevzner, P. A., & Tang, H. (2001). Fragment assembly with double-barreled data. Bioinformatics. *Proceedings of ISMB,* 225-33.

[21] Parsons, R., Forrest, S., and Burks, C. (1995). Genetic Algorithms, Operators, and DNA Fragment Assembly. *Machine Learning*, 1-24.

[22] Kikuchi, S., & Chakraborty, G. (2006, July). Heuristically Tuned GA to Solve Genome Fragment Assembly Problem. *Proceedings of IEEE World Congress on Computational Intelligence - Conference on Evolutionary Computation,* 5640-5647.

[23] Parsons, R., & Johnson, M. E. (1995) DNA sequence assembly and genetic algorithms new results and puzzling insights. *Proc Int Conf Intell Syst Mol Biol*, *(3),* 277-84.

Appendix Logic of Key Assembly Modules

A.1 de Bruijn Graph Creation

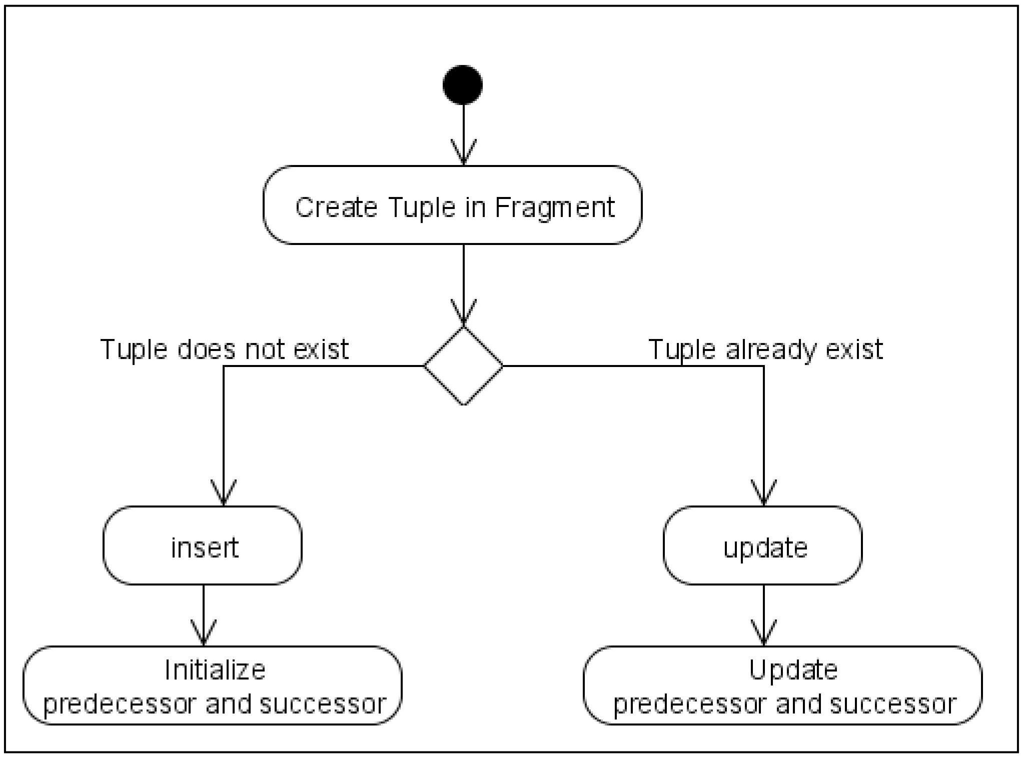

Figure 49: Activity diagram for de Bruijn graph creation

```
void FragmentReader::insertFragmentToDeBruijnTable(char *fragment, int fragmentID)
{
    char query[1000];
    char insertStatement[1000];
    int i;
    int frag_size = 0;
    int predecessorID = -1;
    int successorID;
    int currentID;

    frag_size = strlen(fragment);
    int insertLength = frag_size - TUPLE_SIZE;
    currentID = getTupleIDinFragment(fragment);
    successorID = getTupleIDinFragment(fragment+1);

    for (i = 0; i <= insertLength; i++)
    {
       dbInsertBuffer[0] = (char *)calloc(TUPLE_SIZE+1, sizeof(char));
       strncpy(dbInsertBuffer[0], fragment+i, TUPLE_SIZE);

       if (currentID < 0)
       {
          if(i == insertLength)
          {
            sprintf(insertStatement, "insert into deBruijnTable (TupleID, Tuple,
Multiplicity, FragmentIDs, BestTraversalAmount, TraversedAmount, PredecessorIDs,
SuccessorIDs) values(%d, '%s', 1, '%d', 0, 0, '%d', '')", tupleID, dbInsertBuffer[0],
fragmentID, predecessorID);
         }
         else if(i == 0)
         {
            if (successorID < 0)
                sprintf(insertStatement, "insert into deBruijnTable (TupleID, Tuple,
Multiplicity, FragmentIDs, BestTraversalAmount, TraversedAmount, PredecessorIDs,
SuccessorIDs) values(%d, '%s', 1, '%d', 0, 0, '', '%d')", tupleID, dbInsertBuffer[0],
fragmentID, tupleID+1);
            else
                sprintf(insertStatement, "insert into deBruijnTable (TupleID, Tuple,
Multiplicity, FragmentIDs, BestTraversalAmount, TraversedAmount, PredecessorIDs,
SuccessorIDs) values(%d, '%s', 1, '%d', 0, 0, '', '%d')", tupleID, dbInsertBuffer[0],
fragmentID, successorID);
          }
          else
         {
            if (successorID < 0)
                sprintf(insertStatement, "insert into deBruijnTable (TupleID, Tuple,
Multiplicity, FragmentIDs, BestTraversalAmount, TraversedAmount, PredecessorIDs,
SuccessorIDs) values(%d, '%s', 1, '%d', 0, 0, '%d', '%d')", tupleID, dbInsertBuffer[0],
fragmentID, predecessorID, tupleID+1);
            else
                sprintf(insertStatement, "insert into deBruijnTable (TupleID, Tuple,
Multiplicity, FragmentIDs, BestTraversalAmount, TraversedAmount, PredecessorIDs,
SuccessorIDs) values(%d, '%s', 1, '%d', 0, 0, '%d', '%d')", tupleID, dbInsertBuffer[0],
fragmentID, predecessorID, successorID);
          }

           mysql_processor.updateTable(insertStatement);
           tupleID++;
       }
       else
       {
            //deBruijn fragment already exist, update...;
            if(i == insertLength)
            {
               sprintf(query, "update deBruijnTable set Multiplicity=Multiplicity+1,
FragmentIDs=concat(FragmentIDs,',%d'), PredecessorIDs=concat(PredecessorIDs,',%d')
where TupleID=%d", fragmentID, predecessorID, currentID);
            }
            else if(i == 0)
            {
               if (successorID < 0)
                 sprintf(query, "update deBruijnTable set Multiplicity=Multiplicity+1,
FragmentIDs=concat(FragmentIDs,',%d'), SuccessorIDs=concat(SuccessorIDs,',%d') where
TupleID=%d", fragmentID, tupleID, currentID);
               else
                 sprintf(query, "update deBruijnTable set Multiplicity=Multiplicity+1,
FragmentIDs=concat(FragmentIDs,',%d'), SuccessorIDs=concat(SuccessorIDs,',%d') where
TupleID=%d", fragmentID, successorID, currentID);
            }
            else
            {
               if (successorID < 0)
                 sprintf(query, "update deBruijnTable set Multiplicity=Multiplicity+1,
FragmentIDs=concat(FragmentIDs,',%d'), PredecessorIDs=concat(PredecessorIDs,',%d'),
SuccessorIDs=concat(SuccessorIDs,',%d') where TupleID=%d", fragmentID, predecessorID,
tupleID, currentID);
               else
                 sprintf(query, "update deBruijnTable set Multiplicity=Multiplicity+1,
FragmentIDs=concat(FragmentIDs,',%d'), PredecessorIDs=concat(PredecessorIDs,',%d'),
SuccessorIDs=concat(SuccessorIDs,',%d') where TupleID=%d", fragmentID, predecessorID,
successorID, currentID);
```

```
            }

            mysql_processor.updateTable(query);
            }

            free(dbInsertBuffer[0]);

            predecessorID = currentID;
            if(predecessorID < 0) //predecessorID cannot be absent in the middle of the 
loop
            {
                        predecessorID = tupleID - 1;
            }
            currentID = successorID;
            successorID = getTupleIDinFragment(fragment+i+2);
       }
}
```

Code Listing 1: Fragment insertion to database

A.2 Estimating the Length of Target Sequence

```
int Restorer::calculateTargetLength()
{
   int charAmt = 0;
   int **lenResult;
   char charLenQuery[500];
   int fragmentAmount;
   int tmp;
   int **amtResult;
   char *fragAmtQuery = "select count(*) from fragmenttable";

   amtResult = mysql_processor.queryIntFromDB(fragAmtQuery, &tmp);
   fragmentAmount = *amtResult[0];
   free(amtResult);
   for (int i = 0; i < fragmentAmount; i++)
   {
     sprintf(charLenQuery, "select CHAR_LENGTH(Fragment) from fragmenttable where 
FragmentID = %d", i);
     lenResult = mysql_processor.queryIntFromDB(charLenQuery, &tmp);
     charAmt = charAmt + (*lenResult[0]);
   }

   int targetLength = charAmt/(2*10);

   return targetLength;
}
```

Code Listing 2: Target sequence length estimation

A.3 Euler Path Traversal

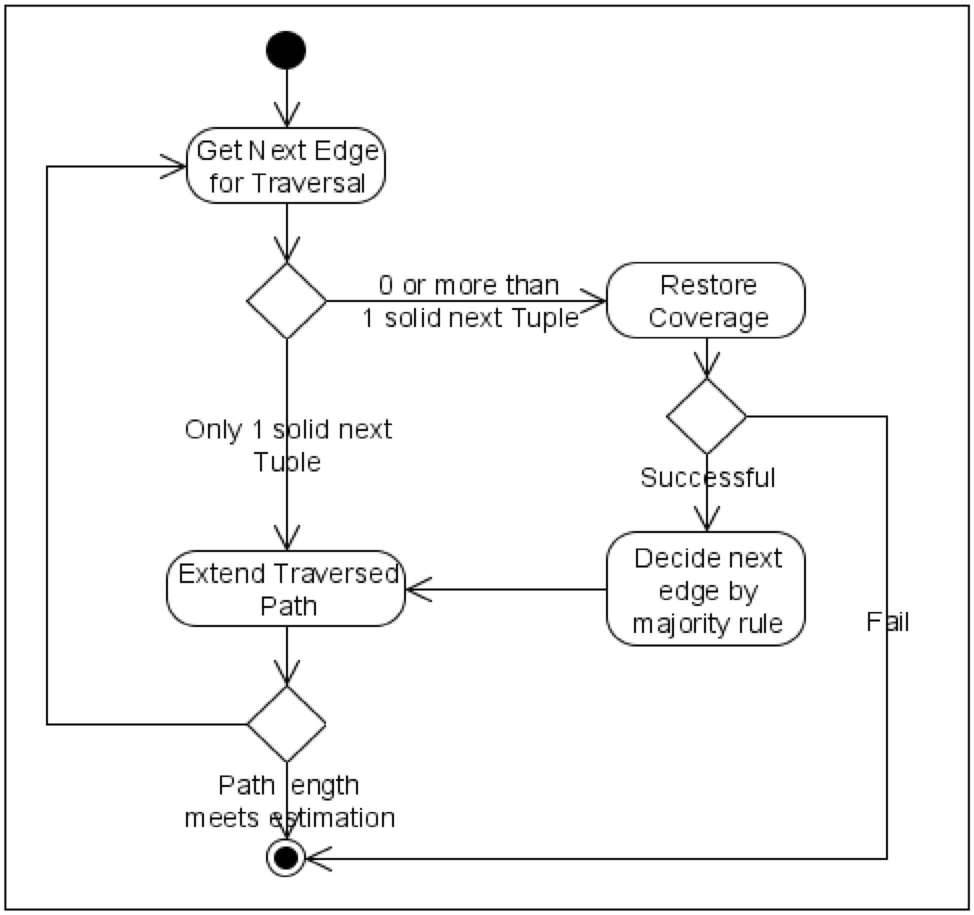
Get Next Edge for Traversal
0 or more than 1 solid next Tuple
Restore Coverage
Only 1 solid next Tuple
Successful
Extend Traversed Path
Decide next edge by majority rule
Fail
Path length meets estimation

Figure 50: Activity diagram for Euler Path Traversal

```
void Restorer::traverseAndRestore(DEBRUIJN_ROW *edge, int coverage)
{
    char *nextEdgeRegexp = (char *)calloc(TUPLE_SIZE+1, sizeof(char));
    char *currentEdge = (char *)calloc(TUPLE_SIZE+1, sizeof(char));
    int targetLength = calculateTargetLength();
    fprintf(stdout, "\n targetLength is %d \n", targetLength);
    DEBRUIJN_ROW **pRowResult = NULL;
    int edgeAmount;
    char updateQuery[500];
    int bDone = 0;

    strcpy(currentEdge, edge->Tuple);
    strcpy(pPath, edge->Tuple);
    pathLength = TUPLE_SIZE;
    sprintf(updateQuery, "update deBruijnTable set TraversedAmount=TraversedAmount+1
    where Tuple='%s'", currentEdge);
    mysql_processor.updateTable(updateQuery);

    while(!bDone)
    {
       pRowResult = getNextEdges(currentEdge, &edgeAmount);

      // more than 1 choice, need to resolve with fragment reads
       if (edgeAmount > 1)
       {
          if(resolveBranches()==0)
             bDone = 1;
       }

       if (edgeAmount == 1)
       {
          pPath[pathLength] = pRowResult[0]->Tuple[TUPLE_SIZE-1];
          pPath[pathLength+1] = '\0';
          pathLength++;
          if (pathLength > targetLength)
          {
             bDone = 1;
          }

       }

       if (edgeAmount == 0)
       {
             if (layerSizes[0] > pathLength)
             {
                pPath[pathLength] = majorityChar(pathLength);
                pathLength++;
                pPath[pathLength] = '\0';
             }
             else
             {
                if(resolveBranches()==0)
                         bDone = 1;
             }
       }

       strcpy(currentEdge, pPath+pathLength-TUPLE_SIZE);
       }
       outputAndCleanTraversal();
}
```

Code Listing 3: Euler path traversal

A.4 Coverage Restoration

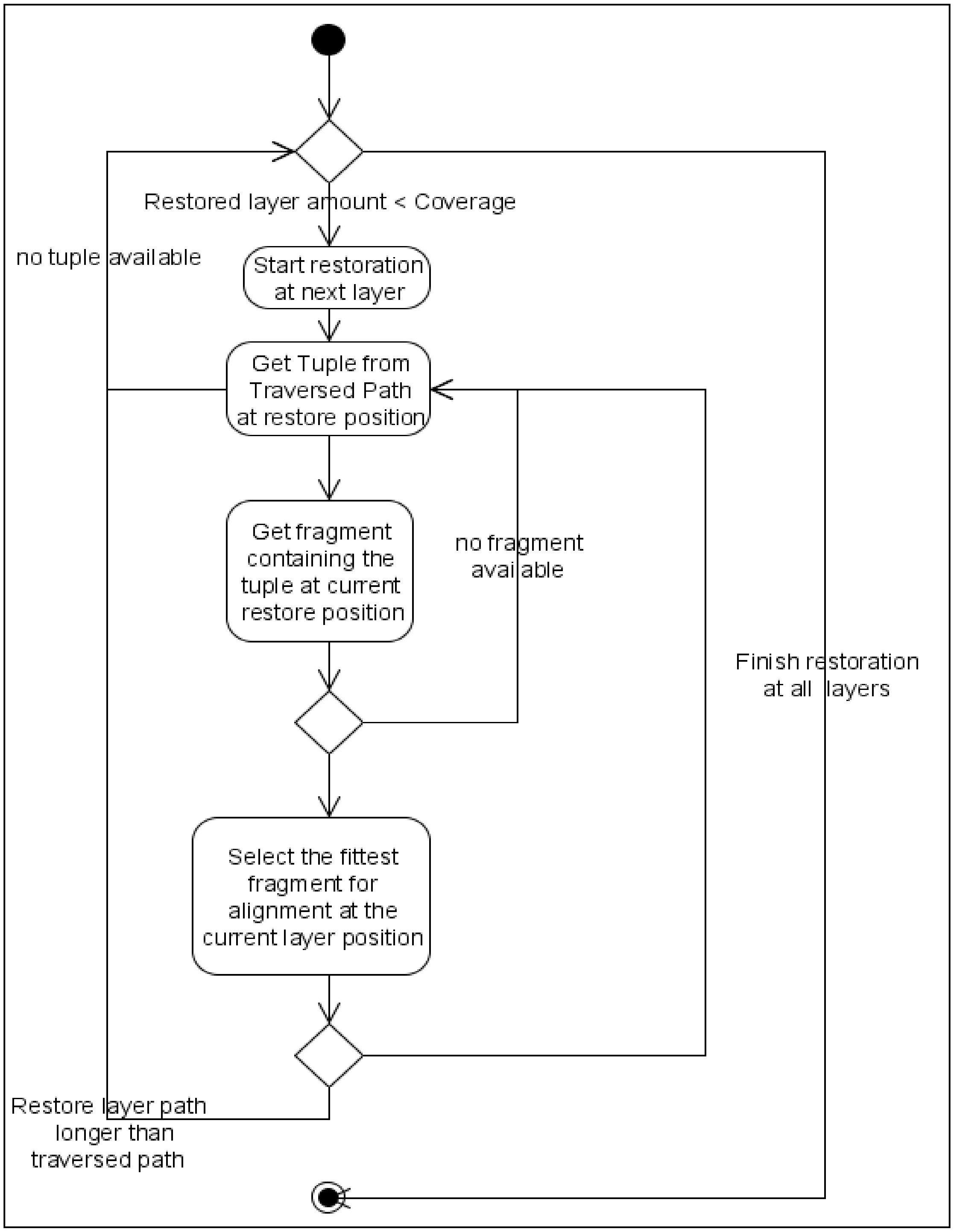
Restored layer amount < Coverage
no tuple available
Start restoration at next layer
Get Tuple from Traversed Path at restore position
Get fragment containing the tuple at current restore position
no fragment available
Finish restoration at all layers
Select the fittest fragment for alignment at the current layer position
Restore layer path longer than traversed path

Figure 51: Activity diagram for Coverage Restoration

```
int Restorer::restoreCoverage()
{
    for (int i = 0; i < 10; i++)
    {
    //trySubPathStartIndex at the beginning is the same as the current coverage layer
length or where a new fragment should be appended to continue fragment restoration
    // if no fitting fragment is found, subPathStartIndex will keep increasing until
meeting the expectLength
        int trySubPathStartIndex;
        trySubPathStartIndex = layerRestoreStarts[i];
        // restoration for one layer is done when the actual coveragePathSize of the
        layer is longer than the expectLength (the best we can expect)
        // or trySubPathStartIndex is longer than the expectLength (cannot continue
trying)
        while ((layerSizes[i] < pathLength)&&(trySubPathStartIndex < pathLength))
        {
             layerRestoreStarts[i] = layerSizes[i];
             char tryTuple[TUPLE_SIZE+1];
             strncpy(tryTuple, pPath + trySubPathStartIndex, TUPLE_SIZE);
             tryTuple[TUPLE_SIZE]='\0';
             char fragmentIDQuery[500];
             sprintf(fragmentIDQuery, "select FragmentIDs from debruijntable where Tuple =
'%s'", tryTuple);

             int resultSize;
             char **strResults = mysql_processor.queryStrFromDB(fragmentIDQuery,
&resultSize, 1000);
             char fragmentCodeIDQuery[500];
             sprintf(fragmentCodeIDQuery, "select Fragment, FragmentID from fragmenttable
where FragmentID in (%s) and Fragment regexp '^%s.*' and Consumed=0", strResults[0],
tryTuple);
             free(strResults);

             STR_INT **strintResults =
mysql_processor.queryStrIntFromDB(fragmentCodeIDQuery, &resultSize, 1000);
             int alignScore = alignFittestFragment(strintResults, resultSize, i,
trySubPathStartIndex);
             if ((alignScore > 0)&&(layerSizes[i]>trySubPathStartIndex))
             {
                 trySubPathStartIndex = layerSizes[i]; // need to continue restoration
             }
             else
             {
                 trySubPathStartIndex++; // no matching fragment to restore at the current
             position; try next index
             }
         }
     }

     for (int j = 0; j < 10; j++)
     {
// as long as 1 layer can be restored to longer than the expect length, restoration
succeeded
             if(layerSizes[j] > pathLength)
                         return 1;
     }

     return 0;
}
```

Code Listing 4: Restoring coverage before branch selection

A.5 Selecting the Best Fragment for Alignment

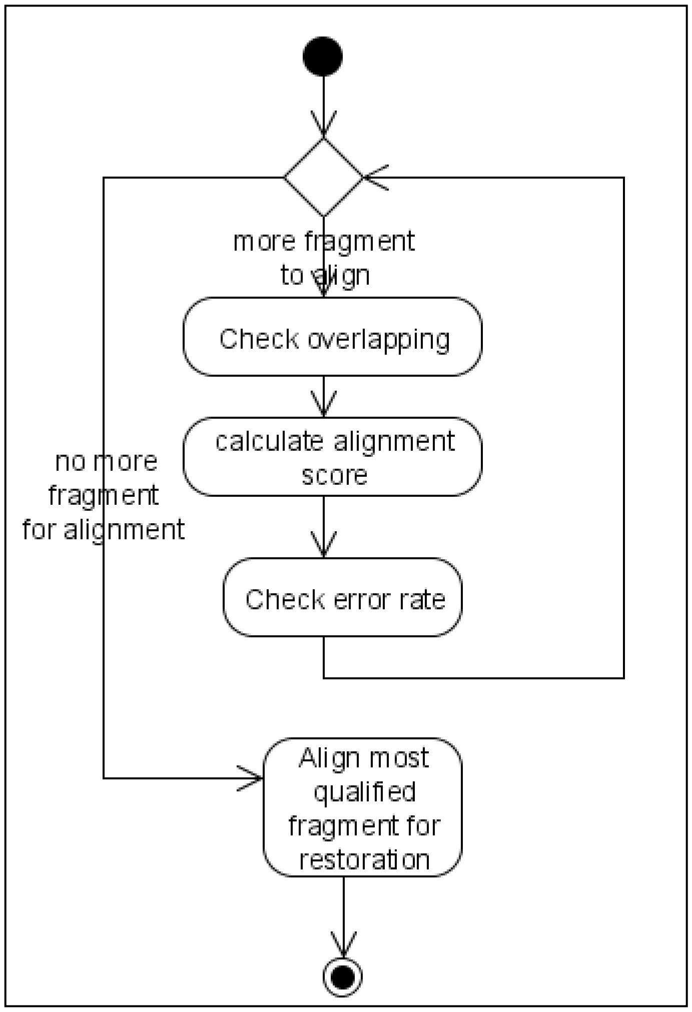
more fragment to align
Check overlapping
calculate alignment score
no more fragment for alignment
Check error rate
Align most qualified fragment for restoration

Figure 52: Activity diagram for best fragment alignment

```
alignFittestFragment(STR_INT **strintResults,int strAmount,int coverageLayerIndex,int
subPathStartIndex)
{
   for (int i = 0; i < strAmount; i++)
   {
      if (maxSubPathLength >= fragLength + 6)
      {
        alignmentLength = fragLength + 6;
        strncpy(pSubPath, pPath + subPathStartIndex, alignmentLength);
        // if mode == 1, alignment will insert space into only fragment
        tmpAlignOutput = alignFragments(pFragment, pSubPath, fragLength, alignmentLength,
&tmpScore, &insertAmount, &agreeAmount, &diffAmount, &removeAmount, diffLocations, 1,
&tmpOffset);
        int actualScore = tmpScore - 6 * insertionScore;
        if ((checkConsumable(fragLength, insertAmount - 6, agreeAmount, diffAmount,
removeAmount, diffLocations) == 1)&&(actualScore > highestScore))
        {
            alignmentOutput = tmpAlignOutput;
            bestAlignmentLength = alignmentLength;
            fittestFragmentID = fragmentID;
            exceeded = 0;
            highestScore = actualScore;
            bestIndex = -1;
            bestOffset = tmpOffset;
        }
      }
      else if (maxSubPathLength >= fragLength)
      {
        alignmentLength = maxSubPathLength;
        strncpy(pSubPath, pPath + subPathStartIndex, alignmentLength);
        // if mode == 1, alignment will insert space into only fragment
        tmpAlignOutput = alignFragments(pFragment, pSubPath, fragLength, alignmentLength,
&tmpScore, &insertAmount, &agreeAmount, &diffAmount, &removeAmount, diffLocations, 1,
&tmpOffset);
        int actualScore = tmpScore - 6 * insertionScore;
        if ((checkConsumable(fragLength, insertAmount - (maxSubPathLength - fragLength),
agreeAmount, diffAmount, removeAmount, diffLocations) == 1)&&(actualScore >
highestScore))
        {
            alignmentOutput = tmpAlignOutput;
            bestAlignmentLength = alignmentLength;
            fittestFragmentID = fragmentID;
            exceeded = 0;
            highestScore = actualScore;
            bestIndex = -1;
            bestOffset = tmpOffset;
        }
      }
      else if (fragLength > maxSubPathLength + 6)
      {
        alignmentLength = maxSubPathLength + 6;
        strncpy(pSubPath, pPath + subPathStartIndex, maxSubPathLength);
        strcpy(remain, pFragment+alignmentLength);
        // fragment is the blue blueprint in this case -- the key is to make best
alignment and get the char right after the last alignment
        // if mode == 0, alignment will insert space into both blueprint and fragment
        tmpAlignOutput = alignFragments(pSubPath, pFragment, maxSubPathLength,
alignmentLength, &tmpScore, &insertAmount, &agreeAmount, &diffAmount, &removeAmount,
diffLocations, 0, &tmpOffset);
        int actualScore = tmpScore - 6 * insertionScore;
        if ((checkConsumable(maxSubPathLength, insertAmount - (fragLength-
maxSubPathLength), agreeAmount, diffAmount, removeAmount, diffLocations) ==
1)&&(actualScore > highestScore))
        {
            alignmentOutput = tmpAlignOutput;
            bestAlignmentLength = alignmentLength;
            fittestFragmentID = fragmentID;
            exceeded = 1;
            bestIndex = i;
            strcpy(bestRemain, remain);
            highestScore = actualScore;
            bestOffset = tmpOffset;
        }
      }
      else
      {
            alignmentLength = fragLength;
            strncpy(pSubPath, pPath + subPathStartIndex, maxSubPathLength);
            // fragment is the blue blueprint in this case -- the key is to make best
alignment and get the char right after the last alignment
            // if mode == 2, alignment will insert space into only blueprint
            tmpAlignOutput = alignFragments(pSubPath, pFragment, maxSubPathLength,
fragLength, &tmpScore, &insertAmount, &agreeAmount, &diffAmount, &removeAmount,
diffLocations, 0, &tmpOffset);
```

```
            int actualScore = tmpScore - (fragLength - maxSubPathLength) * insertionScore;
            if ((checkConsumable(maxSubPathLength, insertAmount - (fragLength-
maxSubPathLength), agreeAmount, diffAmount, removeAmount, diffLocations) ==
1)&&(actualScore > highestScore))
            {
                alignmentOutput = tmpAlignOutput;
                bestAlignmentLength = alignmentLength;
                fittestFragmentID = fragmentID;
                exceeded = 1;
                highestScore = actualScore;
                bestIndex = -1;
                bestOffset = tmpOffset;
            }
          }
        }

        if(highestScore > 0) // only align fragments that are good  match for current
position
        {
            int indexOfLastChar = bestAlignmentLength-1;
            int skipcount = 0;
            int insertcount = 0;

            if (exceeded == 0) //pathLength longer than layer length, so space the at the
end of alignment for layer restoration to remove first
            {
                while (alignmentOutput[0][indexOfLastChar] == ' ')
                {
                        indexOfLastChar--;
                }

                for (int k = 0; k <= indexOfLastChar; k++)
                {
                        pCoverageRestoration[coverageLayerIndex][subPathStartIndex + k] =
alignmentOutput[0][k+bestOffset];
                }

                char updateQuery[300];
                sprintf(updateQuery, "update FragmentTable set Consumed = 1 where
    FragmentID = %d", fittestFragmentID);
                mysql_processor.updateTable(updateQuery);
            }
            else //layer length is longer than pathLength, layer becomes the blueprint
            {
                for (int p = 0; p < maxSubPathLength; p++)
                {
                    if (alignmentOutput[0][p+bestOffset] != ' ')
                    {
                        pCoverageRestoration[coverageLayerIndex][subPathStartIndex + p -
skipcount] = alignmentOutput[1][p+bestOffset];
                    }
                    else
                        skipcount++;

                    if (alignmentOutput[1][p+bestOffset] == ' ')
                        insertcount++;
                }

                    for (int k = maxSubPathLength; k <= indexOfLastChar+insertcount; k++)
                    {
                        pCoverageRestoration[coverageLayerIndex][subPathStartIndex + k -
            skipcount] = alignmentOutput[1][k+bestOffset];
                    }
            }

            layerSizes[coverageLayerIndex] = subPathStartIndex + indexOfLastChar + 1 -
skipcount + insertcount;
```

```
            if (bestIndex >=0)
            {
               strcpy(pCoverageRestoration[coverageLayerIndex] +
            layerSizes[coverageLayerIndex], bestRemain);
               layerSizes[coverageLayerIndex] =
   strlen(pCoverageRestoration[coverageLayerIndex]);
            }
         }

         return highestScore;
}
```

Code Listing 5: Selecting the fittest fragment for alignment at a position