Undergraduate Research Opportunity Program
(UROP) Project Report

Error Correction of Reads in DNA Fragment Assembly

By

Zheng Jia

Department of Computer Science

School of Computing

National University of Singapore

2006/2006

Undergraduate Research Opportunity Program
(UROP) Project Report

Error Correction of Reads in DNA Fragment Assembly

By

Zheng Jia

Department of Computer Science

School of Computing

National University of Singapore

2006/2006

Project No: U038230
Advisor: A/P Leong Hon Wai, School of Computing, NUS
Deliverables:
    Report: 1 Volume

Abstract

Pevzner et al., 2001 introduced a new DNA fragment assembly algorithm based on Eulerian Superpath transformations. This approach is sensitive to sequencing errors. We examine existing error correction algorithms in their paper, and propose a novel scheme based on spectral alignment which utilizes transformed de Buijn graph to redo error correction. The first-time error correction uses local coverages estimated with reads shared by neighboring tuples. Additional techniques are used to speed up spectral alignment. Results show that with simulated data the new scheme leaves significantly fewer mistakes than that used in EULER, and with real data from *TARSIUS SYRICHTA* Euler is now able to produce fewer contigs. In addition, the new scheme is fast enough for genome with 10^6 bases sampled at coverage 10.

Subject Descriptors:

F.2.2 Nonnumerical Algorithms and Problems
G.2.1 Combinatorics
G.1.6 Optimization
J.3 Life and Medical Sciences

Keywords:

error correction, Eulerian superpath, de Bruijn graph, fragment assembly

Implementation Software and Hardware:

FreeBSD 6.1, g++ 3.4, 32-bit Intel PC

Acknowledgement

I would like to express my gratitude to Hon Wai for his unreserved encouragement and guidance. I appreciate being part of the RAS Group for all the Tuesday afternoon discussion sessions, and during when the opportunities to present my progress. Having little background in computational biology when started a year back, I'm also grateful to researchers in the field for their lectures and advice: Ken Sung, Limsoon. Lastly, my parents, for their love and sacrifice over the months.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Since gel-electrophoretic procedures were developed (MaxamG77SangerNC77) in the mid 1970's, sequencing several hundred nucleotides of a DNA strand became possible. With present-day technologies in the gene industry, 1000 or more nucleotides (bases) can be directly interpreted, yet its length is still many orders of magnitude below that of many DNA sequences scientists are now ambitious to know.

Ideally, we would want an automated process capable of sequencing genome several million bases long. Fragment assembly (reconstruction from randomly sampled fragments) is arguably the central solution to quests like this. It is sometimes synonymous to *shotgun* strategy, devised by Sanger et al(SangerCH82).

Typically, the genome under investigation is cloned multiple times, then broken down at random positions. Part of each fragment, usually the initial portion, can be read (possibly with error) for some hundred consecutive bases, depending on the technology. These are called reads. Averagely each position would appear in X reads. X, or the coverage, is the redundancy that makes assembly possible at all.

Still, insufficient coverage due to insufficient sampling, biological bias and various sequencing errors may make the already difficult computational problem impossible to solve; i.e. computation-wise no good answer can be given. This problem became increasingly severe as

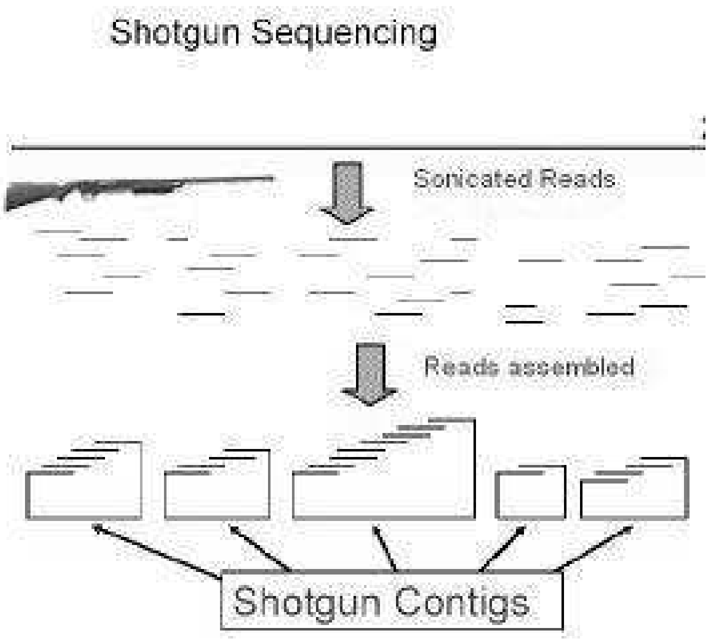

Figure 1.1: Shotgun Strategy

the ratio of DNA length to read length increases. Scientists typically need to perform directed (e.g. PCR experiments), rather than random (as in shotgun strategy), sampling to "fill the gaps" left by assembly programs, something referred to as the finishing step. Such steps are very labor-expensive and time-consuming.

Over the years, better sequencing techniques that alleviate the situation have appeared, the simplest of which is to just increase length of the fragment read. Another technology provides "double-barreled" data by reads both ends of a fragment whose length can be estimated. In a sense it doubles the read length, and is often used in finishing steps to resolve ambiguities or cover up gaps.

1.2 Fragment Assembly Definition

Like many other problems in computational biology, a mathematical definition that is biologically accurate and complete is difficult (Myers95). From the viewpoint of a student of computer science, I prefer a clear, pure definition for the design of assembly algorithm. However, for the algorithm to work on real data, a number of knotty biological complications must be tackled carefully. Let us then consider two versions of the fragment assembly problem: one that is pure and clear, one that is biologically complete and accurate.

Below is a definition similar to that given by (Myers95), which considers the "pure" version of fragment assembly.

Definition 1.2.1. *Fragment Assembly Problem* is to determine the sequence of bases in an unknown target DNA duplex Q. Q is a string over alphabet $\{A, C, G, T\}$. The set S of input reads are the output of the below (sequencing) machine

```
FOR i = 1 to some_number DO
    f := a randomly chosen fragment of S (uniformly random for start and end positions)
    r := a substring of f
    FOR each position j of r DO
        introduce error (insertion/deletion/mutation) to r at j, with probability e
    END
    output r or reverse-complete of r, with equal probability
END
```

Error rate e is a small number from 1% to 5%. Furthermore, the 3 types of changes have fixed probability each.

If each position is covered by an expected X reads, we call X the *input coverage*.

The real-life version of the problem involves specializations or complications in at least these ways.

1. Read length between 500 and 750 (with electrophoretic sequencing); with new technologies it can range below 500 or above 750, possibly with different coverages

2. With electrophoretic sequencing, coverage is 8–10; newer technologies allow higher coverage (e.g. 30)

3. Q contains repeats. In some eukaryotic genome over 90% are repeats

4. The (sequencing) machine also introduces contamination, vector and chimeric reads. Contaminated reads contain garbage information. Vectors are DNA introduced from unrelated genome during experiments. A chimeric read refers to two (or more) reads glued together, giving a bogus read $s = s_1 s_2$

5. Some part of genome is not amenable to cloning or uncovered in any read.

So, our objective is perhaps more aptly put as to find several contiguous substrings of Q, or "contigs". In general, I believe, we should try to minimize the number of contigs rather than make it match some expected number, since

1. there lacks reliable estimation of the expected number of clonable regions

2. contigs produced by assemblers are more likely to result from misassembly then real contigs

1.3 Fragment Assembly Formulations Review

In the early studies of computational biology, the layout stage was often formulated as the Shortest Common Superstring problem:

Definition 1.3.1. *Shortest Common Superstring (SCS)* Given a set of strings $S = \{s_1, \ldots, s_n\}$, find the shortest string s that includes each s_i as substring.

Let alone the NP-complete nature of this computational problem, the formulation has several deadly drawbacks even assuming error-free reads. For example, reads in exact repeats are always considered redundancy (cloning) of the same region as it favors *only* parsimony. Overcompression makes it infeasible even for small-sized genome.

(Myers95) contains a formulation on solid theoretical ground, in which the objective is to find maximal likelihood reconstruction with respect to the 2-sided Kolmogorov-Smirnov distribution. However there lacks an efficient algorithm for this formulation.

For the last two decades of 20th century, fragment assembly in DNA sequencing followed the "overlap-layout-consensus" paradigm (PevznerTW01). In the overlap step, reads are checked pair-wise for overlap (i.e. the longest prefix that matches a suffix of the other, allowing some error). The layout stage, arguably the hardest, attempts to put each read back to the original, unknown sequence, while satisfying the overlaps. Finally, since overlaps are “approximate” due to errors in reads, a consensus is needed among conflicting bases.

The classical approach culminated in some excellent fragment assembly tools (Phrap, CAP3, TIGR, Celera)(?HuangM99SuttonWAK95?). They work well with non-repetitive sequences, but fail for even eukaryotic genome containing repeats. Repeats are notoriously hard to deal with in fragments assembly, especially when they are long. To avoid the mistake of SCS, popular assemblers mask repeats. This, however, results in too many contigs or misassembly.

1.4 Eulerian Superpath Formulation

In 2001 Pevzner, Tang, Waterman (PevznerTW01) proposed a new approach based on finding Eulerian superpath, one that abandons the traditional overlap-layout-consensus paradigm. They developed the EULER assembly package, with components including EULER-EC (error correction), EULER-DB (catering to double-barreled data), EULER-SF (scaffolding). Their work can be viewed as continuation of Idury and Waterman’s idea (IduryW95) in 1995, which in turn drew inspiration from the SBH (Sequencing By Hybridization) algorithm proposed by Pevzner. (Pevzner89)

Definition 1.4.1. *Sequencing By Hybridization* Given a set of l-tuples $S = \{s_1, \ldots, s_n\}$, find the shortest string s such that every l-tuple from s is in Q, and every l-tuple from Q is in s.

Initially it was modeled as a shortest Hamiltonian path problem following the overlap-layout-consensus paradigm. Despite apparent similarity to SCS, the complexity of SBH is dramatically different. If we make each $(l-1)$-tuple as a vertex, each l-tuple as an edge, the Eulerian path solution reduce the running time to polynomial.

Idury and Waterman suggested mimicking fragment assembly as SBH, cutting each read of length n into a collection of $n-l+1$ l-tuples. The loss of information that which l-tuples belong to the same read can be recovered later. However, this approach does not answer two important questions: (a) errors in reads transform the de-Bruijn graph into a mess of erroneous edges (b) repeats, which make the graph much tangled, are not dealt with. That motivated EULER.

Defines a read path to be the path in G that corresponds to the read, and $P = \{\text{read paths}\}$. The Eulerian superpath algorithm make a series of equivalent transformations (i.e. there exists one-to-one correspondence between superpaths in (G, P) and (G_1, P_1)). It tries to match every read to what (subset of reads) follows it in the target DNA.

$$(G, P) \to (G_1, P_1) \to \ldots \to (G_k, P_k)$$

Unlike other assemblers, no heuristic guesses or assumptions are made about the genome. Repeats are implicit in the graph, and any unresolvable repeat is only due to insufficient information instead of pitfall of the computation model. Details of superpath transformations used is discussed in next chapter, where Bruijn graph transformations is used for error correction.

Due to sensitivity to errors, error correction must be done beforehand with a greedy procedure, unlike traditional assemblers which do so at the consensus step.

1.5 Problem and Solution

The Eulerian Superpath approach is sensitive to errors, but the current error correction algorithm, although efficient, does not work well. EULER produces significantly more contigs than Phrap with low coverage data (PevznerTT04).

Pevzner et al. introduced a Spectral Alignment approach to model error correction. In another paper (ChaissonPT04) Chaisson and Tang gave a dynamic programming formulation for the Spectral Alignment Problem. However it suffers from both inefficiency and over-rigid thresholding.

This study introduces a novel error correction scheme, which (i) finds similar reads to estimate coverage in the initial error correction, (ii) uses output from EULER to re-do error correction, (iii) is based on Spectral Alignment. Hence a scheme for EULER fragment assembly that performs error correction twice is proposed.

Chapter 2

Error Correction

Error correction replaces each read with a possibly error-free read. It can be formulated in various ways. The idea of *cover set* and *coverage*, however, is central in all formulations.

Definition 2.0.1. *Cover Set and Coverage.* Let the substring $s[i \dots j]$ of s, $s \in S$, be mapped to $Q[i' \dots j']$ in the correct assembly.

Cover set of (s, i, j) is the set of reads including s that, in the correct assembly, cover $Q[i' \dots j']$.

Coverage of (s, i, j) is the cardinality of the cover set, or intuitively, how many reads cover a given region. Define also the coverage at k of s to be coverage of (s, k, k)

First, for each read s, both s or $\bar{s}$ (reverse complement) can be in the actual fragment. Let's augment the set of reads S to include all reverse complements of input.

Below is one formulation based on *multiple alignment* at each read.

Definition 2.0.2. Error Correction as Multiple Alignments For each read s, do multiple alignment of the cover set of $(s, 1, s.length)$. If s is changed to s' after alignment, replace s with s'.

In this formulation, two reads which are neighbors in the correct assembly are likely to be aligned in a compatible way, since their cover sets do not differ much. We favor parsimony (fewest corrections) by the same argument as we favor fewest contigs.

Ignoring complexity of multiple alignment, this error correction scheme seems ideal, if we can find the cover set of each $(s, 1, s.length)$. It makes none of the types of mistakes of the existing algorithms. In fact, I would like to use this as a reference formulation, and propose one which makes just as few mistakes (or fewer), with lower time complexity feasible for real projects.

2.1 Existing Algorithms

2.1.1 Spectral Alignment Formulation

(PevznerTW01) introduced the Spectral Alignment formulation for error correction in fragment assembly. (ChaissonPT04) contains a DP formulation.

Let us assume that Q (the genome) is known, hence all its l-tuples, T. Error correction can be done by aligning each read $s \in S$ to T, such that every l-tuple of s is in T. Call such strings T-strings. We favor parsimony by the same argument as minimizing number of contigs; that is, with fewest changes to s.

Definition 2.1.1. *Spectral Alignment Problem* Given a string s and a spectrum T, find the minimum number of corrections (insertion,deletion,mutation) that transform s into a T-string. If a string cannot be fixed in Δ corrections, or multiple solutions exist (ambiguity), the string is discarded.

Of course, the assumption that Q is known is a catch-22: to obtain it we need error correction first. Nonetheless, estimation of T is possible. Let frequency $freq(t)$ of the l-tuple t be the number of reads in which it appears. Pevzner suggests using a threshould frequency M, and let T be just those with frequency at least M ("strong" tuples; otherwise "weak" tuple).

Intuitively, a high frequency tuple is likely error-free. For an exception to occur, we need roughly M l-tuples to contain a same error at a same position. An erroneous l-tuple, unless the mutation makes it coincide with a correct l-tuple, is likely to be a low frequency one. On the other hand, a low frequency tuple may not contain error at all – coverages differ over the genome.

To find fewest corrections for each read s, the DP formulation defines function $score(i, a)$, which represents the sub-problem on $(s[1 \ldots i], a)$, with a being the last l-tuple of what $s[1 \ldots i]$ should be aligned to. That is, $score(1, a)$ is the least number of corrections to $s[1 \ldots i]$ to make it a T-string whose last l-tuple is a. Now use the recurrence relations

$$score(i,a) = min(X \in \{\text{A,C,G,T}\}) \begin{cases} score(i-1, X \circ \overrightarrow{a}) + \begin{cases} 1, \text{if } a_l \neq s_l \\ 0, \text{otherwise} \end{cases} \\ score(i-1, a) + 1 \\ score(i, X \circ \overrightarrow{a}) + 1 \end{cases}$$

where $X \circ \overrightarrow{a}$ denotes the l-tuple formed using X and first $l-1$ symbols of a. The first case corresponds to matching or mismatching, second for deletion, and the third for insertion.

This DP formulation is perhaps a little misnomer – if we build the graph where vertices are (i, a) pairs and edges represent dependency, it is cyclic due to insertion. (ChaissonPT04) suggests using shortest path algorithm such as Bellman-Ford. Since the edge costs are small integers (0 and 1 in this particular scoring function), a simple breadth-first search with at most some basic transformation will reduce it to $O(n)$ where n is the number of vertices, i.e. possible (i, a) pairs.

Although not explicitly mentioned in their paper, choice of M at least depends on (i) average coverage X (ii) e.

Due to inefficiency of Spectral Alignment, it has never been implemented for EULER. However, we have implementated the algorithm for comparison against our scheme.

2.1.2 Greedy Error Correction

(PevznerTW01) gives a different formulation suitable for a greedy heuristic, which is adopted in EULER. Define S_l to be the set of all l-tuples in the set of reads S.

Definition 2.1.2. *Error Correction Problem* Given S, Δ, l, introduce up to Δ corrections in each read in S in such a way that $\|S_l\|$ is minimized.

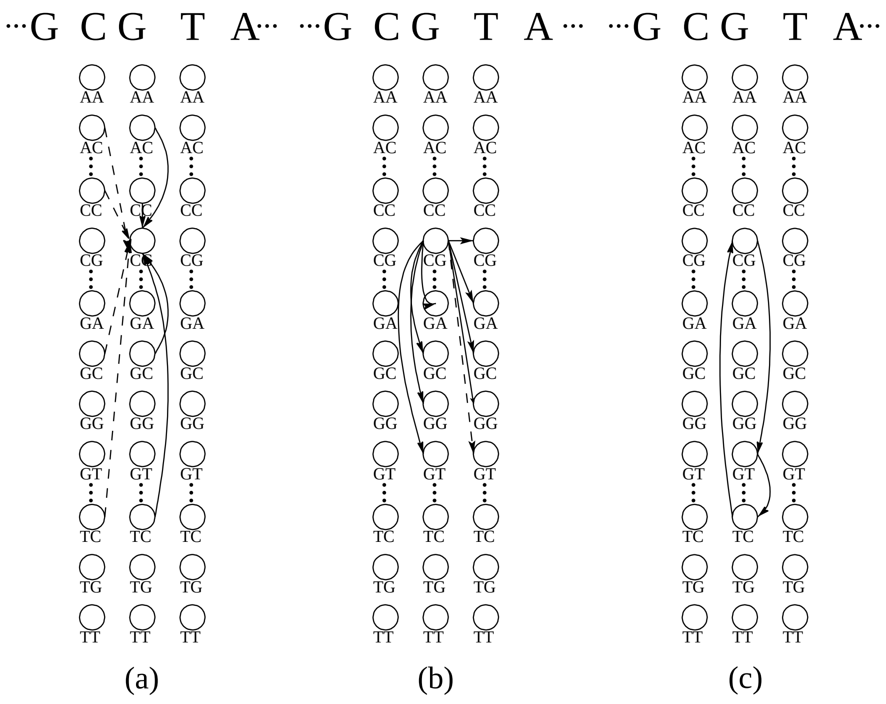

Figure 2.1: Graph Representing Recurrence Dependency. Dashed edges have score 0 and solid edge 1. (a) Vertices used to compute the score of CG. (b) Vertices that depend on CG. (c) A cycle (always positive). Copied from [ChaissonPT04]

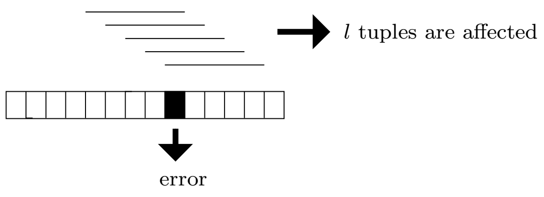

Figure 2.2: One mutation brings error to l consecutive tuples ($2l$ including their reverse complements)

Their greedy heuristic relies on assumption that, if an l-tuple contains an error, it brings error to the $l-1$ tuples adjacent to it and the reverse complement of each, in the read. That could hopefully be detected. To describe this method, call l-tuples $t1$ and $t2$ *neighbors* if they are a *single* mutation apart. Call an l-tuple i *orphan* if (i) $freq(i) < M$ (ii) it has exactly one neighbor which is not orphan.

The greedy approach first corrects the orphan with a *single* mutation such that number of orphans will be reduced by $2l$ (or $2d$ if the tuple starts at offset d and $d < l$). Since $2l$ is too restrictive (imagine that one of the $2l$ tuples happens to coincide with a non-orphan tuple even though there is an error in it), decrease $2l$ after each scan until it drops to $2l - \delta$.

2.2 Tuple Length

Choice of l impacts the scheme enormously. Let us first use symmetric Bernoulli model with multiple outcomes for generation of Q. In the symmetric Bernoulli process, outcomes of different trials have the same probability distribution.

The probability that t occurs in Q depends on the auto-correlation of t, which measures how much it overlaps itself. Define the auto-correlation of t to be the binary string c_t:

$$c_t[i] = 1(\text{if } t[1 \dots i] = t[t.length + 1 - i \dots t.length]), \text{or } 0 \text{ otherwise}$$

(Mansson02) shows that tuple t' has a higher chance to occur than t if and only if $c_{t'} > c_t$ lexicographically, as long as length of Q is at least $2l - 1$.

(Guibas81) contains a formula for number of strings of a fixed length that contain a given pattern. The probability that an l-tuple t occurs in Q r times is better answered in (RegSzp98b), in which overlapping occurrences are counted separately. The author gives the moment generating function for the number of strings of a given length which contains a given pattern.

The expected number of t in Q is the sum of expectations of individual l-tuples: $1/4^l \times Q.length$. With Q being 10^6, a suitable value for l would be at least 15. With $l = 14$, the expectation is roughly $1/4^4$, implying a duplicate for about each 256 unique l-tuples, way too many for $Q = 10^6$.

Errors may turn distinct l-tuples into duplicates. The chance also decreases with larger l. The probability that two l-tuples with k different symbols turn into duplicates due to errors is

$$(1-e)^k e^k ((1-e)^2 + e^2)^{l-k}$$

Given these two observations, a larger l implies exponentially fewer duplicates. Since duplicate l-tuples in Q lead to mistakes in error correction, a larger l has the benefit of much fewer such mistakes. A main drawback is the reduced number of l-tuples that cover the *same* position in Q in the correct assembly (consider the extreme case when l is the length of a read). Another is the larger search space for l-tuples. In spectral alignment for example, that implies increased time and space complexity.

In practice, $l = 20$ for $Q = 10^6$ is reasonable, since the expected number of duplicates is now at most 1.

In actual genome, Q is never generated with a Bernoulli process. Exact and non-exact repeats both contribute to much more duplicate l-tuples over Q. This is what makes error correction difficult, and is the purpose of the following sections.

2.3 First Time Error Correction

Both the SAP and the greedy solution have problems of false positives and false negatives. Coverage in different parts of the genome can vary between 0 and kX for some $k > 1$. A low coverage region will very unlikely contain tuples with frequency above M. Using a fixed threshold is unfair in this sense, as frequencies of l-tuples are not compared on the same ground. Repeats make things even worse. Consider a repeat R which appears at 3 different positions

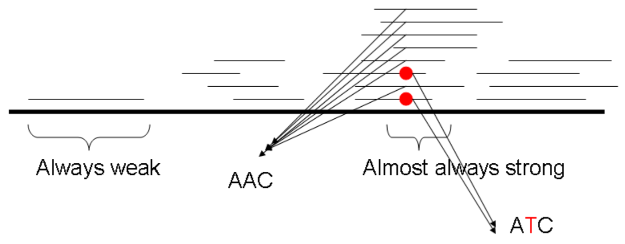

Figure 2.3: With fixed threshold 2. Red dots are error tuples.

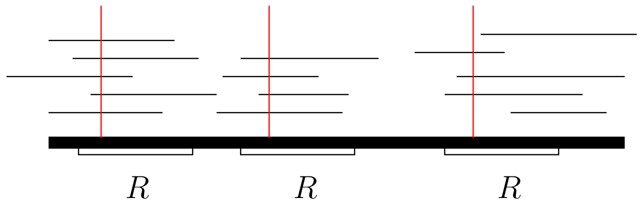

Figure 2.4: A repeat R which appears at 3 different positions in the genome

in the genome. Each l-tuple in R will have frequency roughly 3 times higher than l-tuples in non-repeat regions.

Thus it is unwise to use the same threshold M. Knowing coverage at all positions on the target DNA is important for a "fair" error correction algorithm. Ideally, the ratio between coverage and *local frequency* should be used to measure the likelihood whether a tuple needs correction. For tuple t, define local frequency of t to be the number of times t occurs in its cover set.

Computing local frequency is computational costly. We approximate it with (global) frequency. This is based on the observation that, most often, (global) frequency is higher than local frequency only if t is correct (that is, due to repeats), in which case over-estimation does not matter anyway.

Having identified the tuple t that needs correction, we must decide which set of tuples t can be corrected into. Our algorithm uses all (global) strong tuples (with "strong" defined later), similar to Spectral Alignment. It is tempting to use *local strong tuples*, those strong tuples in the cover set of t. This is again computationally costly, as one must recompute the tuple pool for different cover sets. On the other hand, choosing all (global) strong tuples will not in general

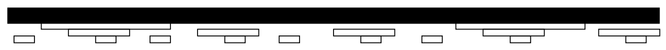

Figure 2.5: Repeats of different lengths containing the same l-tuple

give worse alignments, since only *local strong tuples* will minimize the number of corrections.

Finding coverage will be dealt with in the sub-sections. Assuming that coverage information is available, First Time Error Correction uses the frequency–coverage ratio to decide whether a tuple needs correction. This ratio is compared against some parameter $\gamma \leq 1/2$.

Tuples with same content can occur at multiple places, giving different frequency–converage ratios. Thus the set of "strong tuples", T, is defined to contain any tuple that has ratio $\geq \gamma$ for at least one occurrence.

A few other optimization techniques are discussed in later sections.

2.3.1 Approximating Coverage from Similar Reads

Taken literally, a "repeat" can be long or short. In this notion, we can view any substring in a read as a repeat. To find the true coverage, we should filter out as many repeats as possible. Since each long repeat contains a short repeat (see figure below), we try to filter out as many shorter repeats as possible as it is easier. Not being able to filter long repeats, which is often the case, is not that disastrous, as we shall see later this section.

Let $\mathit{copy}(s[i \ldots j])$ be the *repeat count* for substring $s[i \ldots j]$ in read s: the number of times the substring (after error correction) appears in Q. Let $\mathit{similar}(s[i \ldots j])$ be the number of *reads* "similar" to $s[i \ldots j]$, with "similar" defined in terms of edit distance (but ignoring penalties for start and end spaces for the two sequences respectively). For each position k of s, if we take a substring ss of s with a suitable length and is centered at k, then $\mathit{similar}(ss)/\mathit{copy}(ss)$ will give an approximation for the coverage at k, assuming no errors exist in all reads that actually cover k.

However, at the error correction stage little can be done to figure out *copy* of a substring of a read. The best one can do is perhaps to find excessively many similar reads (e.g. 80 when

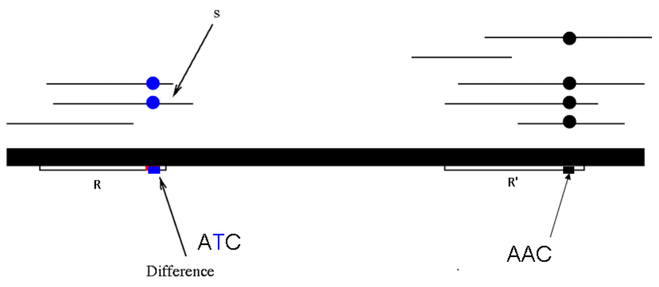

Figure 2.6: Non-exact repeats with different coverages cause problems

$X = 10$) and decide *copy* to be $80/X$. It doesn't help us here, since $similar(ss)/copy(ss)$ will now always give expected coverage at any position.

An important observation is that, if the substring ss we take falls in exact repeats (i.e. exact copies of the same sequence), we can use $similar(ss)$ instead of the actual coverage at ss. This is true because the repeats are exact: whatever is the majority in *all* repeats should be “strong”, and others “weak”.

The only problematic situation is when repeats are non-exact (due to mutations in evolution), and have different coverages at the symbols where they differ. This situation includes repeat borders, if we view $ATGTC$ and $ATGTT$ as non-exact repeats. Since we cannot distinguish non-exact repeats when estimating coverage (we need error tolerance), the estimated coverage will be the sum of coverages of the corresponding regions of two repeats. Assume two non-exact copies R, R' differ at offset k. Let t and t' be l-tuples that contain position k. If they have different coverages, the tuple with lower coverage will almost always be considered “weak”, since the estimated coverage is the sum. This situation cannot be avoided without knowing *copy*. However, as we will show, by using output from EULER after initial error correction, such situations can be avoided.

Although *similar* can be found by pairwise sequence alignment, the complexity is probably prohibitive. Below we look at ways to compute $similar(ss)$, a problem similar to the repeat finding problem.

from Neighborhood Substring Frequencies

In this approach, we extend domain of $freq$ to strings: $freq(ss)$ is the number of reads that contain as substring ss. $similar(ss)$ (therefore coverage) can be estimated from $freq$.

We compute $freq(ss')$ for all substrings ss' using suffix tree. Let T be the generalized suffix tree for all reads. For every internal node v of T, $freq(ss')$ where v represents ss' can be computed as size of $\{s \in S : \exists$ a leaf node from s in the subtree rooted at $v\}$. Computing $freq$ over all internal nodes can be done in $O(n)$ with $O(n) - O(1)$ online LCA (Lowest Common Ancestor) algorithm(Hui92).

To estimate coverage at position k, we check frequencies of all βD substrings of length D whose total offset from k is minimal (i.e. centered around k). Among the frequencies, take the maximum. The reasons to pick the maximum is (i) reads that cover k may have errors around k (ii) they may cover k, but fail to cover k' where distance between k, k' is less than $\beta D/2$ (iii) with sufficiently large β, the chance for a substring to be contained in unrelated reads is minimal.

With high coverage X and long (10^6) genome, it can be desirable to use suffix array to reduce space requirement. Once the suffix array is built, with careful design, frequencies of all length D substrings can be computed in $O(n)$, based on a recent linear-time algorithm that computes Longest Common Prefix for every two neighboring entries in the suffix array(KasaiLAAP01).

from Reads shared by Neighboring Tuples

When estimating coverage from neighborhood substring frequencies, we need to keep D to be small, considering that we should not have a small $(1 - e)^D$ (chance to have no errors in the substring). Then, with moderate e (1%), D cannot be larger than 40. This quite often fails to filter most of the repeats.

In view of that, we consider the D l-tuples in the read which are closes to the l-tuple whose coverage we wish to ascertain. Initially when hashing the tuples, for each l-tuple t we maintain a linked list of reads, $LL[t]$, that contain it. Then, among all the D neighboring l-tuples, the algorithm finds the reads that are shared by βD such neighboring l-tuples.

This can be done for all l-tuples in all reads, within time $O(Xn)$, with n being the input size, if we assume constant look-up time of the hash table. Basically, we keep a sliding window of the D tuples. Associated with the window is a hash table HT of reads. When a new l-tuple t shifts into the window, we increment the count in HT for each read in $LL[t]$. If a count becomes greater than or equal to βD, increment a global counter. When an l-tuple shifts out of the window, decrement the counts and decrement the global counter if some count drops below βD. The global counter then contains the estimated coverage for all tuples as the window slides over the read.

2.3.2 Spotting Low Coverage

It is arguably more important to know that a region has low coverage region than to know it has high coverage, since such regions *will* always be error corrected even if it is perfectly error-free. On the other hand, high coverage regions may or may not contain false positives. Fortunately some simple observation enables us to identify low coverage: many of its l-tuples are weak. With this heuristic equipped, we can leave such portions of a read as it is, since it is the only information available for that region.

The probability of m errors in n consecutive symbols follows the binomial distribution with mass function $\binom{n}{m}e^m(1-e)^m$; $C(n,m)$. With m close to n the probability is negligible.

The algorithm we use masks portions of read s that have low coverage. It starts by looking for intervals of length $n+l$ (n is a program parameter) with ml weak tuples, such that the probability above is smaller than some threshold. The reason for ml is that each error affects l consecutive tuples. For each such interval, extend it in both directions until the probability will go above the threshold.

The danger is that the low coverage may be due to contamination. By considering it correct, later stages in fragment assembly may be misled. It is best to leave it an option whether to enable this heuristic, depending on contamination rate of the sequencing technology.

2.3.3 Approximating Coverage with de Buijn Graph: Read Paths

Let us look at some properties of de Buijn graph during Eulerian Superpath transformations. There are two important properties

1. For each read there is a unique path (read path) $p \in P$, although p evolves during transformation, the correspondence does not change
2. Each edge a dynamically represents a string of base pairs $\delta(e)$. In the initial de Bruijn graph, each edge represents an l-tuple.

Two types of transformations are allowed: $x-y$ *detachment* and *cut*. *cut* is relatively simple (PevznerTW01). The paper defines $x-y$ *detachment* with respect to a pair of consecutive edges. For sake of discussion, below is an equivalent definition with respect to vertex.

Let $\{a_i\}$ be vertices connected to v, and $\{b_i\}$ be vertices v is connected to. The transformation equivalent to $\{(a_i, v) - (v, bj)$ *detachment*$\}$ is as follows:

1. In each read path $p \in P$ that contains some subpath $(a_i, v)(v, b_j)$, replace the subpath with a single edge (a_i, b_j)
2. In each read path $p \in P$ whose start (or end) vertex is v, replace the first (or last) edge (v, b_j) (or (a_i, v)) with (a_i, b_j) if
   (a) any read path $\in P$ consistent with p does not contain (a_k, v) with $k \neq i$ (or (v, b_k) with $k \neq j$), and
   (b) there exists a read path $\in P$ consistent with p that contains (a_i, v) (or (v, b_i))

Two read paths are consistent if their union is a path again. Such transformation covers many situations, e.g. contraction of edges on a path without branches (similar to contraction used in planarity testing).

When replacing edges, we update $\delta(a)$ for the new edge a. We make the following observations: (i) each unique l-tuple may appear multiple times, in $\delta(a)$ of several different edges a, (ii) the start and end of a read path $p \in P$ may contain some extra base pairs not in the read.

We can infer coverage for any $(s, i, i + l - 1)$ from de Bruijn after transformations.

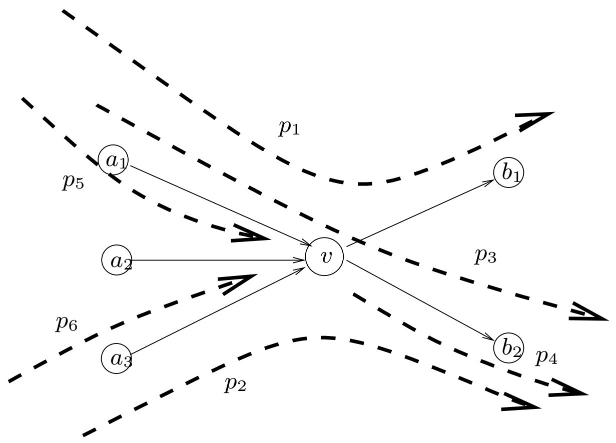

Figure 2.7: A complex situation with 6 paths. p_6 is consistent with both $p2, p4$, and can be resolved; p_5 is consistent with $p1, p3, p4$, but cannot be resolved.

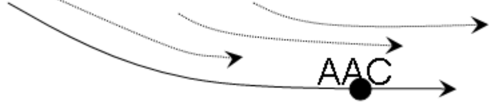

Figure 2.8: Estimating coverage by counting read paths that cover an edge. Dashed lines are read paths.

Assume first that all tangles are resolved. If some read paths share edge a, those reads *will* cover any l-tuple in a (with the exception from the second observation above, which can be checked quickly). Thus for any l-tuple on edge a of a read path, by checking how many read paths include a, we have now a more accurate estimate of coverage than First Time Error Correction, even if the de Buijn graph is based on results of First Time Error Correction.

For example, let $l = 6$, two regions $\ldots ACTTGC$ and $\ldots ACTTGA$ have coverages 1 and 3 respectively, and some read s covers $ACTTGC\ldots$. Although s is corrected into $ACTTGA\ldots$, it is with high likelihood that its read path in the de Buijn graph will be separate from those that truly cover $ACTTGC\ldots$. That is, the part of s several symbols to the right of $ACTTGA$ will not be error corrected, and hence different from those that truly cover $ACTTGC\ldots$. The new coverage estimate will be 1, making $ACTTGA$ a strong tuple. This extra accuracy comes from EULER's capability to resolve repeats.

2.4 Re-doing Error Correction

2.4.1 Approximating Coverage with de Buijn Graph: Copy Number

Things are more complicated in the presence of unresolvable tangles, which usually correspond to long repeats in target genome. If we find as above the set of reads whose read paths include a, we are over-counting. Fortunately, structure of de Buijn graph gives us hint again.

(PevznerTW01) applies Eulerian Superpath to The Copy Number Problem, which gives us a chance to estimate *copy*. The idea is to first build the de Buijn graph.

Definition 2.4.1. *Copy Number Problem* For an edge a in a graph $G(V, E)$ find a flow $G(V, E, f)$ minimizing the multiplicity $f(a)$ of the edge a. Every edge has capacity lower bound 1.

(PevznerTW01) gives a heuristic, but did not prove its correctness. The Copy Number Problem can be solved by finding a *minimal feasible flow*, taking two ends of a as source and sink.

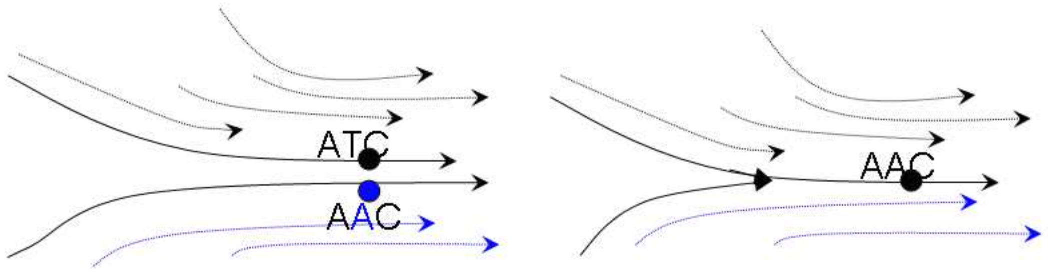

Figure 2.9: Non-exact repeat: estimated coverage reduced after de Buijn transformations. (Left) The correct de Buijn graph (Right) Incorrect de Buijn graph, but by dividing number of read paths by 2 (copy number), the new estimate becomes quite accurate. Dashed lines are read paths

A more general formulation could be to minimize sum of all flows, rather than the flow on the edge under investigation.

Definition 2.4.2. *All-Repeat Copy Number Problem* Given graph $G(V, E)$, find a flow $G(V, E, f)$ minimizing the multiplicity sum of $f(a)$ over each edge a. Every edge has capacity lower bound 1.

This can be solved as a Min-cost Circulation Problem in polynomial time. (AMO) contains a comprehensive introduction to both Min Feasible Flow and Min-cost Circulation Problem.

If we have the copy numbers, we can divide the estimated coverage by *copy* for more accurate coverage estimate. For the earlier example of non-exact repeat, now with the better coverage estimate, the lower coverage l-tuple will now be deemed strong. The figure shows when an non-exact repeat transforms the tuple with coverage 2 into the tuple with coverage 3, this false negative can be removed with new coverage estimate (in the figure, (2+3)/2). Observe that even if the lower coverage l-tuples in one repeat are changed into the strong l-tuples in the other repeat, the copy number obtained from de Buijn graph will generally still be correct, since the overall structure of $A - R' - B - R - C$ is usually preserved.

Non-exact repeats with unequal coverages and repeat borders are commonplace. The use of transformed de Buijn graph applies also to non-exact repeats that occur more than 2 times in Q, or that have more than 1 differences in an l-tuple. With new coverage estimated with copy number, we will have reduced most of the false negatives.

We compute *copy* depending on the way coverage was estimated. If we used D neighboring l-tuples to compute in which reads they appear in common, we need to map the D l-tuples in the read s to the path in de Buijn graph. This is done by finding the read path of s and searching for the subpath that contains those tuples.

2.5 More Techniques

2.5.1 Masking High-confidence Portions

To reduce time complexity of spectral alignment, it makes sense to ignore parts of the read in which we have high confidence. Do the following preprocessing on each read s

```
FOR i = 1 to s.length-l+1
    t := l-tuple at position i
    a[i] := 1 if t is strong; 0 if t is weak
END FOR
FOR each streak of consecutive 1's
    IF has length > K THEN mask this region
END FOR
```

with K being a user-supplied parameter representing the trade-off point between performance and accuracy. Smaller K introduces more greediness into alignment.

2.5.2 Dealing with Chimeric Reads

A chimeric read refers to two (or more) reads glued together, giving a bogus read $s = s_1s_2$. They may be detected and restored during spectral alignment, provided that neither of s_1, s_2 lies in a low coverage region. If from some position k, most of the l successive l-tuples all have frequency 1, and more than σl corrections are needed to align it (to the spectrum), we consider it a chimeric read and break it before position $k + l$. σ can be a number between 0.5 and 1, depending on many factors (e.g. X, l, error rate, chance to have chimeric reads).

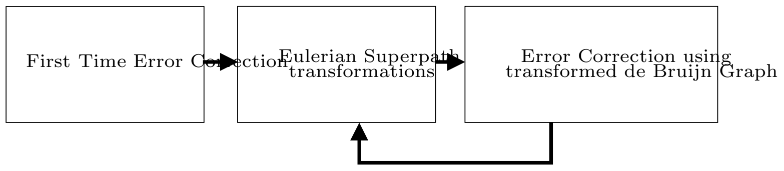
First Time Error Correction
Eulerian Superpath transformations
Error Correction using transformed de Bruijn Graph

Figure 2.10: Proposed Error Correction Scheme

2.6 Proposed Error Correction Scheme

We propose first doing error correction with coverage estimated from reads shared by neighboring tuples; having obtained the transformed de Buijn graph, re-do error correction with more information found by EULER.

When re-doing error correction, we use de Bruijn graph for 2 purposes (i) obtaining coverage as number of covering read paths (ii) estimating *copy*. Coverages are estimated now directly from de Buijn graph, eliminating the need to compute reads shared by neighboring tuples.

It may be even tempting to think that we could *iteratively refine* error correction by continuing the loop. However, as the coverage information recovered from de Buijn graph depends on the coverage in the *previous* error correction, any l-tuple which was considered "strong" (i.e. left unchanged) will continue to be so. In general, we believe that the gain is little to do error correction iteratively.

2.7 Complexity

First look at complexity of First Time Error Correction. Finding reads shared by neighboring tuples takes $O(X^2 \times Q.length)$ (shown earlier) assuming constant time hash table look-up. To apply the technique for spotting low coverage regions in each read, we need linear time to scan for such regions once the coverages have been computed. Detecting chimeric reads is done in a similar linear scanning fashion.

Spectral alignment is done based on the recurrence relations, which has a very loose upper bound $O(n\|T\|)$ for each read of length n. Since we assume at most Δ changes, and high-confidence or low-coverage portions of each read are ignored, the running time is much faster. In the test it is shown that it runs in empirically linear time.

Computations related to de Buijn graph depends on the representation of de Buijn graph used in EULER. Studying the representation is still on-going.

Chapter 3

Results

3.1 First-time Error Correction

In this section three error correction schemes are compared: Spectral Alignment from (ChaissonPT04) (SP), Greedy Error Correction (GD) (one that EULER uses), and the First-time Error Correction (FT) (as part of the new scheme). SP is implemented for comparison purposes, while GD is bundled in EULER.

Testing environment: FreeBSD 6.1 running on Intel P4 with 1024M memory.

For compatibility with GD all tests assume equal probability insertion/deletion/mutation.

For simulated data, reads are generated by picking a uniformly random starting offset, then reading L characters, where L follows a given normal distribution $N(mean, variance)$. The sequence used is a random portion of the recently sequenced *T.whipplei*, obtained from website of *Sanger Institute.*

Good parameters for FT are $\beta = 0.4$ for Test B and $\beta = 1 - 2le$ for Test A,C,D (which have error rate 1%), $\gamma = 0.5, D =$ average read length/5.

For all algorithms, Δ (maximum number of corrections before discarding a read) is set to (Read Length $\times$ Error Rate $\times$ 8), and it seems to give better results in all tests except Test F *TARSIUS SYRICHTA*.

For simulated data, the number of errors after correction is computed as the edit distance between the correct read and the corresponding read given by the program. For *TARSIUS*

Table 3.1: Test A: with $l = 10$

	SP	GD	FT
Corrections	1136	1330	913
Errors introduced	119	220	84
Errors after Correction	193	304	165
Unfixable Reads	13	10	0

Table 3.2: Test A: with $l = 20$

	SP	GD	FT
Corrections	1132	1249	959
Errors introduced	52	113	54
Errors after Correction	112	202	87
Unfixable Reads	12	13	0

SYRICHTA, this information is not available.

Both GD and SP work best when M is set to 4, interestingly for both $X = 10$ and $X = 20$. Only best results are shown.

In each test, the results are the average of 3 runs.

- Test A. With simulated reads generated using the following parameters: Error Rate = 1%, $L \sim N(600, 2500)$, Sequence Length = 100000, Number of Reads = 1600. Average number of simulated errors = 1032

- Test B. With simulated reads generated using the following parameters: Error Rate = 5%, $L \sim N(600, 2500)$, Sequence Length = 100000, Number of Reads = 1600. Average number of simulated errors = 5133

- Test C. With simulated reads generated using the following parameters: Error Rate = 1%, $L \sim N(600, 2500)$, Sequence Length = 100000, Number of Reads = 3200. Average number of simulated errors = 979

- Test D. With simulated reads generated using the following parameters: Error Rate =

Table 3.3: Test B: with $l = 10$

	SP	GD	FT
Corrections	7499	5932	5012
Errors introduced	849	1553	544
Errors after Correction	1210	2130	1068
Unfixable Reads	61	89	7

Table 3.4: Test B: with $l = 20$

	SP	GD	FT
Corrections	7109	5633	5243
Errors introduced	776	1297	682
Errors after Correction	1230	1842	934
Unfixable Reads	74	80	8

Table 3.5: Test C: with $l = 10$

	SP	GD	FT
Corrections	945	1103	988
Errors introduced	59	124	48
Errors after Correction	132	175	121
Unfixable Reads	4	2	0

Table 3.6: Test C: with $l = 20$

	SP	GD	FT
Corrections	913	1130	1042
Errors introduced	43	74	43
Errors after Correction	80	124	64
Unfixable Reads	4	2	0

Table 3.7: Test D: with $l = 10$

	SP	GD	FT
Corrections	864	1022	841
Error introduced	44	103	40
Errors after Correction	119	125	115
Unfixable Reads	3	3	0

Table 3.8: Test D: with $l = 20$

	SP	GD	FT
Corrections	860	993	843
Errors introduced	38	79	29
Errors after Correction	71	110	59
Unfixable Reads	4	4	0

1%, $L \sim N(300, 900)$, Sequence Length = 100000, Number of Reads = 6400. Average number of simulated errors = 1032

- Test E. Running time of FT. With simulated reads generated using the following parameters: Sequence Length = 1000000, Error Rate = 1%
- Test F. Real Reads from NCBI Trace Archive: *TARSIUS SYRICHTA* (Sequence length 932377, average coverage 8.9)

When run on simulated data, FT makes fewer corrections in general, but the number of errors in the output is the fewest among all, especially when coverage is low (Test A and B). In some cases the number of errors after error correction is higher, but considering the number of

Table 3.9: Test E: FT running times

Read Length Mean	300	600	1200
3200 reads	74s	166s	362s
6400 reads	154s	335s	730s
12800 reads	303s	674s	1467s

Table 3.10: Test F: *TARSIUS SYRICHTA* with $l = 10$

	SP	GD	FT
Corrections	n.a.	14029	10012
Unfixable Reads	n.a.	158	39
Contigs	n.a.	73	52
Super tangles	n.	168	179

Table 3.11: Test F: *TARSIUS SYRICHTA* with $l = 20$

	SP	GD	FT
Corrections	n.a.	13032	11133
Unfixable Reads	n.a.	156	41
Contigs	n.a.	85	54
Super tangles	n.a	144	136

"unfixable" reads, it in fact still gives better results. This is likely due to accurate estimation for coverage in low coverage regions. However the difference becomes small in Test C,D, which have average coverage of 20. In the presence of higher error rate (Test B), the greedy algorithm loses out quite drastically. This is expected, since it cannot make coordinated corrections. FT produces very few unfixable reads, in contrast to GD and SP. We can attribute this to the low-coverage detection heuristic.

FT has fewest number of errors introduced in all cases, but the difference is again small with high coverage.

The running times of FT are acceptable, as shown in table for Test E. Due to time limitation not much code optimization was done, and the hashing method used may not be the optimal. Between two cells, the ratio of their running times is close to the size of input, giving the hint that it runs mostly in linear time. With the largest input size, FT is able to finish in 24 minutes, making it a realistic replacement for GD.

When run on real sequence data for *TARSIUS SYRICHTA*, output from FT led to much fewer (by around 25% to GD) contigs after running EULER. This should be attributed to adjusted threshold for low coverage regions. The number of super tangles is close to that for GD and SP. Our interpretation is that, like the other two, First Time Error Correction cannot avoid false negatives when dealing with non-exact repeats and repeat borders, which introduces tangles. We do not have data for SP in this test, since it took too long to complete.

3.2 Repeated Errors Correction

It is strongly expected that, with Second Time Error Correction, much better results will be obtained for both number of errors after correction and super tangles. Being able to deal with non-exact repeats and repeat borders, the number of errors introduced is expected to be close to 0.

The implementation is still under progress, and will be reported once completed.

Chapter 4

Conclusion and Extension

Error corrections using First Time Correction in the new scheme already decreases number of errors by up to 24%, when compared with the better result of the 2 other schemes when run on test data simulated from the known sequence *T.whipplei*. In the best scenario the number of errors after correction is 15% of that before correction.

It's expected that when integrated with the second part, which utilizes transformed de Buijn graph, much better results will be given after error correction is redone. We hope to finish the on-going implementation soon.

Very recently (ZhengZW06) proposed an alternative way, also making use of structure of de Buijn graph for error correction. They transform the de Bruijn graph while doing error correction, trying to remove erroneous edges dynamically. Future work may be to compare it with our scheme, and possibly incorporate the idea of doing error correction *on* the de Buijn graph.

References

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network flows : Theory, algorithms, and applications.* Englewood Cliffs, NJ: Prentice Hall.

Chaisson, M., Pevzner, P. A., & Tang, H. (2004). Fragment assembly with short reads. *Bioinformatics*, *20*(13), , 2004.

F. Sanger, A. R. Coulston, G. H. e. a. (1982). Nucleotide sequence of bacteriophage lamda dna. *J. Mol. Biol.*, *162*, , 1982, 729–773.

F. Sanger, S. N., & Coulson, A. R. (1977). Dna sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA*, *74*, , 1977, 5463–5467.

G. Sutton, O. White, M. A., & Kerlavage, A. (95). Tigr assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science & Technology*, *1*, , 95, 9–19.

Guibas, L. J., & Odlyzko, A. M. (1981). String overlaps, pattern matching, and nontransitive games. *J. Comb. Theory, Ser. A*, *30*(2), , 1981, 183–208.

Huang, X., & Madan, A. (99). Cap3: A dna sequence assembly program. *Genome Research*, *9*, , 99, 868–877.

Hui, L. C. K. (92). Color set size problem with application to string matching. *Combinatorial Pattern Matching, Third Annual Symposium*, Vol. 644 of *Lecture Notes in Computer Science* (pp. 230–243), Tucson, Arizona, 29 April–1 May, 92: Springer.

Idury, R. M., & Waterman, M. S. (95). A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, *2*(2), , 95, 291–306.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., & Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In A. Amir, & G. M. Landau (Eds.), *CPM*, Vol. 2089 of *Lecture Notes in Computer Science* (pp. 181–192), , 2001: Springer.

Månsson, M. (2002). Pattern avoidance and overlap in strings. *Combinatorics, Probability & Computing*, *11*(4), , 2002.

Maxam, A. M., & Gilbert, W. (1977). A new method for sequencing dna. *Proc. Natl. Acad. Sci. USA*, *74*, , 1977, 560–564.

Myers, E. W. (95). Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, *2*(2), , 95, 275–290.

Pevzner, P. A. (1989). L-tuple DNA sequencing: Computer analysis. *J. Biomol. Struct. Dyn.*, *7*, , 1989, 63–73.

Pevzner, P. A., Tang, H., & Tesler, G. (2004). De novo repeat classification and fragment assembly. In P. E. Bourne, & D. Gusfield (Eds.), *RECOMB* (pp. 213–222), , 2004: ACM.

Pevzner, P. A., Tang, H., & Waterman, M. S. (2001). A new approach to fragment assembly in DNA sequencing. *RECOMB* (pp. 256–267), , 2001.

Regnier, & Szpankowski (1998). On the approximate pattern occurrences in a text. *SEQS: Sequences '91*, , 1998.

W.M. Zheng, H. Z., & Wang, X. (2006). An approach to correcting dna sequencing error. *Journal of Software, China*, *2*, , 2006, 193–199.