Error Correction for DNA Sequencing via Disk Based Index and Box Queries

by

Yarong Gu

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science (Computer and Information Science) in The University of Michigan-Dearborn 2017

Master's Thesis Committee:

Professor Qiang Zhu, Chair
Professor Jie Shen
Associate Professor Brahim Medjahed

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my thesis advisor Professor Qiang Zhu for his continuous help and support. He is knowledgeable in the field of database and bioinformatics, and he never hesitates to offer guidance. I can always recall the days we had heated discussions on the topics, on all the challenging parts and technical issues. I learnt from his knowledge, and yet I learnt the most from his honorable academic attitude. Professor Zhu is an incredible professor, and an even better mentor.

I would also like to thank the rest of my thesis committee: Professor Jie Shen and Professor Brahim Medjahed, I am gratefully indebted to their valuable comments and insightful suggestions on this thesis.

Next, I want to thank Professor Sakti Pramanik and Professor C. Titus Brown at the Michigan State University for collaborating with us. Thanks to my colleague Xianying Liu, who graduated in the year of 2015, for discussing various interesting topics with me, and for providing guidance starting from my initial stage of research.

Finally, I must express my very profound gratitude to my family for providing me with endless assistance and love throughout my study and research.

This research was supported by the US National Science Foundation (NSF) (under Grants #IIS-1320078 and #IIS-1319909) and the University of Michigan.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The vast increase in DNA sequencing capacity over the last decade has quickly turned biology into a data-intensive science. Nevertheless, current sequencers such as Illumia HiSeq have high random per-base error rates, which makes sequencing error correction an indispensable requirement for many sequence analysis applications. Most existing methods for error correction demand large expensive memory space, which limits their scalability for handling large datasets. In this thesis, we introduce a new disk based method, called DiskBQcor, for sequencing error correction. DiskBQcor stores *k*-mers for sequencing genome data along with their associated metadata in a disk based index tree, called the BoND-tree, and uses the index to efficiently process specially designed box queries to obtain relevant *k*-mers and their occurring frequencies. It takes an input read and locates the potential errors in the sequence. It then applies a comprehensive voting mechanism and possibly an efficient binary encoding based assembly technique to verify and correct an erroneous base in a genome sequence under various conditions. To overcome the drawback of an offline approach such as DiskBQcor for wasting computing resources while DNA sequecing is in process, we suggest an online approach to correcting sequencing errors. The online processing strategies and accuracy measures are discussed. An algorithm for deleting indexed *k*-mers from the BoND-tree, which is a step stone for the online sequencing error correction, is also introduced. Our experiments demonstrate that the proposed methods are quite promising in error correction for sequencing genome data on disk. The resulting BoND-tree with correct *k*-mers can also be used for sequence analysis applications such as variant detection.

CHAPTER 1

Introduction

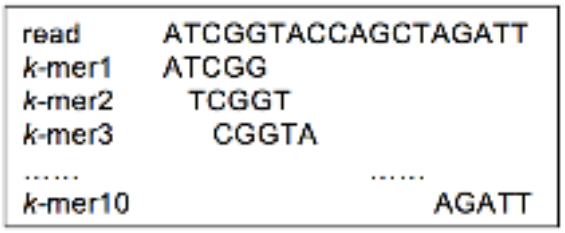
1.1 Motivation

DNA sequencing has been increasingly serving studies on biological problems including biomedical diagnostics, gene expression analysis, drug resistance, complex microbial ecosystems, and basic molecular biology. However, current sequencers have quite high random per-base error rates, ranging from 1% for Illumina HiSeq to 15% for Pacific Biosciences SMRT [1]. Dealing with sequencing errors is a significant challenge for both mapping and assembly based approaches to sequence analysis. Error correction has emerged as one of the dominant practical problems in sequence analysis [1]. Moreover, comparing to the conventional sequencing methods, the next-generation sequencers produce shorter read lengths, leading to more repeats [2], which makes it more difficult to decide an appropriate sequencing error correction method.

Fortunately, the low sequencing cost of the next-generation sequencers has made it possible to detect and correct the errors by providing a high redundant coverage with sequencing reads for the target genome sequence. For a sufficiently large coverage and k, almost all the sequencing errors alter the relevant k-mers (i.e., short substrings of length k obtained from sequencing reads) to versions that do not exist in the target genome sequence. Therefore, k-mers with low counts, particularly those occurring just once or twice, usually inherit sequencing errors from the corresponding reads. Error correction and other relevant sequence analysis applications [3–5] have made counting large amounts of k-mers a paramount need for research in bioinformatics.

1.2 Problem Statement

Given a sample DNA, a DNA sequencer, such as Illumina HiSeq [6], determines the order of the four types of nucleotides/bases: Adenine, Guanine, Cytosine, and Thymine (A, G, C, T), reported as a text string (read). Each read usually contains several hundreds to thousands base pairs. A *k*-mer is obtained from the read by retrieving overlapping *k*-length bases. For example, Figure 1.1 demonstrates how a sequence read is decomposed into shifted *k*-mers.



```
read      ATCGGTACCAGCTAGATT
k-mer1    ATCGG
k-mer2     TCGGT
k-mer3      CGGTA
......                ......
k-mer10                AGATT
```

Figure 1.1: A read is decomposed to shifted *k*-mers.

These sequencing reads may contain errors, normally at the rate of 0.5% per base, caused by limitations in the DNA sequencing technique or by errors during PCR amplification.

Most of the researches on error correction methods can be classified into three types: k-spectrum based, suffix tree/array-based, and MSA-based [7]. The core problem falls on how we deal with the computational challenges for the highly redundant large *k*-mer datasets. Existing data structure used including hash table [8], Bloom filter [9], suffix tree [5], and sorted bin set [10] to store *k*-mers. These in-memory structures provide efficient random access to *k*-mers in memory.

However, they usually have a high demand on computing resources. For example, Jellyfish [8] is a hash based counting method designed for shared memory parallel computers, which was run on a computer with 32 cores and 256GB RAM. Such high-end computing equipment is not common for most biology laboratories today.

One way to reduce the expensive memory requirement for an error correction method is to develop a new technique utilizing relatively cheap disk space. There are two challenges in doing so: (1) how to efficiently search target *k*-mers from a large *k*-mer dataset on disk, and (2) how to utilize search results to correct sequencing errors. If we could utilize an efficient disk based data structure with suitable properties for supporting genome analysis problems, especially, the DNA sequencing error correction problem, the above two challenges can be nicely solved.

1.3 Our Approach

To tackle the above challenges, we propose a novel disk based method, called DiskBQcor, that stores the *k*-mers with a recently developed index structure, called the BoND-tree [11], on disk and adopts a vast majority voting mechanism to verify and correct the sequencing errors at suspicious positions in the sequencing reads generated by a sequencer for a target genome sequence. Specifically, *k*-mers are kept in the leaf nodes of the BoND-tree, and each *k*-mer is associated with a pointer pointing to the relevant metadata (e.g., the set of ids of the sequencing reads that contain the *k*-mer). Each (leaf or non-leaf) node of the BoND-tree is saved in a disk block, and the metadata is stored in additional disk blocks linked from the leaf nodes. For a suspicious error position in a sequencing read, a set of special box queries (BQ) are performed on the BoND tree to count and vote for each possible base at that position and determine what the correct base is at the position. Various scenarios are considered in DiskBQcor. The extreme cases are handled by an efficient binary encoding based assembly technique. Experiments demonstrate that DiskBQcor is quite promising in achieving high accuracy for error correction with reasonable efficiency, besides the scalability benefit warranted by a disk based approach.

DiskBQcor focuses on error verification and correction. We are able to detect the suspicious error positions in a sequencing read with *k*-mer abundance analysis [12]. Specifically, through analyzing on the *k*-mer abundance, the separation between the high-abundance *k*-mers and the low-abundance *k*-mers will become clear. A suspicious error position is typically at the boundary between these two types of *k*-mers. By choosing a proper cutoff value, suspicious error positions in sequencing reads can be identified. The base at each suspicious error position can be checked (verified) by DiskBQcor to see if it is indeed an error. If so, DiskBQcor can find the correct base to replace the erroneous base at the position. Furthermore, our method has been extended to an online/streaming fashion, so that we can start error detection and correction in the very early stage while DNA sequencing is still in process, resulting in a better utilization of computer resources.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 discusses the related works about index methods for non-ordered discrete data, *k*-mer counting tools, genome error correction approaches, and online/streaming analysis of DNA sequencing reads. Chapter 3 discusses the details of the proposed DiskBQcor, which aims to utilize the BoND-tree to verify and correct errors on suspicious positions in sequencing reads. Chapter 4 presents an algorithm to delete *k*-mers in the BoND-tree, which serves as a preparation for Chapter 5. Chapter 5 extends DiskBQcor introduced in Chapter 3 to an online sequencing error correction method, and discusses its theoretical accuracy measures. Chapter 6 concludes the thesis and highlights some future research directions.

CHAPTER 2

Related Work

In this chapter, we discuss the work related to our study. Section 2.1 presents the work related to indexing techniques for non-ordered discrete data. Section 2.2 reviews a few recent developed *k*-mer counting tools. Section 2.3 discusses some popular genome sequencing error correction approaches related to our work. Finally, Section 2.4 overviews the related work on streaming/online analysis of DNA sequencing reads.

2.1 Index Methods for Non-ordered Discrete Data

Existing indexing methods for multidimensional datasets are mostly developed for a multidimensional (ordered) Continuous Data Spaces (CDS) such as the R-tree [13]. However, characteristics of genome sequences require another type of index method to store *k*-mers that are non-ordered and discrete. The essential geometric concepts that appear frequently in a CDS, such as a minimum bounding rectangle and the area of a region, cannot be used in a multidimensional Non-Ordered Discrete Data Spaces (NDDS) directly.

The ND-tree [14] is an R-tree-like index structure designed for an NDDS. The tree was originally introduced to support similarity searches/queries, which require a robust indexing technique, and was later modified into a so-called BoND-tree to support efficient box queries in [11]. In [14], related essential geometry concepts in a CDS were extended to an NDDS, and the structures in the R-tree and R*-tree were modified according to the new concepts. Specifically, a leaf node in an ND-tree contains an array of entries of the form (op, key), where key is a vector in the underlying

NDDS and op is a pointer to the object represented by key in the database. A non-leaf node N in an ND-tree contains an array of entries of the form ($cp, DMBR$), where cp is a pointer to a child node N' of N in the tree, and *DMBR* is the discrete minimum bounding rectangle of N'.

The M-tree [15] and the Slim-trees [16] are two examples of the indexing methods based on metric spaces, which only require the distance measure between two vectors. However, such metric index methods are too general to optimize the query performance since the unique characteristics of an NDDS are not utilized. Experiments in [14] also demonstrated superior performance obtained by the ND-tree for real genome sequence data.

On the other hand, much work has been proposed to index DNA sequences including the work in [17–20]. Most of the indexing methods are main memory based. Cao *et al.* [17] proposed a two-level index of hash table and c-trees based on q-grams of DNA sequences, which is efficient in detecting similarity regions. However, since the c-trees are a group of dynamic trees, this method is hard to be utilized to correct a *k*-mer from the trees for future use. Huang *et al.* [19] proposed a scheme to index highly similar sequences based on the Burrows-Wheeler Transform. Although their method has much less memory requirement, it is only suitable for nearby overlapping sequences, and not to be used for a large number of both overlapping and non-overlapping reads as we need in our sequence analysis and error correction application. Kahveci *et al.* [20] presented a wavelet-based method to map the substrings of the data into an integer space with the help of wavelet coefficients, and index these coefficients using minimum bounding rectangles. For range queries and nearest neighbor queries, they managed to split them into subqueries of available resolutions, and that large amounts of data strings are pruned from the database. While this method compresses database and provides fast filtering, it does not support efficient box queries, since the cost to remove false hits is high.

As previously discussed, the BoND-tree introduced in [11] is an index tree developed by modifying the ND-tree to support box queries in an NDDS. It also optimizes some of the structures of the ND-tree to achieve an improved efficiency when processing box queries. The authors provided three node splitting heuristics and theoretical analysis to show the optimality of the proposed heuristics. Extensive experiments were also conducted to show effectiveness of their scheme for supporting efficient box queries on genome sequence databases.

2.2 *k*-mer Counting Tools

In order to deal with the computational challenges for large *k*-mer datasets, a number of efficient *k*-mer counting methods have been proposed in the literature. Most of them use a large in-memory structure such as the hash table [8], the Bloom filter [9], the suffix tree [5], and the sorted bin set [10] to store *k*-mers. These in-memory structures provide efficient random access to *k*-mers in memory. However, they usually have a high demand on computing resources. For example, Jellyfish [8] is a hash based counting method designed for shared memory parallel computers, which was run on a computer with 32 cores and 256GB RAM. Such high-end computing equipment is not common for most biology laboratories today.

Two *k*-mer counting methods, i.e., DSK [21] and KMC 2 [22], that utilize disk space to reduce the memory requirement were proposed. The goal of these two methods is to count the number of occurrences for every *k*-mer in a (multi-)set of *k*-mers. Both methods adopt a similar approach. The basic idea is to divide the (multi-)set of *k*-mers into a number of groups/partitions, save each group to the disk, load each group into a temporary in-memory structure separately, and count *k*-mers by traversing each temporary in-memory structure. Since these two methods aim at counting all the *k*-mers in the given set, they do not have to support efficient random access to the *k*-mers on disk. On the other hand, the goal of our DiskBQcor is to verify and correct a suspicious error(s) at a given position(s) in a read, which implies that we need to search and count only those interested *k*-mers from the given dataset. As a result, efficient random disk access is required for DiskBQcor. The recent BoND-tree mentioned in Section 2.1 is adopted to provide such a searching capability for a large *k*-mer dataset on disk.

2.3 Genome Error Correction Approaches

Error correction methods can be classified into three types: the *k*-sepctrum based, the suffix tree/array-based, and the MSA-based [7] methods. Quake in [23] is a widely-used error correction tool/method. It utilizes Jellyfish to perform the counting job during its error correction process. Beyond a plain *k*-mer counting, Quake also takes into account the quality scores of base calls when distinguishing untrusted *k*-mers (i.e., having low counts) from trusted ones. Lighter in [24] and the spectral alignment based parallel error correction method in [25] utilize a Bloom filter for error correction, while SHREC in [26] and its variant in [27] utilize suffix trees for error correction. Coral [28] puts the *k*-mers as keys, with their related reads as value into a hash table, and uses multiple alignments to identify errors.

As mentioned earlier, Quake [23] is a *k*-spectrum based method, widely used to correct errors in sequencing reads with high coverage. It takes *k*-mer statistics as an important component to accomplish its task. However, Quake has some limitations: (1) it has to use a sufficiently large *k* to achieve high accuracy; (2) it cannot handle the cases in which untrusted *k*-mers are repeated at several places or an erroneous base occurs near the boundary of a read. Limitation (1) leads to a large size of the *k*-mer dataset, while limitation (2) causes Quake to trim off some true error cases from consideration. As we will see, with carefully designed box queries and other strategies, DiskBQcor is able to mitigate both limitations. To our knowledge, no similar work has been reported in the literature.

2.4 Streaming/Online Analysis of DNA Sequencing Reads

Streaming/online analysis of DNA sequencing reads is a new topic attracting researchers recently. There are two interesting observations. First, streaming analysis to analyze sequencing reads is done in a linear time complexity. Second, a sequencer produces reads in a streaming fashion and takes a certain amount of time, which allows us to analyze them in parallel during the process, and get the result as soon as the sequencing completes. A semi-streaming approach may read the input for more than one time, while a fully streaming approach reads the input only once, but both run in a linear time complexity. KmerStream [29] introduces a streaming algorithm to estimate the number of *k*-mers that occur exactly once, which takes a liner time complexity. It works as follows: (1) compute the hash value of a *k*-mer; (2) sample the stream at different rates; (3) select a most suitable rate. KmerStream can be further utilized for *k*-mer abundance analysis, and for the estimation of error rate in a given dataset.

Zhang *et al.* in [12] presents a semi-streaming algorithm for *k*-spectrum analysis, and utilizes digital normalization which can effectively detect and remove errors. It builds a De Bruijin graph to detect the graph saturation to see whether a certain region reaches the expected coverage. After that, it works as follows: first pass - if the coverage is less than the desired coverage, load the read to the graph, else analyze the read; second pass - if the coverage is greater or equal to the desired coverage, analyze the read. In the paper, the approach was also reduced to a fully streaming approach, which was used to estimate the per-base error rate.

CHAPTER 3

A Disk Based DNA Sequencing Error Correction Method

In this chapter, we propose a disk based sequencing error correction method, called DiskBQcor, which stores the *k*-mers in a BoND-tree. We will first briefly go through key concepts used for the BoND-tree in Section 3.1, and then introduce a vast majority voting mechanism to verify and correct the sequencing errors in Section 3.2. Various scenarios and extreme cases are considered in DiskBQcor. Experimental results to evaluate DiskBQcor are reported in Section 3.3.

3.1 Preliminaries - Building the BoND-tree

We can view a *k*-mer (e.g., "agc" with $k = 3$) as a vector in a *k*-dimensional Non-ordered Discrete Data Space (NDDS), where the letter (base) on each dimension of the vector is from alphabet $\{a, t, c, g\}$. In general, an NDDS Ω_d is a multi-dimensional vector space, where d is the number of dimensions in Ω_d. Each dimension in Ω_d has an alphabet (domain) A_i ($1 \leq i \leq d$) consisting of a finite number of letters, where no natural ordering exists among the letters. Let b_i ($1 \leq i \leq d$) be a subset of alphabet A_i (i.e., $b_i \subseteq A_i$). The Cartesian product $b_1 \times b_2 \times ... \times b_d$ is called a discrete box (rectangle) in Ω_d. More concepts about an NDDS can be found in [30, 31].

A (discrete) box query q on a dataset S in an NDDS is defined as a query with a specified box w that returns all the vectors from S that lie within w. For example, a box query with box $\{a\} \times \{g, t\} \times \{c, t\}$ on a *k*-mer dataset ($k = 3$) fetches those *k*-mers from the dataset that have letter (base) a on the first dimension, g or t on the second dimension, and c or t on the third dimension. Thus, this box query is equivalent to four exact queries to search for four individual *k*-mers: agc, atc, agt, and att. As we will see, box queries can be utilized to help efficiently solve the sequencing error correction problem.

The BoND-tree is a recent disk based index technique that was specially designed for supporting efficient processing of box queries in an NDDS [11]. It has a balanced hierarchical indexing tree structure (see Figure 3.1). Each (leaf or non-leaf) node consists of a set of entries and occupies one disk block. Each entry in a non-leaf node consists of a pointer pointing to a subtree and the minimum bounding box (mbb) for all the vectors stored in the subtree. Each entry in a leaf node consists of a vector/*k*-mer (as a key) and a pointer pointing to relevant metadata. Special strategies making use of the characteristics of an NDDS are adopted to build the tree so that box queries can be processed efficiently on the tree [11].

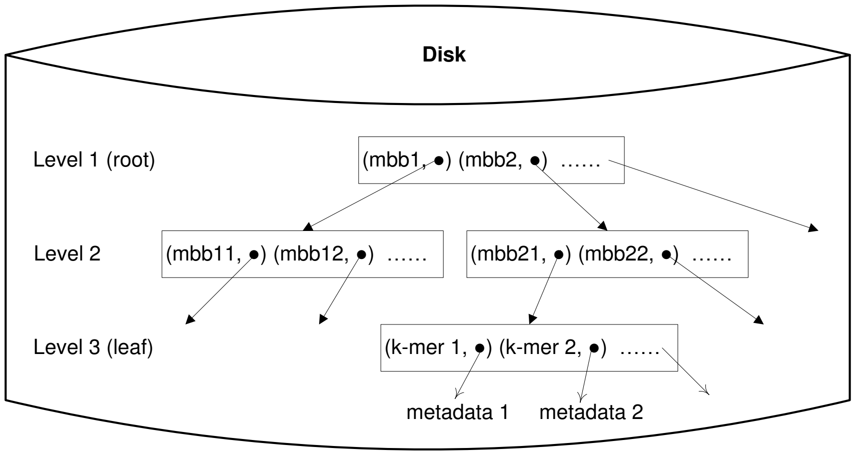


Figure 3.1: The BoND-tree Structure.

When the overlapping *k*-mers obtained from the given sequencing reads are loaded into a BoND-tree in DiskBQcor, the canonical representation of each *k*-mer, which is either the *k*-mer itself or its reversed complement, is calculated and inserted into the tree. The metadata associated with each *k*-mer (in the canonical form) in the corresponding leaf node in DiskBQcor is a list of ids for the reads that contain this *k*-mer or its reserved complement (indicated by a flag), which is saved in one or more linked disk blocks. Note that a conventional memory-based method can only afford saving the *k*-mers and their counts in their in-memory structures due to their restricted scalability for handling large datasets. Hence, more detailed information such as the read ids are not stored with the *k*-mers, which need to be identified on the fly via expensive operations such as alignments. Since DiskBQcor uses disk space, all necessary metadata can be saved with the *k*-mers, which reduces a large amount of unnecessary dynamic computing overhead.

3.2 The Method

The basic idea of our disk based sequencing error correction method, DiskBQcor, works as follows. The overlapping *k*-mers obtained from the sequencing reads for a target genome sequence along with their relevant metadata are loaded into a BoND-tree on disk. For a given suspicious error position in a sequencing read, a set of special shifted box queries are formulated and performed on the BoND-tree to retrieve the relevant *k*-mers and their counts. With a special voting mechanism, the possibly erroneous base at the given position can be verified positively or negatively, and the correct base at the position can be identified if an error is found. In the latter case, the erroneous base at the suspicious position in the corresponding read is replaced/corrected by the correct one. The relevant details of the above correcting procedure are discussed in the following subsections.

3.2.1 Error Correction via Vast Majority Voting

Given a sequencing read (e.g., $r = acctgga[t]tcgtag......$) and a suspicious error position in the read (e.g., the 8th position $[t]$ in r), the possibly erroneous base (e.g., t) at the position can be verified and corrected (if proven to be an error) by a voting approach described as follows.

We can choose a suspicious *k*-mer (e.g., $gga[t]tc$ with $k = 6$ in the above example) that covers the suspicious error position, replace the possibly erroneous base at the position (e.g., t in the above example) by each of the four possible bases (i.e., a, t, c, g) to form four *k*-mers (e.g., $ggaatc$, $ggagtc$, $ggattc$, and $ggactc$ in the above example), and use these *k*-mers[1] as exact queries

[1]It is assumed that each *k*-mer is converted into its canonical representation before searching it in the BoND-tree.

to find the number of occurrences (i.e., the count) for each of them at the suspicious position. Since there usually is a high coverage (e.g., about 20 times) with sequencing reads at the suspicious error position, most of the reads typically contain the correct base at the position. Using a vast majority voting rule, we can discover if the possibly erroneous base (e.g., t) is indeed an error and, if so, what the correct base is.

One optimization that can be further done here is to group the above four exact queries into one box query by using set $X = \{a, t, c, g\}$ at the suspicious position of the suspicious *k*-mer (instead of using four individual bases separately) to indicate all the possible bases at that position. In other words, the box used for the box query in the above example is: $\{g\} \times \{g\} \times \{a\} \times X \times \{t\} \times \{c\}$ (simply denoted by $ggaXtc$ in the remaining discussion). The votes/counts can be found by analyzing the result of the box query. Since only one (box) query instead four (exact) queries is executed on the BoND-tree, the efficiency is usually further improved due to the reduced latency and shared processing. Once an erroneous base has been verified, the error correction is just a matter of replacing the erroneous base by the correct one.

Figure 3.2 shows an example for error verification and correction. Assume that the average coverage is 20 (times), which implies that each position in the target genome sequence is covered by about 20 sequencing reads on average. In the figure, reads 1, 2, 3, and seventeen other sequencing reads are produced by a sequencer to cover a certain range of the target genome sequence. Assume that the sequencer has produced an erroneous base (altering a to t) at the indicated position for read 3, making any *k*-mer that contains the position suspicious. After performing a box query whose box is obtained by replacing the erroneous base by set $X = \{a, t, c, g\}$ in a chosen suspicious *k*-mer β on the BoND-tree, the counts for a and t can be found to be 19 and 1, respectively[2]. Apparently, the correct base at the suspicious position is a.

In general, the correct base is the one that has the maximum count: $\underset{y \in X}{max} \{count(y)\}$ provided

[2]The counts for a and t (with respect to suspicious *k*-mer β) are the counts for modified β with a and t being placed at the suspicious position, respectively. The counts for c and g (with respect to β) can be defined similarly.

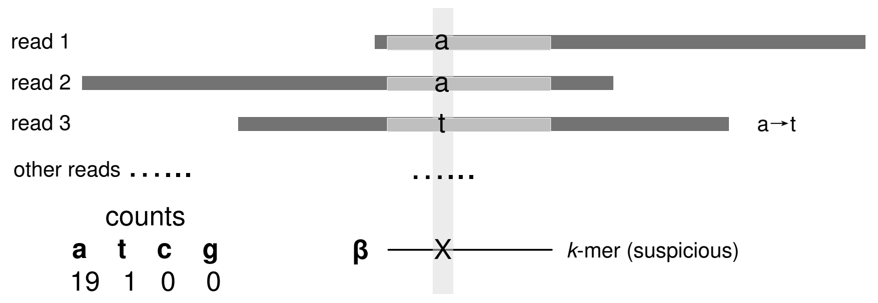


Figure 3.2: Voting in Case of Solo Occurrence of a Suspicious *k*-mer.

that this maximum count is significantly larger than the count of any other base at the position. If the possibly erroneous base at the position has the maximum count, it implies that this base is not an error. Otherwise, this base is determined to be an error.

However, the above simple voting approach cannot handle the situation when a suspicious *k*-mer β that we use to convert into a box query for a suspicious error position happens to also appear as a trusted (without error) *k*-mer in another place of the target genome sequence. Figure 3.3 shows such a scenario, where β appears as a suspicious *k*-mer in read 2 and also appears as a trusted *k*-mer in read 1. Assume that the average coverage is still about 20 (times) in this example. In such a case, the counts for the erroneous base t and the correct base a are about the same (e.g., 21 = 1 + 20 vs. 19 = 19 + 0) since the counts for t in β from both places are combined/added. Hence, it is difficult to determine which base is correct at the suspicious error position (note that a small variance in the coverage is normal).

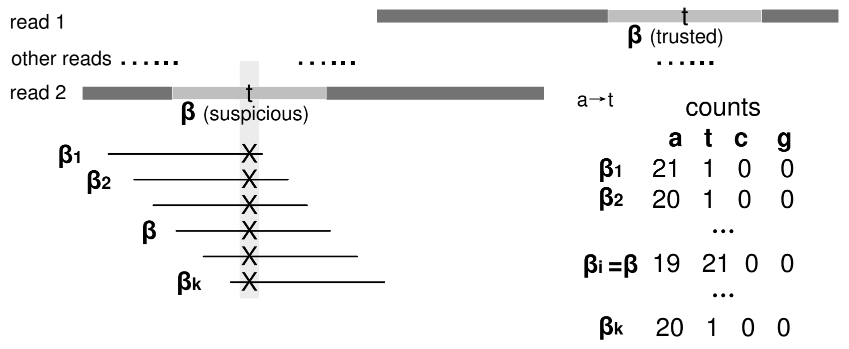


Figure 3.3: Voting in Case of Repetitive Occurrences of a Suspicious *k*-mer.

To solve this problem, we consider all the (shifted) suspicious *k*-mers $\beta_1, \beta_2, ..., \beta_k$ that cover the suspicious error position as shown in Figure 3.3. For each β_i ($1 \leq i \leq k$), we form a box query by replacing the possibly erroneous base (e.g., t) at the suspicious error position by set $X = \{a, t, c, g\}$.

After these k number of (shifted) box queries are performed on the BoND-tree, the following $k \times 4$ counting matrix representing all the counts with respect to the k suspicious *k*-mers can be obtained:

$$\begin{pmatrix} a_1 & t_1 & c_1 & g_1 \\ a_2 & t_2 & c_2 & g_2 \\ \cdots & & & \\ a_k & t_k & c_k & g_k \end{pmatrix} \tag{3.1}$$

where a_i, t_i, c_i, g_i ($1 \leq i \leq k$) are the counts for suspicious *k*-mer β_i when its possibly erroneous base at the suspicious error position is replaced by base a, t, c, g, respectively. Let the vote of base y be

$$v(y) = \min_{1 \leq i \leq k} \{y_i\} \tag{3.2}$$

where $y \in \{a, t, c, g\}$. For the example shown in Figure 3.3, we have $v(a) = 19, v(t) = 1, v(c) = 0$, and $v(g) = 0$.

It is noted that $v(y)$ is usually close to the coverage number (e.g., 20) if base y is a correct one at the suspicious position since the counts for all the β_is are usually close to the coverage number when y is placed in the suspicious position[3]. Hence, $v(y)$ is relatively large in this case. On the other hand, when base y is an error, y_i is usually small if β_i does not appear as a trusted *k*-mer in another place, and y_i is large if β_i also appears as a trusted *k*-mer in another place. Hence, $v(y)$ is usually small if there exists at least one β_i that does not appear as a trusted *k*-mer in another place. Note that the chance for every β_i is repeated as a trusted *k*-mer in another place is small especially when *k* is reasonably large. Therefore, we can apply the following voting rule to determine a correct base:

[3]For a reasonable sequencer, it should produce most reads at each position of the target genome correctly.

Vast Majority Voting Rule (VMVR): The correct base at the suspicious error position is determined to be the one that has the following maximum vote:

$$MV =_{y \in X}^{max} \{v(y)\} \tag{3.3}$$

if MV is significantly larger than the vote for any other base at the position.

If the vote for the possibly erroneous base e at the suspicious error position is close to MV, e is determined to be not an error. Otherwise, it is an error. For the example in Figure 3.3, since $MV = v(a)$ (=19) and no other votes are close to MV, base t is determined to be an error at the suspicious error position, and base a is identified as the correct one at the position.

The condition for the vast majority voting mechanism based on Formula (3.3) to fail to work is that the votes for two or more bases are close to MV. We will present a strategy to handle error correction under this condition in Section 3.2.3.

3.2.2 Correcting Multiple Errors in One Read

Let us consider the situation in which a read has multiple suspicious error positions that need to be verified and corrected. If these suspicious error positions are located at least *k* positions/bases (distance) apart from each other in the read, they can be verified and corrected individually by the procedure presented in Section 3.2.1. However, when the distance between two consecutive suspicious error positions is less than *k* positions/bases, the above method cannot be directly applied. The reason is explained as follows.

Figure 3.4 shows an example in which we have two erroneous bases g and t at suspicious error positions p1 and p2, respectively, in read 3. Assume that the distance between p1 and p2 is $k - 2$ ($< k$). When we form all the (shifted) suspicious *k*-mers β_1, β_2, ..., β_k that cover the suspicious position p2, we notice that the first two suspicious *k*-mers β_1 and β_2 also cover the other suspicious position p1. After the box queries that are obtained by replacing the erroneous base at p2 with set

$X = \{a, t, c, g\}$ in each suspicious *k*-mer, the counting matrix is computed as shown in the figure. We can see that the counts for correct base a at p2 from β_1 and β_2 are 0 although most of reads have this correct base a at p2. The reason for this phenomenon is that β_1 and β_2 also contain another erroneous base g at p1, which prevent them from matching the corresponding correct reads at p2 (since they have correct base c at p1). As a result, this counting matrix cannot be used by the vast majority voting rule described in the last section since the correct base a at p2 cannot be identified in this way ($v(a) = 0$).

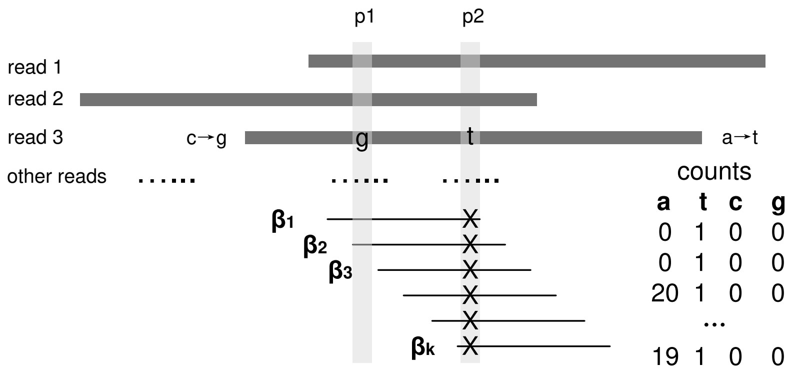


Figure 3.4: Multiple errors on one read.

In such a case, we adopt the following approach to correcting multiple errors in a read. We consider each suspicious error position in a read from the right to the left. For each suspicious position, we form all those (shifted) suspicious *k*-mers that cover the current suspicious position but do not cover the next suspicious position (e.g., β_3, ..., β_k for position p2 in Figure 3.4), convert them into box queries by using $X = \{a, t, c, g\}$ at the current suspicious position, use their results from the BoND-tree to form counting matrix (3.1), and apply the vast majority voting rule to determine if the possibly erroneous base at the current suspicious position is indeed an error. Once the current error is corrected in the read, we consider the next rightmost suspicious error position and correct its possibly erroneous base in a similar way. This process continues until all the possibly erroneous bases in the given read are verified and corrected. Note that, if the rightmost *k*-mer in the read contains more than one erroneous base, the process can start from the leftmost *k*-mer or any *k*-mer in the read that contains only one erroneous base and then expand the corrected region bigger and bigger. If no *k*-mer containing only one erroneous base can be found, we adopt the alignment method to be discussed in the following subsection to correct the erroneous bases covered by the rightmost *k*-mer first and then apply the method in this subsection to correct the remaining erroneous bases (if any) in the read.

3.2.3 Alignment Strategy

As the condition at the end of Section 3.2.1 indicates, our vast majority voting mechanism does not work if the votes for two or more bases are close to the maximum vote. Figure 3.5 shows such a scenario, in which segment δ (with length $2k-1$) covered by all the k suspicious shifted *k*-mers around the suspicious error position (with erroneous base t changed from correct base a) in read 2 also appears in read 1 that has no error. In such a case, every (shifted) suspicious *k*-mer in read 2 is repeated (as a trusted one) in read 1. Hence, the counts for the two places (from reads 1 and 2) are combined/added, which are close to the coverage number. As a result, we are unable to determine whether a or t is a correct base at the suspicious position in read 2 since they have similar votes (e.g., $v(a) = 19$ and $v(t) = 20$). Although such a case does not normally occur (especially when *k* is large), we need to have a way in our DiskBQcor to handle such a case when it does occur. The idea of an alignment based strategy to handle such a case is described as follows.

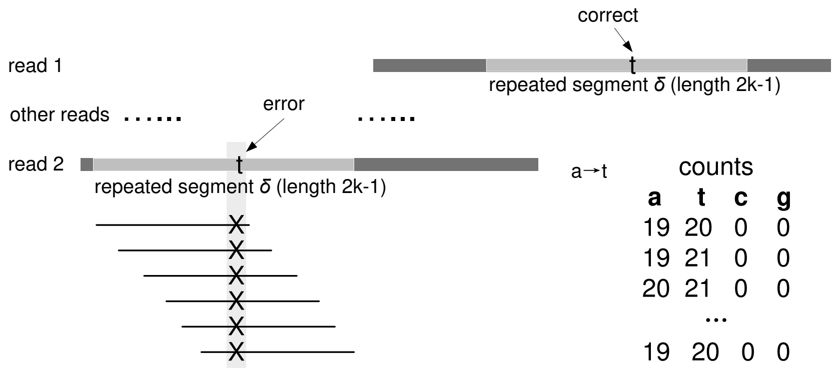


Figure 3.5: Reads with a long repeated segment.

For a given suspicious error position p in a read r that our vast majority voting mechanism cannot handle, we take a suspicious k-mer β that contains position p from read r and convert β into a box query q by replacing the possibly erroneous base[4] at position p in β by set $X = \{a, t, c, g\}$. From the result of q, we can find the set S of candidate reads that might be alignable with read r. Our goal is to identify those candidate reads in S that can indeed be aligned with r. After r is aligned with these candidate reads, the correct base at position p in r can be determined since most of them have the correct base at the position.

The problem now boils down to how to align the reference read r with a candidate read $r_1 \in S$. For this special alignment problem, we introduce an efficient method here to solve it in two stages (see Figure 3.6). In the first stage, we try to match a (short) seed segment σ from r that contains the suspicious error position p with a (short) segment σ_1 from r_1. If such σ_1 can be found, the alignment moves to the second stage. Otherwise, reads r and r_1 are not alignable. In the second stage, we first place r and r_1 into their aligned positions according to the alignment of σ and σ_1. We then try to match the pairs of corresponding bases at the remaining positions from r and r_1. If the total number of mismatches between r and r_1 is within a tolerance, r and r_1 are aligned successfully. Otherwise, they are not alignable.

To efficiently identify a segment(s) σ_1 from r_1 that is alignable with seed segment σ in r in the first stage, we use a binary encoding based technique. The key idea is described as follows. We first encode each of reads r and r_1 into a binary string/sequence by changing base "a" to "00", "t" to "01", "c" to "10", and "g" to "11" in the read. For example, a read "$aggctacgttaattga$" is converted into a binary sequence "00111110010010110101000001011100". This binary representation allows us to efficiently apply one integer (32 bits) operation to compare 16 bases together rather than apply 16 separate character/base comparisons to get the same result.

Let n be an integer (representing seed segment σ) from r that contains the possibly erroneous base e (in the binary form) at the suspicious position p. Assume that e is represented by the s-th and $(s + 1)$-th bits (counting from the right end) in n. We then check if n is alignable with each

[4]If β contains multiple possibly erroneous bases, each of them is replaced by set X.

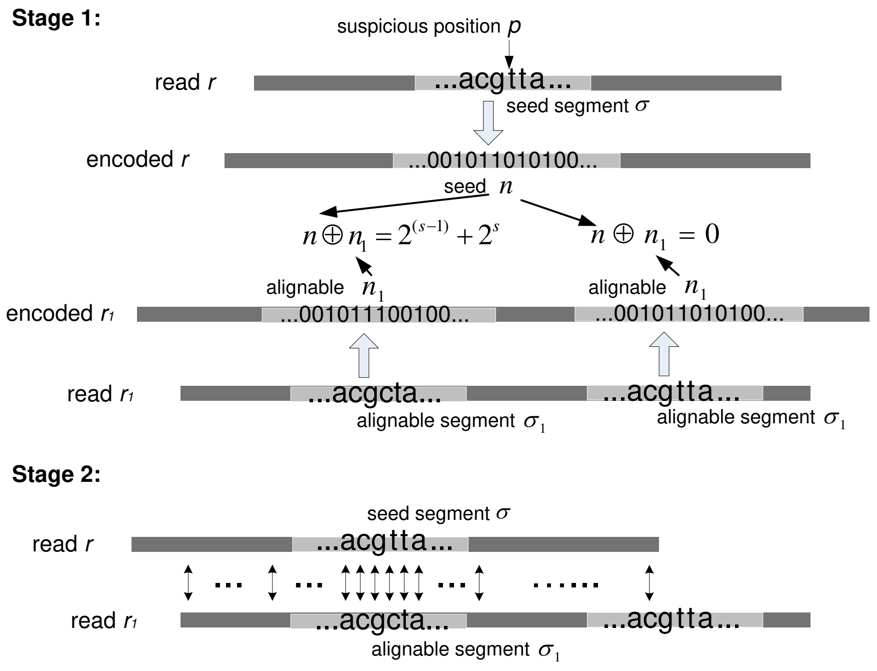


Figure 3.6: Two-stage Alignment Method.

integer n_1 shifted one base at a time from the left to the right in candidate read r_1[5]. If all the corresponding bits from n and n_1 match except that the s-th and/or $(s+1)$-th bits may mismatch, we say n and n_1 are alignable. Specifically, we perform an Exclusive OR ($\oplus$) operation on n and

[5]If an integer for a base b near the left or right boundary of r_1 does not have enough bases on the left or right side of b, we may pad it with relevant bases from n on the missing left or the right side.

n_1 to test if they are alignable as follows:

$$n \oplus n_1 = \begin{cases} 0 & => \ n \ and \ n_1 \ are \ alignable \\ & (they \ match \ exactly) \\ 2^{(s-1)} & => \ n \ and \ n_1 \ are \ alignable \\ & (they \ mismatch \ only \ at \ s\text{-}th \ bit) \\ 2^{s} & => \ n \ and \ n_1 \ are \ alignable \\ & (they \ mismatch \ only \ at \ (s+1)\text{-}th \ bit) \\ 2^{(s-1)} + 2^{s} & => \ n \ and \ n_1 \ are \ alignable \\ & (they \ mismatch \ only \ at \\ & \quad s\text{-}th \ and \ (s+1)\text{-}th \ bits) \\ otherwise & => \ n \ and \ n_1 \ are \ not \ alignable. \end{cases}$$

In this criterion for alignability, we allow only one base mismatch at the suspicious error position p. If seed segment σ also contains another erroneous base(s) that is not at position p, the chance to find an alignable segment σ_1 in candidate read r_1 is small since this would require that σ_1 also happen to have the same erroneous base(s) at the relevant position(s) other than p. Thus, the alignment would fail at the first stage in such a case. To increase the chance to use a seed segment that meets the no-other-error requirement, we adopt the following strategy (see Figure 3.7). We first use an integer n that represents a seed segment σ having position p in the middle. If n is found to be alignable with one or more integers in r_1, the alignment process moves to the second stage. Otherwise, we use an integer n' that represents a seed segment σ' having position p as its leftmost position. If n' is found to be alignable with one or more integers in r_1, the alignment process moves to the second stage. Otherwise, we use an integer n'' that represents a seed segment σ'' having position p as its rightmost position. If n'' is found to be alignable with one or more integers in r_1, the alignment process moves to the second stage. Otherwise, r and r_1 are considered to be not alignable.

Note that, if r has no other erroneous base at a position with a distance $\leq$ 16/2 = 8 bases from position p, seed segment σ (i.e., n) will meet the above no-other-error requirement. If r has only one erroneous base at a position (other than position p) with a distance $\leq$ 16/2 = 8 bases from p, seed segment σ' (i.e., n') or σ'' (i.e., n'') will meet the above no-other-error requirement. The above seed segment choosing strategy fails only when r has two or more erroneous bases at positions (other than position p) with a distance $\leq$ 16/2 = 8 bases from p and each side of p has at least one of such erroneous bases. However, the chance for such a case to occur is very small.

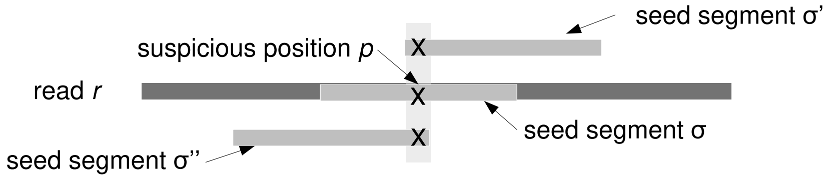


Figure 3.7: Strategy for Choosing Seed Segment.

In the second stage, for integer n_1 that is alignable with n (or n' or n''), we try to align reads r and r_1 according to the alignment of the segments represented by integers n and n_1. For each pair of corresponding bases from r and r_1 that are not covered by the segments represented by n and n_1, we compare them to see if they match. If the total number of mismatches is within a given tolerance, r and r_1 are aligned successfully. Otherwise, we try another n_1 (if any) that is alignable with n to see r and r_1 can be aligned using the new segment represented by n_1. If none of the integers from r_1 that are alignable with n leads to a successful alignment of r and r_1, reads r and r_1 are considered to be not alignable.

3.3 Experimental Results

To examine the performance of our disk based sequencing error correction method, DiskBQcor, we conducted extensive experiments. The performance was evaluated in terms of accuracy, efficiency and scalability. DiskBQcor was implemented in the C++ programming language. All the experiments were conducted on a Dell PC with a 3.2 GHz Intel Core i7-4790 CPU, 12 GB RAM, 5 TB Hard Drive, and Linux 3.16.0 OS. The performance data was measured based on the average from three executions.

The genome data used in the experiments was collected from *E. coli* 536 (GenBank: NC008253, 5.5 M). Different coverages with sequencing reads were tested in the experiments. Simulated 36 bp reads with an error rate at 0.5% were used. Different sets of overlapping *k*-mers for various *k* values were obtained from these reads for the experiments. For each error position p, all the shifted *k*-mers covering p are generated. For each shifted *k*-mer, its possibly erroneous base at position p is replaced by box $X = \{a, t, c, g\}$ to form the corresponding box query used for our error correction process.

The first set of experiments were conducted to evaluate the accuracy of our method. For each experiment, we measured the accuracy, i.e., the percentage of the errors that were properly corrected, from our method. We also measured the number of the errors that were mis-corrected and the number of the errors that were left uncorrected. Table 3.1 shows the results when *k* ranges from 12 to 16 and the coverage is fixed at 15 (times). The results demonstrate that, in general, the accuracy of our method is increasingly better as *k* increases, which is consistent with what is expected since a large *k* would reduce the chance for a *k*-mer to appear in multiple places, leading to a better determination of an error. $k = 15$ is the optimal length determined by Quake [23] for such experimental data. For this *k*, although the accuracy 99.87% of our method is better than the overall accuracy 90.5% of Quake, it is not a fair comparison since Quake also includes a step to determine the suspicious error positions while our method assumes that these positions are given. For those error positions that Quake tries to correct, it can correct 99.8% of them, which is comparable to the observed accuracy of our method. On the other hand, our accuracy covers all the error positions including those that Quake cannot handle such as the cases having repetitive suspicious *k*-mers. Hence, our method is very attractive for error correction when some suspicious error positions need to be checked. Furthermore, the experimental results also demonstrate that our method can yield a quite high accuracy even when *k* is relatively small (e.g., $k = 12$). Utilizing this advantage could help us reduce the space requirement by storing shorter *k*-mers, leading to an enhanced scalability for our method.

Table 3.1: Correction Accuracy for Simulated 36 bp *E. coli* with Coverage=15

k	Total errors	Corrections	Mis-corrections	Errors kept	Accuracy (%)
12	812764	786076	21	26667	97.00%
13	812869	806472	21	6375	99.21%
14	813890	811684	21	2185	99.73%
15	814122	813080	11	1031	99.87%
16	813816	813141	16	660	99.92%

To examine the impact of the coverage factor on the performance of our method, we conducted experiments with various coverages. Table 3.2 shows the results when the coverages are 10, 15, and 20 and *k* is fixed at 15. From the table, we can see that the coverage is not a significant factor affecting the performance of our method, which indicates that our vast majority voting mechanism is quite robust. For various coverages, the accuracy of our method remains almost steady. This advantage could help us further reduce the space requirement by adopting a low coverage genome dataset as input, which provides another way to further improve the scalability for our method. For example, changing the coverage from 20 to 10 would reduce the BoND-tree size by about 32% in our experiments, while the accuracy decreases very little (from 99.88% to 99.82%).

Table 3.2: Correction Accuracy for Simulated 36 bp *E. coli* with *k*=15

Coverage	Total errors	Corrections	Mis-corrections	Errors kept	Accuracy
10	541662	540672	4	985	99.82%
15	814122	813080	11	1031	99.87%
20	1084518	1083262	17	1239	99.88%

To evaluate the efficiency of our method, we recorded the relevant measures in our experiments. Table 3.3 shows the index creation time, the error correction time and the percentage of alignment cases when *k* ranges from 12 to 16 and the coverage is fixed at 15 (times), while Table 3.4 shows the same measures when the coverage ranges from 10 to 20 (times) and *k* is fixed at 15. From the tables, we can see that the error correction time for our method was relatively small, which indicates that the disk based BoND-tree can help achieve reasonable efficiency for our disk based sequencing error correction method. From the tables, we can also see that more alignment cases needed to be handled when *k* became smaller. However, the error correction time, which includes alignment time, was still small even when a significant number of alignment cases (e.g., $k = 12$) were processed, which indicates that the binary encoding based alignment technique adopted in our method is efficient. From the tables, we can also see that, although the error correction itself did not take much time, the creation of the BoND-tree took significant amount of time. Note that the index creation time is bound to the efficiency of the index building algorithm given in [11]. To improve the index building efficiency, a bulk loading technique like those in [32, 33] could be developed for the BoND-tree. In fact, such a bulk loading technique has recently been suggested [34]. Alternatively, the index tree could be built at the time during DNA sequencing in an online/streaming fashion, which avoids a separate index tree building process. Once the index tree is built, our method could be implemented as a service for error verification and correction to efficiently process user's requests on checking some suspicious error positions in a genome sequence. The built index tree storing *k*-mers and relevant metadata could also be utilized to support other sequence analysis applications such as sequence alignment, terminus searching, and variant detection.

Table 3.3: Correction Time for Simulated 36 bp *E. coli* with Coverage=15

k	Creation Time (minutes)	Correction Time (minutes)	Alignments (%)
12	194.2	33.3	23%
13	216.0	20.0	6%
14	254.1	20.7	2%
15	264.0	21.3	0.6%
16	284.1	25.3	0%

Table 3.4: Correction Time for Simulated 36 bp *E. coli* with k=15

Coverage	Creation Time (minutes)	Correction Time (minutes)	Alignments (%)
10	167.7	14.5	0.6%
15	264.0	21.3	0.6%
20	343.0	38.4	0.6%

We also conducted an experiment using reads with a larger size, i.e., 124 bases. The results are shown in Table 3.5. From the table, we can see that similar results were obtained when using a larger read size, which indicates that our method scales well with the read size.

Table 3.5: Experimental Results for Simulated 124 bp *E. coli* with k = 15 and Coverage=15

Total errors	Corrections	Mis-corrections	Errors kept	Accuracy(%)
820526	820266	5	255	99.97%

Creation Time (minutes)	Correction Time (minutes)	Alignments (%)
361	29.6	0.26%

To examine if our method would have similar performance on other genomic data, we conducted an experiment using *C. elegans* chromosome I data (Genbank: NC003279.8, 15M) with read size = 124, k=16 and coverage=10. We observed that, for 1,491,895 sequencing errors in the reads, our method achieved an accuracy of 99.70%, which demonstrated a similar behavior. The error correction time and the index creation time were 207 and 937 minutes, respectively.

The preliminary results of the work discussed in this chapter were reported in [35, 36].

CHAPTER 4

A Deletion Technique for BoND-tree

DiskBQcor corrects errors in sequencing genome data with the BoND-tree. However, there is a shortcoming for this method presented in Chapter 3. The current method does not correct sequencing errors until all the reads are obtained, i.e., in an offline fashion. As a result, computing resources are idle and wasted during the DNA sequencing process. To develop an online method for sequencing error correction, we need to utilize a deletion technique for the underlying storage structure, the BoND-tree. In this chapter, we present a technique for deleting *k*-mers in the index tree. This technique serves as a preparation for the work to be presented in Chapter 5.

In [37], three deletion strategies are suggested: deletion via reinserting vectors, deletion via reinserting nodes, and deletion via borrow reinsertion. In this chapter, we develop an algorithm to delete a vector from the BoND-tree using the strategy of reinserting nodes.

4.1 Main-deletion procedure

In order to delete a vector, or specifically, to delete an erroneous k-mer from the BoND-tree, the vector needs to be located following a path from the root to a leaf node. The algorithm deletes the vector from the leaf node first, and then checks if the node underflows or not. By saying "underflow", it is the opposite of "overflow", meaning that this node has entries less than the required minimum number. Different strategies can be used to handle the underflows. During this process, the underflow may propagate to the root of the tree, and adjustments on DMBRs along the path are needed. The function *adjust_DMBR* is the same as in [11]. Function *delete_use_link* is described as follows in Algorithm 4.1 to delete a leaf entry.

Algorithm 4.1 Main-deletion procedure

```
Input:  Leaf_entry $\alpha$ to be deleted, BoND-tree
Output:  root RN of BoND-tree
 1: function DELETE_USE_LINK
 2:     Locate the leaf_entry $\alpha$ following a path $p$ by invoking find_entry_block($\alpha$, p)
 3:     if $\alpha$ does not exist in the tree then
 4:         return not_present
 5:     end if
 6:     delete $\alpha$ from its leaf node N
 7:     if N does not underflow then
 8:         invoke function $adjust\_DMBR$()
 9:     else
10:         repeat
11:             put N into a buffer reinsert_buf
12:             delete N from its parent node
13:             make N = parent node
14:         until N does not underflow or N is already root
15:         invoke function adjust_DMBR()
16:         invoke function reinsert_node(reinsert_buf, BoND_tree)
17:     end if
18: end function
```

Step 2 locates the leaf entry that contains α, and follows the path from the root to that entry. Steps 3-5 handle the case where α is not in the tree. Step 6 deletes α from the leaf node. Steps 7-17 discuss whether the leaf node underflows, and if it does, we will put the node to a buffer, and this underflow may propagate to the top of tree. After we reinsert the nodes in the buffer in step 16, we adjust the tree from the bottom to the top.

4.2 Locate Leaf Entry

Before the deletion begins, we need to locate the target vector/*k*-mer. In other words, we need to find the leaf node that contains the vector first. Meanwhile, after the vector has been deleted from the leaf node, a series of impacts may happen to its parent: it may also underflow and the relevant DMBR needs to be adjusted. The same process applies to the grandparent, and up to the root node if necessary. This implies that the whole path from the root to the leaf entry with the target vector needs to be kept for convenience. Function *find_entry_block* uses a breadth-first search on the BoND-tree to find the disk block number of the leaf entry, and also returns the path in two arrays. The first array contains all the block numbers of the tree nodes along the path, and the second array contains the index numbers of the tree nodes among all the siblings at each level.

Algorithm 4.2 Breadth-first search to locate leaf entry

Input: vector α

Output: path from root to leaf containing α

```
 1: function FIND_ENTRY_BLOCK
 2:     if height of tree is 1 then
 3:         if root node contains $\alpha$ then
 4:             return success
 5:         else
 6:             return not_present
 7:         end if
 8:     else
 9:         initialize stack check_list
10:         push root into check_list
11:         while check_list is not empty do
12:             pop node N from check_list
13:             if N contains $\alpha$ then
14:                 put N's block number and index number into the path
15:                 go through every covering entries E of N
16:                 push E to check_list
17:                 if N is one level above leaf and E contains $\alpha$ then
18:                     add E to path
19:                     return success
20:                 end if
21:             end if
22:         end while
23:     end if
24:     return not_present
25: end function
```

Steps 2-7 handle the special case where the tree height is one. In this case, we only need to check whether α is among the vectors in the root. Steps 9-22 is the breadth-first search process starting from the root, traversing all possible nodes whose DMBRs contain α, until α is found, or until there is nothing to traverse. The path is recorded during the process.

4.3 Reinsert nodes

Nodes stored in the buffer *reinsert_buf* from Algorithm 4.1 need to be reinserted into the tree. Each element in the buffer is a structure containing the level and disk block number of each node. Using the level and path information we have discussed in Section 4.2, we are able to find the parent node of each node in the buffer. Function *reinsert_node* reinserts node N into its parent, handles the situation where the parent overflows, and adjusts *DMBR* along the path to the top if necessary.

Algorithm 4.3 Reinsert Nodes

Input: buffer *reinsert_buf* with nodes
Output: root of BoND_tree

```
function REINSERT_NODES
    for each node N in reinsert_buf do
        find sibling S with least overlap enlargement with N
        insert N's every entry to S
        if S overflows then
            split S to N1 and N2, put N1 into S's block
            assign a new node block to N2
        end if
        if parent has only one child then
            make S to root
        end if
    end for
    invoke function adjust_DMBR()
end function
```

Steps 3-4 describe the main idea of the reinsertion procedure - to insert node N into its sibling node. Steps 5-8 handle the overflow condition. Steps 9-11 handle the condition where the root has only one child, and we make that child to the root in these steps. In [37], different heuristics were applied to find the best suitable sibling to insert N. Those heuristics are able to accommodate different situations and to break ties. In our algorithm, for simplicity, we only use Heuristic 1. As long as the node can be correctly deleted, it does not affect our final error correction accuracy.

4.4 Experiment

Experiments were conducted to examine the accuracy, efficiency and effectiveness of the deletion algorithm. The algorithm was implemented in C++. All the experiments were conducted on a Dell PC with a 3.2 GHz Intel Core i7-4790 CPU, 12 GB RAM, 5 TB Hard Drive, and Linux 3.16.0 OS. Since the algorithm is not an approximate one, the accuracy is 100%. Efficiency is measured in terms of the number of disk I/Os, and effectiveness is evaluated through the quality of the resulting BoND-tree, i.e., how the performance is in terms of executing queries.

Three BoND-trees were built, with sets of 25-dimensional *k*-mers of sizes 1M, 2M and 4M respectively. The *k*-mers were all generated randomly from *E.coli* 536. Since this data has a total length of 5.5M, the non-duplicate vector dataset size is a little bit less than the total vector dataset size. In each of the three experiments, we first collected experimental data from the whole tree, and then 50%, 75% and 100% deletions are performed.

Table 4.1 and Figure 4.1 show the comparison of disk utilization and I/Os for 50%, 75% and 100% deletions. While performing deletions, disk I/Os are spent on the four parts: 1) read cost of locating the target vector; 2) cost for adjusting *DMBR*s and writing back to disk; 3) read cost along the path from leaf nodes to the root when leaves underflow; 4) write cost or adjust cost along the path from sibling leaf nodes to the root.

Table 4.1: Number of Disk I/O and Disk Utilization

Total Vec. DB Size	Original	I/O			Utilization after Deletions		
(Distinct Vec. DB Size)	Utilization	50%	75%	100%	50%	75%	100%
1M (902063)	67.05%	3562588	5835147	8310199	46.36%	44.79%	/
2M (1632829)	71.44%	7976160	13052123	17851491	40.39%	45.28%	/
4M (2713546)	71.01%	15781226	26230400	36748479	40.57%	43.92%	/

The disk utilization rate maintains around 40%-50%, which is obviously lower than the normal disk utilization rate. This is reasonable since we keep deleting the vectors from leaf nodes until the

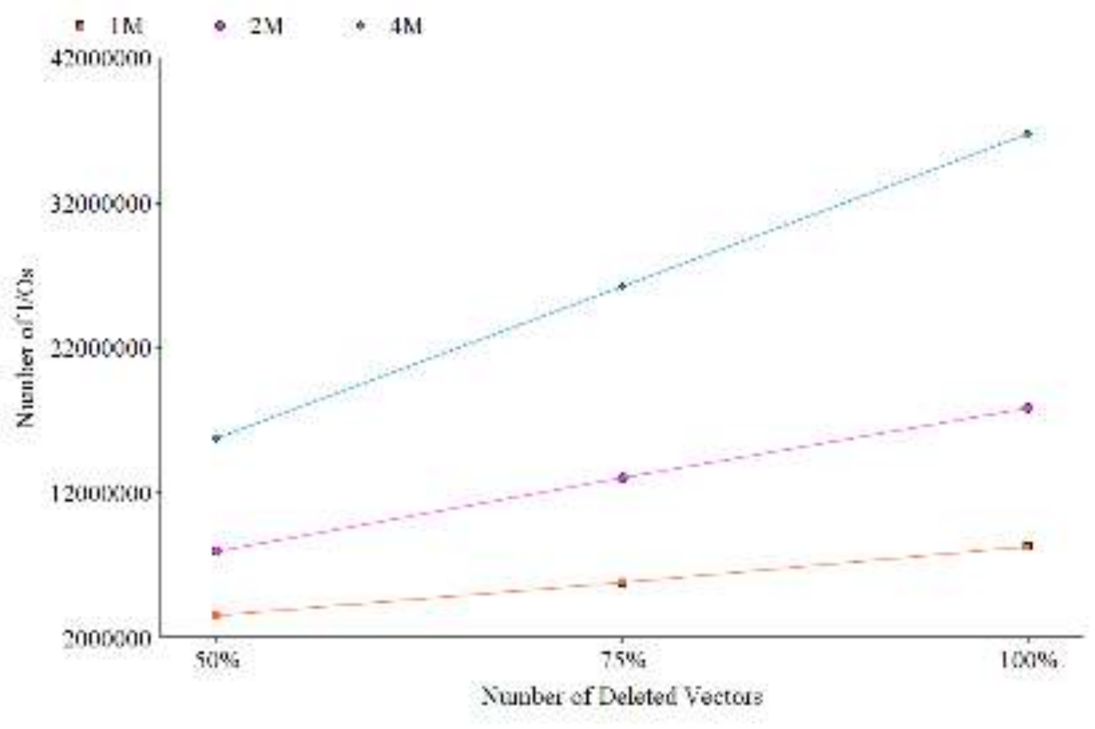


Figure 4.1: Deletion Performance: Number of I/Os

nodes reaches the minimum utilization, which was set to 30%. We then delete the underflowing leaf nodes from their parent non-leaf nodes, and repeat the same process. In the worst case, every leaf node and every non-leaf node may have only 30% of entries. We can see from the results that the disk utilization is lower at 75% deletions than at 50% deletions for vector dataset size 1M, but higher at 75% deletions than at 50% deletions for vector dataset sizes 2M and 4M. In fact, the utilization is maintained between the minimum and maximum utilizations, and there is no inherent relationship between the number of vectors deleted and the disk utilization.

Table 4.2: Box Query Performance

Total Vec. DB Size	I/O		
(Distinct Vec. DB Size)	0%	50%	75%
1M (902063)	4.01	8.27	14.07
2M (1632829)	5.01	9.51	15.42
4M (2713546)	5.01	12.05	18.18

Table 4.2 and Figure 4.2 show the comparison of box query performance. In each experiment,

the result shows the average number of I/Os for 100 random box queries performed on the resulting tree. Note that all the sets of queries were conducted at a tree height of 4 even when 75% of deletions are performed, so that the tree height does not interfere of the number of I/Os.

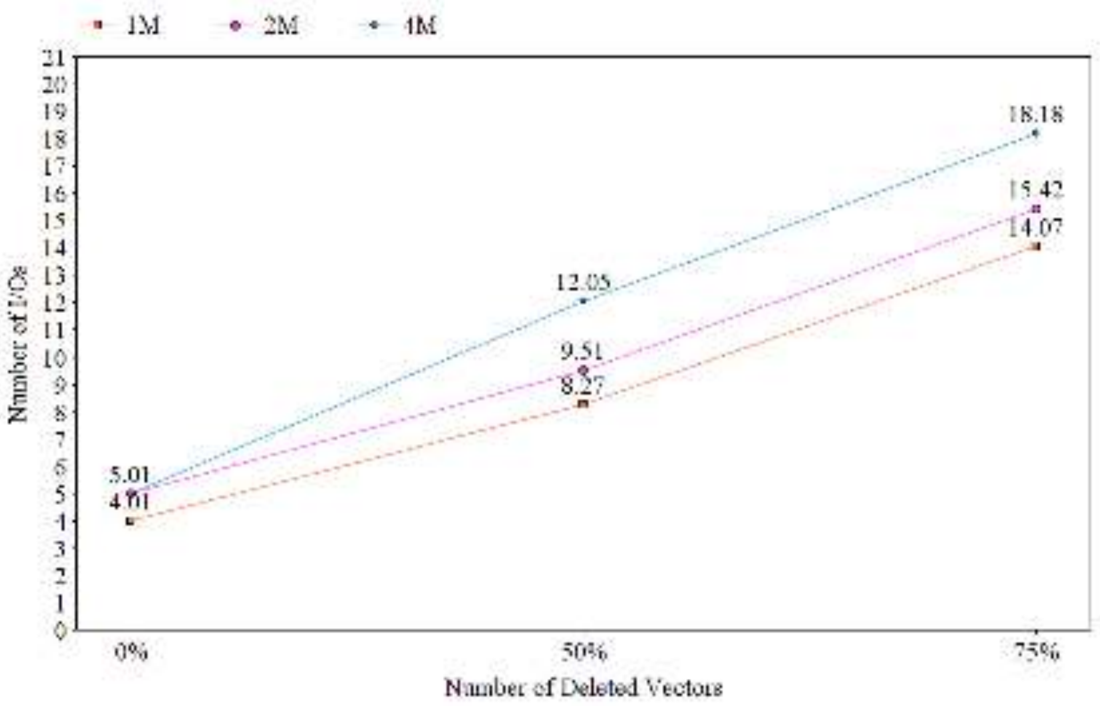


Figure 4.2: Average I/O performance

Our deletion algorithm does decrease the query performance to a reasonable degree, and it decreases more when performing more deletions. However, we plan to perform deletions for erroneous *k*-mers, which constitutes only a tiny portion of the whole tree. As long as the performance is within the tolerance, we still consider accuracy as the most important factor.

CHAPTER 5

An Online Approach for Sequencing Error Correction

In this chapter, we propose an online sequencing error correction method. It extends the ideas from DiskBQcor in Chapter 3 by adding another online error analyzing phase while the sequencing reads arrive in a streaming fashion. A detailed discussion about correction accuracy estimation is also presented in Section 5.3, followed by an experimental evaluation in Section 5.4.

5.1 Preliminaries

In reality, a sequencer generates reads for a DNA sequence in a relatively slow streaming fashion. For example, Illumina MiSeq takes about 24 hours to generate a 5Gb dataset of 150bp reads [38]. This fact has inspired us to consider the following question: can we begin the error correction as soon as sufficient reads are generated, instead of waiting until all the reads are produced, so that the two processes of DNA sequencing and error correction can be integrated to better utilize the sequencing and computer resources?

5.1.1 Observations

Our technique introduced in Chapter 3 corrects sequencing errors after all the reads are obtained. It presents a limitation on utilizing both computing resources and time. While reads are still being generated, the computing resources are idle and wasted in waiting for them. Meanwhile, a large *k*-mer set has to be loaded into disk all in once, and corrected altogether, which leads to a longer time delay.

Using a large, complete BoND-tree can certainly give us a higher accuracy level on error correction since the information for the entire read set is available. However, it has to handle a large tree when performing insertions and queries, resulting in a non-optimized efficiency.

In fact, correcting erroneous *k*-mers does not have to wait until the whole read set is obtained. The voting approach distinguishes the correct and incorrect bases through their relative counts. Thus it can be done as long as the two kinds of bases have unbalanced occurring frequencies. Given a reasonable minimum coverage at which a correct base is already distinguishable from the incorrect ones, the voting method can be applied, although the reliability may be affected to some extent. In an ideal circumstance, error corrections can start earlier and accuracy is gradually improved when the tree grows. In this case, a better utilization of the computing resources is achieved, since the DNA sequencing and error correction are performed in parallel.

Moreover, the index tree with correct *k*-mers can be obtained as soon as the DNA sequencing is completed. This final tree can be used for sequence analysis applications such as variants detection and error verification. If we want to maintain an index tree for correct k-mers only, this procedure can continue to proceed on top of the correct index tree.

5.1.2 Assumptions

Based on the above observations, we introduce an online approach for (sequencing) error correction. The assumptions are:

- A DNA sequencer produces reads one by one.
- The generated reads are randomly distributed and covering the whole underlying genome sequence.
- Each read is decomposed into fixed length k-mers and inserted into a BoND-tree for the storage and future processing purpose.
- The sequencer may produce sequencing erroneous bases at some positions of a read.

- The error rate is low, compared to the correct bases in reads.
- A high percentage of errors (but may not be all of them) are corrected.
- The quality of the BoND-tree, in terms of keeping correct k-mers, is gradually improving as the tree grows bigger.

5.2 The Method

Correcting errors as the sequencing reads arriving one by one will be done in two phases. First, before the minimum coverage (threshold) is reached, the reads are decomposed into *k*-mers and inserted into the BoND-tree (Section 5.2.2, Insertion Phase). Then, once the minimum coverage is reached in the tree, we can start to correct the errors (Section 5.2.3, Correction Phase).

5.2.1 Measures

Coverage Statistics

Given a sample DNA sequence, we assume that the sequencer outputs sequencing reads one by one. The average coverage of each base at a specific time during the sequencing process is calculated through:

$$rc = \frac{N * l}{G} \tag{5.1}$$

in which N is the total number of reads arrived, l is the length of one read, and G is the whole genome size. rc is an important indicator of the current loading situation in the tree. Other similar measures include: the (current) maximum coverage Mc, which reflects the peak k-mer coverage of a base; the (current) minimum *k*-mer count mc, which reflects the smallest coverage of a base. Ideally, we want all the *k*-mers to be distributed evenly. If some regions have very high Mc, the weak coverage regions may have nearly no coverage, and those weak regions cannot be captured since *k*-mers for such regions are not in the tree at all.

Changing rates

The count change rate cr for two consecutive shifted *k*-mers $k1$ and $k2$ is calculated as:

$$\frac{\Delta c}{\Delta x} = \frac{count(k2) - count(k1)}{1} = count(k2) - count(k1) \tag{5.2}$$

In this equation, Δx denotes the number of steps between two *k*-mers, which is 1 for two consecutive *k*-mers $k1$ and $k2$. A large negative cr may indicate an error has occurred (i.e., k2 hits an erroneous position (from left to right)), and may also indicate entering a low coverage area from a high coverage area; while a large positive cr may indicate k2 entering a correct area without erroneous positions, and may also indicate entering a high coverage area from a low coverage area.

The region change rate rr for two consecutive shifted *k*-mers $k1$ and $k2$ is calculated as:

$$\frac{\Delta r}{\Delta x} = size(readId\_set(k2)) - size(readId\_set(k1)) \tag{5.3}$$

where $readId\_set(k)$ denotes the set of ids for the reads containing *k*-mer k. The reverse region change rate $rr(-1)$ for two consecutive shifted *k*-mers $k1$ and $k2$ is calculated through:

$$\frac{\Delta(-r)}{\Delta x} = size(readId\_set(k1)) - size(readId\_set(k2)) \tag{5.4}$$

We only consider high-count areas in read r. Low-count areas can be handled using the count change rate in Equation(5.3). There are several cases to consider:

- Case 1: small rr and small $rr(-1)$: indicates that $k2$ is correct without a duplication in another region.
- Case 2: large rr and small $rr(-1)$: indicates that $k2$ is correct but is duplicated in another region not in r, and we are entering the correct duplicate area.

- Case 3: small rr and large $rr(-1)$: indicates that $k2$ makes us leave the correct duplicate area and return to the non-duplicate area ($k1$ is still duplicated).

- Case 4: large rr and large $rr(-1)$: At the odd time $(2n - 1)$, $k2$ has an error at its *k*-th position and is duplicated in another region not in r, and we are entering the incorrect duplicate area. At the even time $(2n)$, $k1$ has an error at its first position and is duplicated in another region, $k2$ makes us leave the incorrect duplicate area and return to the non-duplicate area in read r.

5.2.2 Insertion Phase

In the Insertion Phase, our goal is to decompose the reads generated by the sequencer into *k*-mers, and insert the *k*-mers into the tree, without considering whether a *k*-mer is a correct one or not. This is the original tree building phase.

Algorithm 5.1 Insertion Phase Procedure

```
Input: reads, minimum coverage $C$
Output: BoND-tree $BT$, k-mers of first part inserted
1: function INSERT_BEFORE_MINIMUM
2:     repeat receive a new read $r$
3:         Decompose $r$ into shifted overlapping k-mers
4:         for each shifted k-mer k from left to right do
5:             Insert k into BoND-tree $BT$
6:         end for
7:         update $rc$
8:     until $rc >= C$
9: end function
```

In Algorithm 5.1 Insertion Phase Procedure, lines 2-8 are a loop to continuously insert *k*-mers into the BoND-tree. Line 3 decomposes the arriving read into overlapping *k*-mers. Lines 4-5 insert all the *k*-mer generated from the read into the tree.

5.2.3 Correction Phase

In the Correction Phase, our goal is to continue to decompose the generated reads into *k*-mers, and compare them with those already inserted in the tree. Specifically, through analyzing on the *k*-mer abundance (counts), the separation between the high-abundance *k*-mers and the low-abundance *k*-mers will become clear. A suspicious error position is typically at the boundary between these two types of *k*-mers. The base at each suspicious error position can be checked (verified) by DiskBQcor to see if it is indeed an error. If so, DiskBQcor can find the correct base to replace the erroneous base at the position. We then update the relevant high-abundance area and the relevant low-abundance area on the read. The Correction Phase is described in Algorithm 5.2.

In Algorithm 5.2 Correction Phase Procedure, lines 2-4 are similar to those in the Insertion Phase, except that we keep the count of each overlapping *k*-mer to find an untrusted area, and also keep the read ID sets to correct the error if such an area exists. Lines 5-7 find the low-count and high-count area lists. In lines 8-17, we correct the errors in each low-count area.

Line 16 invokes an error correction function *error_correct()*. For each base (suspicious position) in LL list, we utilize DiskBQcor to verify if it is indeed an error. If so, DiskBQcor can find the correct base to replace the erroneous base at the position. After the base is successfully corrected, we update every *k*-mer overlapping the position by deleting it from the BoND-tree first, and reinserting the correct *k*-mer into the tree. Meanwhile, we search for its associated read ID set, and correct all the reads in the set.

5.3 Correction Accuracy Estimation

Once the method is developed, correction accuracy is largely affected by the threshold used to distinguish erroneous reads from correct ones. With a larger low count threshold, the initial insertion phase takes longer, so that more errors are inherited from this phase; while a smaller low count threshold makes the initial insertion phase shorter. However, if a correct base is falsely "corrected"

Algorithm 5.2 Correction Phase Procedure

```
Input: reads, minimum coverage $C$
Output: BoND-tree $BT$, $k$-mers inserted
 1: function INSERT_BEFORE_MINIMUM
 2:     while receive a new read $r$ do
 3:         Decompose $r$ into shifted overlapping $k$-mers
 4:         Insert these $k$-mers into $BT$ and find their counts, read ID sets and calculate count
            change rates $rc's$ and region change rate $rr's$ for $r$
 5:         Use the count change rates $cr's$ to find areas boundaries with dramatic changing $cr's$
 6:         Let LL = the list of low-count areas in $r$ ordered from left to right
 7:         Let HL = the list of high-count areas in $r$ ordered from left to right
 8:         while LL is not empty do
 9:             Let Al[l1,l2] be an area in LL that
10:              (1) has a left adjacent high-count area in HL, or
11:              (2) has a right adjacent high-count area in HL, or
12:              (3) is the first area in LL if no area in LL satisfies (1) or (2)
13:             Let (p,direction,pk) = find_start_error_position(Al[l1,l2], HL, $BT$)
14:             more_error_in_Al = true
15:             while more_error_in_Al do
16:                 error_correct(Al[l1,l2], $BT$)
17:             end while
18:         end while
19:         Use the region change rate $rr's$ to find the boundary positions of incorrect duplicate
            areas in read $r$ based on a dramatic changing $rr$ and keep all incorrect duplicate areas in list
            DL
20:     end while
21: end function
```

to another base, more correct bases will become false negative ones due to a chain reaction.

There are three parameters required to calculate the threshold: k, length of k-mer; l, length of read; and C, current average coverage. We have already demonstrated an appropriate determination of the length of k-mers in Chapter 3 according to Quake's method. Now, given a specific coverage, we want to find the threshold based on the distribution of these k-mers.

5.3.1 Sensitivity

Let T_i be a random k-mer starting at the position i of a genome sequence G, and C be the current coverage. In most cases where T_i is not sampled from the very beginning or end of the sequence, only the reads starting from positions $i + k - l$ to i contain T_i. Thus T_i can be obtained from $l - k + 1$ reads. When G's size g is big enough to make $g >> l$, the probability that T_i is sampled s times can be calculated by

$$P(s) = \binom{n}{s}(\frac{l-k+1}{g})^s(1 - \frac{l-k+1}{g})^{n-s} \tag{5.5}$$

where n is total number of reads, calculated by

$$n = \frac{gC}{l} \tag{5.6}$$

When g is very large, the distribution can be approximated by the Poisson distribution, with a mean value

$$\begin{aligned} \lambda &= n * \frac{l-k+1}{g} \\ &= C * \frac{l-k+1}{l} \end{aligned} \tag{5.7}$$

In fact, the coverage of sequencing reads follows the Poisson distribution in theory in the sense that:

- The sampling process is random.
- The occurrence of one sampling read does not affect the second one. Thus, they are independent.
- The whole genome sequence length is close to infinite, comparing to one sample read.

Similar conclusions have been made in [23,39–41]. In some sequencers, for example, Illumina, the sequencing process has substantial biases, which adds a variance [42]. Some studies model it as the Gaussian distribution [23]. In our discussion, we adopt the Poisson distribution because our Correction phase may start earlier from a coverage as small as 5. In either way, the accuracy estimations have similar results.

For a coverage of 5, the distribution of k-mers, without considering errors, can be plotted in Figure 5.1. When x-axis has a value less or equal to 0, it means that some of the k-mers are not covered at all. Let $k = 15$, $l = 36$, then $\lambda = 3.06$. If we start the Correction Phase at this coverage and set threshold to 1, we are claiming that those k-mers with more than one count to be correct. The proportion of k-mers that we have correctly identified as incorrect k-mers (sensitivity) is equal to the possibility that a random k-mer has more than one count, i.e., $P(x > 1) = 0.8096$.

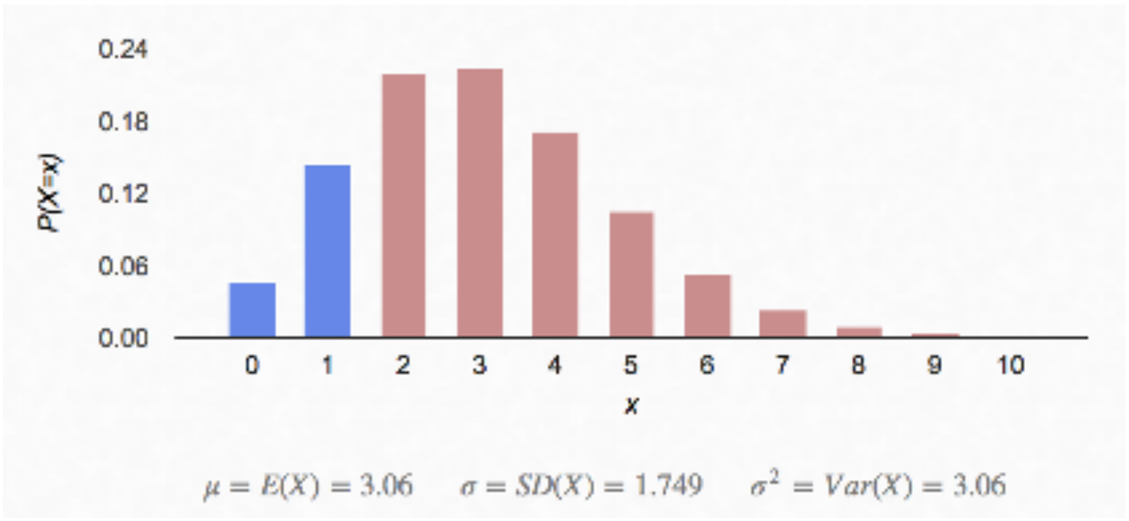


Figure 5.1: Distribution when coverage is 5

Similarly, if we start the Correction Phase at a coverage of 10, and make the threshold to 2, the theoretic sensitivity can be estimated as $P(x > 2) = 0.9428$. See Figure 5.2.

On the other hand, when the algorithm suggests that some k-mer T's count falls into the "un-trusted" area, we apply the Voting Approach for error correction. It is possible that T has a smaller count only because of a low coverage. In this case, the Voting Approach is unable to find another base replacement for the position. As a result, this position is not "corrected", which may be the correct result. Therefore, the sensitivity value may be much higher.

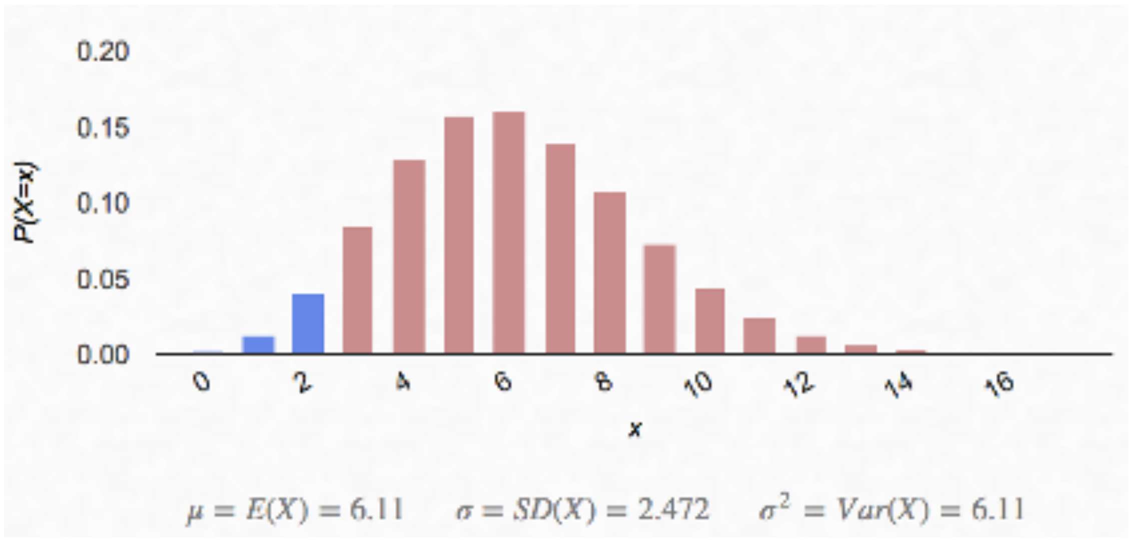


Figure 5.2: Distribution when coverage is 5

5.3.2 Specificity

In order to estimate the specificity, we look at one specific base in the genome. The possible number of times it has been covered follows the Poisson distribution too:

$$P(k) = e^{-\lambda}\frac{\lambda^k}{k!} \tag{5.8}$$

in which λ is the average coverage times of the genome sequence, and *k* is the possible coverage times we need to know.

While we are at a certain average coverage C, and the threshold value is M, the false positive ones are those *k*-mers with a frequency greater than M, but are actually generated as a variance of correct *k*-mers because of errors. Similar to Section 5.3.1, in order to calculate the specificity, we need to generate a curve chart with x-axis as the occurrence times of *k*-mers, and y-axis as the possibility/proportion of the erroneous *k*-mers that appear that many times.

Let us assume that a *k*-length area is *k*-mer T_t originally, and turns to T because of t errors. If each base has an error rate of p, the possibility that we get *k*-mer T with specified t errors, instead of T_t is

$$P_{err}(t) = (\frac{p}{3})^t (1-p)^{k-t} \tag{5.9}$$

The possibility that a *k*-mer T is generated by another *k*-mer T_t, with t bases falsely sequenced is

$$P_{occ}(t) = \frac{3^t \binom{k}{t}}{4^k} \tag{5.10}$$

Knowing a *k*-mer is sampled s times, the possibility that r of them is T, $s-r$ of them is T_t can be calculated using Equations 5.9 and 5.10:

$$P_r(s,r) = \sum_{t=1}^{k} P_{occ}(t) P_{sam}(s) (\binom{s}{r} P_{err}(t)^r (1 - P_{err}(t))^{s-r}) \tag{5.11}$$

In [43], the probability that one *k*-mer T appears r times has been fully studied using the above possibilities. Because the number of sampling times for a normal *k*-mer follows the Poisson distribution as discussed in Section 5.3.1, there are 3 parameters in Equation 5.11. The exact overall possibility is hard to give by a simple function. Instead, let us consider the basic condition where there is only one error in T_t, and all of the *k*-mers are sampled λ times using the mathematical expectation, i.e., $t = 1$ and $s = \lambda$. Equation 5.11 can be simplified to

$$P_r(r) = P_{occ}(1) (\binom{\lambda}{r} P_{err}(1)^r (1 - P_{err}(1)^{\lambda-r}) \tag{5.12}$$

To further simplify the equation, suppose that each region $[i, i+k-1]$ yields only 1 error *k*-mer at most. There are

$$m = P_{occ}(1) * g \tag{5.13}$$

kinds of *k*-mers with 1 base distance with target *k*-mer T in the dataset, and the possibility that this *k*-mer has x times of coverage equals to

$$P^{'}(x) = \binom{m}{x} P_r(1)^x (1 - P_r(1))^{m-x} \tag{5.14}$$

Through Equation 5.14, when m is large, this possibility follows the Poisson distribution. Since this equation measures the possibility of one target *k*-mer, the possibility for all possible *k*-mers follows

$$P(x) = 4^k \binom{m}{x} P_r(1)^x (1 - P_r(1))^{m-x} \tag{5.15}$$

with the following mean value

$$\lambda = 4^k m P_r(1) \tag{5.16}$$

For a coverage of 5, $\lambda = 0.0728$. Let $k = 15$. If we set the threshold M to 1, $P(x > 1) = 0.0025$, and $P(x = 1) = 0.0677$, the proportion of *k*-mers that we have correctly identified as correct *k*-mers (specificity) is equal to

$$\frac{P(x = 1)}{P(x >= 1)} = 0.9640 \tag{5.17}$$

Similarly, for a coverage of 10, if we make threshold M to 2, $\lambda = 0.1445$, theoretic specificity will be

$$\frac{P(x = 1) + P(x = 2)}{P(x >= 1)} = 0.9966 \tag{5.18}$$

On the other hand, our algorithm in the Correction Phase requires a set of continuous low-count *k*-mers to decide a low-count area, and that if one or more of the *k*-mers in this area has abnormally higher count, it may affect our determination of low-count areas, and thus makes some of the error not able to be correctly identified. The possibility that one or more *k*-mers has a higher count in a "low-count" area can be calculated by:

$$P_e = 1 - (\frac{P(1 <= x <= M)}{P(x >= 1)})^k \tag{5.19}$$

When $k = 15$ and a coverage of 10, a threshold of 2, P_e is only 0.049. However, for a coverage of 5, a threshold of 1, P_e can be as high as 0.42. That is to say, once we have found out a suspicious position, the specificity level can reach 96.40%. However, in order to successfully find out the falsely sampled base, the algorithm's ability to consider some of the conditions described above is much more important. For example, we still make the area to qualify as "low-count" regions where one of the overlapping *k*-mers has a high count, and all others have low counts.

5.4 Experiment

Experiments were conducted on a Dell PC with a 3.2 GHz Intel Core i7-4790 CPU, 12 GB RAM, 5 TB Hard Drive, and Linux 3.16.0 OS.

Genome data was collected from *E. coli* 536 (GenBank: NC008253, 5.5 M). The dataset and environment are both the same as what we used in Chapter 3. Let $k = 15$, $readlength = 36$. The threshold of counts to separate a low-abundance area with a high-abundance area is set to a ratio of 0.2, i.e., when the coverage is 10, if a *k*-mer *K* has a count of less than or equal to 2, it belongs to a low-abundance area; otherwise it is classified to a high-abundance area.

In experiment 1, the coverage (times) was set to 40. We set the minimum coverage as a threshold, and perform the Insertion Phase until the minimum coverage is reached, and then proceed with the Correction Phase until the coverage of 40 times is reached. Table 5.1 shows the comparison of the results when we set the minimum coverage from 5 to 9. From the table we can see that while the threshold is increasing, the accuracy is decreasing, which is obvious because we arbitrarily insert the first part of reads without correcting them before the minimum coverage is reached; yet mis-corrections is decreasing when the threshold is increasing.

Table 5.1: Correction Accuracy, low count ratio=0.2, no repeat

Minimum Coverage (Threshold)	Corrections (TP)	Mis-corrections (FP)	Errors Kept (FN)	Untrusted (trimed)	Sensitivity	Specificity
5	864148	152008	185837	50096	0.8230	0.9993
6	845307	119307	207969	42152	0.8026	0.9995
7	824330	103459	231143	37837	0.7810	0.9995
8	802396	95665	254884	34995	0.7589	0.9996
9	779951	92190	278650	33385	0.7368	0.9996

We are able to correct the first several times of coverages by correcting and inserting them again into the tree, which will actually provide us with a more beneficial set of data. In fact, the overall accuracy level can be improved by running the whole process again. That is to say, to repeat the Correction Phase for all the reads after all the *k*-mers have been inserted for the first time. In experiment 2, the coverage times were still set to 40. Similar to experiment 1, we set the minimum coverage as a threshold, and perform the Insertion Phase until the minimum coverage is reached (let R be the set of these reads), and then proceed with the Correction Phase until the coverage of 40 times is reached. After that, we run the reads in set R again with the Correction Phase algorithm. Table 5.2 shows the comparison of the result when we set the minimum coverage from 5 to 9. From the table we can see that the sensitivity increases when the threshold increases. The increase rate of sensitivity is higher at first, and becomes more steady when the minimum coverage is set to 9. Figure 5.3 plots the sensitivity change with the minimum coverage change.

Table 5.2: Correction Accuracy, low count ratio=0.2 with repeat

Minimum Coverage (Threshold)	Corrections (TP)	Mis-corrections (FP)	Errors Kept (FN)	Untrusted (trimed)	Sensitivity	Specificity
5	983910	160568	59908	54623	0.9426	0.9993
6	989022	129783	56900	47530	0.9456	0.9994
7	991979	115810	54984	44038	0.9475	0.9995
8	994004	109934	53541	42088	0.9489	0.9995
9	995218	108196	52511	41311	0.9499	0.9995

In order to see how the threshold of counts to separate a low-abundance area from a high-abundance area affects the accuracy of the result, another experiment was conducted, keeping the minimum coverage to 7, and changing the threshold ratio. Table 5.3 shows the results. Note

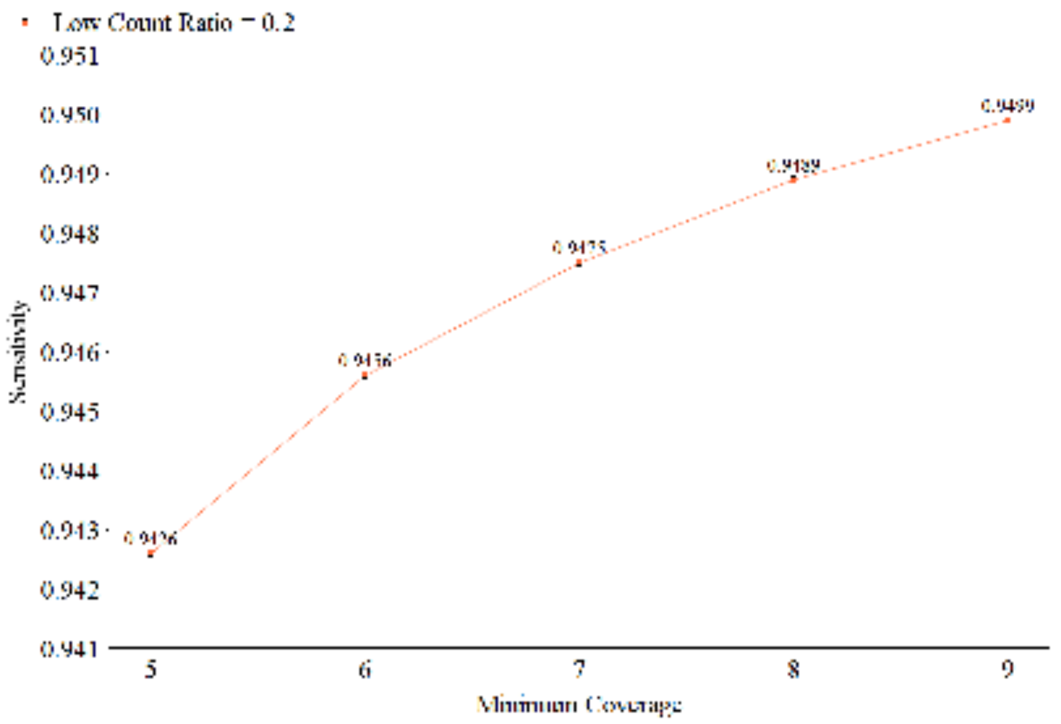


Figure 5.3: Sensitivity, low count ratio = 0.2

that the ratio starts at 0.15 because we want the Correction Phase to truly operate as soon as the coverage reaches 7, in that case the threshold count is 1.

Table 5.3: Correction Accuracy, minimum coverage = 7, with repeat

Ratio (Threshold)	Corrections (TP)	Mis-corrections (FP)	Errors Kept (FN)	Untrusted (trimed)	Sensitivity	Specificity
0.15	993609	88933	56005	37777	0.9466	0.9996
0.20	991979	115810	54984	44038	0.9475	0.9995
0.25	982929	212117	55262	68075	0.9468	0.9990

From Table 5.3 we can see that the ratio of 0.20 is the best for the minimum coverage of 7, although the difference of the sensitivity result between the three ratios is less than 0.001. Figure 5.4 plots the sensitivity change with the ratio change.

The overall accuracy is comparable to existing error correction methods. [12] has a higher sensitivity of 0.997 and a lower specificity of 0.688. Other methods all have a specificity of almost 1.0. According to BLESS [41], Quake has a sensitivity of 0.838, and BLESS has a sensitivity of

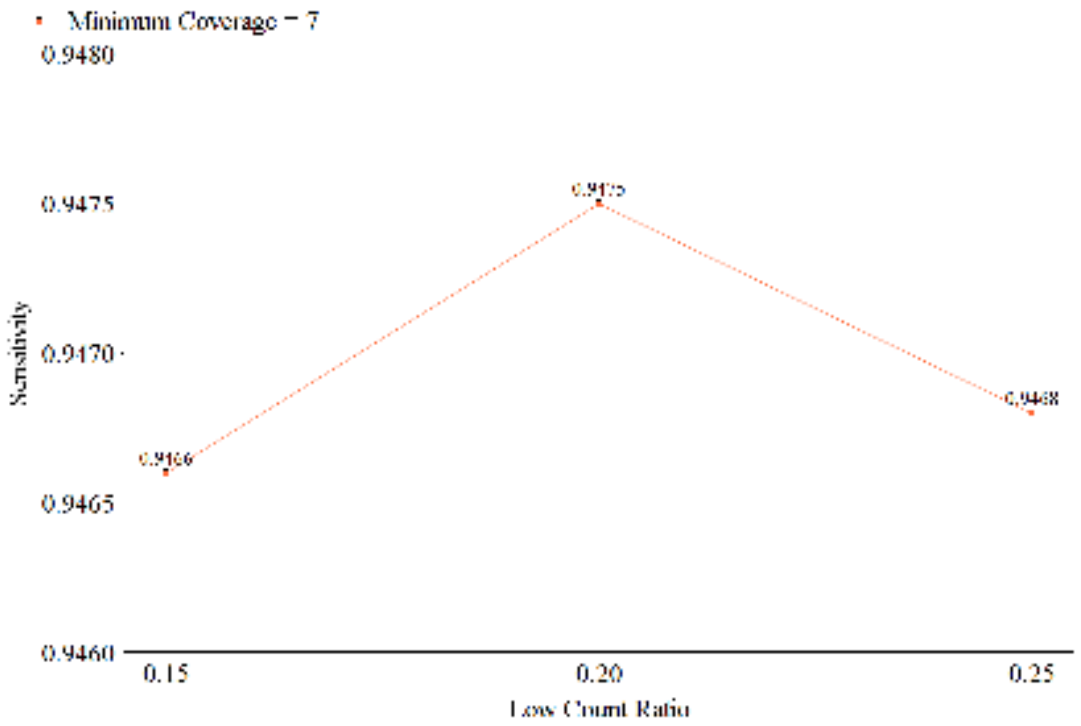


Figure 5.4: Sensitivity, minimum coverage = 7

0.968.

CHAPTER 6

Conclusions

The dramatic increase in DNA sequencing capacity over the last decade has quickly turned biology into a data-intensive science. Areas as diverse as human medicine, microbial ecology, and basic molecular biology are undergoing a rapid transformation as DNA sequencing becomes quick, easy, and inexpensive. However, current sequencers suffer from the problem of having high random per-base error rates. Hence, sequencing error correction is crucial to many sequence analysis applications in bioinformatics. Existing sequencing correction techniques cannot scale well to large datasets due to their requirements on huge expensive computer memory space.

To overcome the above limitation, we present a novel disk based sequencing error correction method, DiskBQcor, in this thesis. Although DiskBQcor has an improved scalability, comparing to the main memory based methods, it also suffers from a drawback, i.e., not fully utilizing computing resources when the DNA sequencing is in process. To overcome this drawback, we extend the way to apply DiskBQcor to make it an online sequencing error correction method. The main contributions of our work are summarized as follows:

- The BoND-tree, which is a recently-developed index structure on disk [11], is utilized in our method to perform sequencing error verification and correction. Since input *k*-mers and their relevant medata are stored on disk with the BoND-tree that supports efficient processing of box queries, our method provides an efficient way for error verification and correction on disk in addition to the benefit of scalability warranted by inexpensive disk space.

- A special set of box queries is designed to efficiently retrieve relevant *k*-mers and their counts from the BoND-tree so that suspicious sequencing errors can be verified and corrected even if a suspicious *k*-mer is repetitive in the target genome sequence.

- A comprehensive vast majority voting mechanism and its relevant formulas are derived to effectively determine sequencing errors for both repetitive and non-repetitive suspicious *k*-mers.

- An efficient alignment strategy is adopted in our method to handle extreme cases when the vast majority voting mechanism does not work. In particular, a special binary encoding based technique is applied to efficiently locate a pair of alignable segments from two target sequencing reads.

- A vector deletion algorithm for the BoND-tree is suggested as a step stone for the online sequencing error correction method. Experiments demonstrates that this algorithm is both efficient and effective.

- A *k*-mer abundance analysis technique is proposed on top of DiskBQcor to analyze the low-abundance *k*-mer area and verify the bases on the area boundaries. DiskBQcor is able to find the correct base to replace the erroneous base at the position.

- An online sequencing error correction approach is discussed, which starts the *k*-mer abundance analysis at the early stage so that a BoND-tree with correct k-mers can be obtained as soon as the DNA sequencing is done. This final tree can be used for genome sequence analysis applications such as variants detection. If we want to maintain a BoND-tree with correct *k*-mers only, this procedure can proceed with the correct index tree.

- Extensive experiments were conducted on real genomic datasets to evaluate the performance of each of our proposed methods. The results demonstrate that the proposed methods are quite promising in achieving high accuracy for error correction with reasonable efficiency, besides the scalability benefit warranted by a disk based approach.

To our knowledge, the presented methods are the first in the field that utilizes a disk based index tree to achieve scalability and efficiency for sequencing error correction. It can be used to locate sequencing errors, and to verify and correct the errors in an online/streaming fashion. Furthermore, memory restrictions with existing memory-based error correction methods require that only a summary of the reads (typically a combined summary of the *k*-mers and associated quality information) be kept in memory, limiting the opportunity to make use of positional information and sample metadata (e.g., sequencing chemistry, machine version) used to generate the data. The approaches adopted in our method to use a BoND-tree storing a large set of *k*-mers along with their relevant metadata not only creates a potential to use the extra metadata in error correction but also can benefit a number of other sequence analysis applications such as local alignment searching, sequence assembly, and terminus searching.

Our future research includes incorporating base quality scores into our error correction methods, improving the accuracy when localizing error regions, conducting alignment on reads, developing an associative correction strategy to correct sequencing errors in the initial set of reads for the online method, exploring our methods in the Hadoop/MapReduce environment, and applying our BoND-tree storing *k*-mers and their relevant metadata to other sequence analysis applications.

Bibliography

[1] Michael L. Metzker. Sequencing technologies-the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2010.

[2] Todd J. Treangen and Steven L. Salzberg. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, 13(1):36–46, 2012.

[3] Phillip E.C. Compeau, Pavel A. Pevzner, and Glenn Tesler. How to apply *de bruijn* graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[4] Robert C Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[5] Stefan Kurtz, Apurva Narechania, Joshua C. Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9(1):517, 2008.

[6] Michael L Metzker. Sequencing technologies?the next generation. *Nature reviews genetics*, 11(1):31–46, 2010.

[7] Xiao Yang, Sriram P. Chockalingam, and Srinivas Aluru. A survey of error-correction methods for next-generation sequencing. *Briefings in Bioinformatics*, 14(1):56–66, 2013.

[8] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[9] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C Titus Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one*, 9(7):e101271, 2014.

[10] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[11] Changqing Chen, Alok Watve, Sarah Pramanik, and Qiang Zhu. The BoND-Tree: An efficient indexing method for box queries in nonordered discrete data spaces. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2629–2643, 2013.

[12] Qingpeng Zhang, Sherine Awad, and C Titus Brown. Crossing the streams: a framework for streaming. *PeerJ PrePrints*, 2015.

[13] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[14] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. The ND-tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. *Proc. of The 29th International Conference on Very Large Data Bases*, pages 620–631, 2003.

[15] M. Patella, P. Ciaccia, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Proc. of The 23rd International Conference on Very Large Data Bases*, pages 1241–1253, 1997.

[16] Caetano Traina Jr., Agma Traina, Christor Faloutsos, and Bernhard Seeger. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260, 2002.

[17] Xia Cao, Shuai Cheng Li, and Anthony K.H. Tung. Indexing DNA sequences using q-grams. *Database Systems for Advanced Applications*, pages 4–16, 2005.

[18] Xia Cao, Beng Chin Ooi, Anthony K.H. Tung, Hwee Hwa Pang, and Kian-Lee Tan. DSIM: A distance-based indexing method for genomic sequences. *Proc. of The 5th IEEE Symposium on Bioinformatics and Bioengineering*, pages 97–104, 2005.

[19] Songbo Huang, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Indexing similar dna sequences. *Algorithmic Aspects in Information and Management*, pages 180–190, 2010.

[20] Tamer Kahveci and Ambuj K. Singh. An efficient index structure for string databases. *Proc. of The 27th International Conference on Very Large Data Bases*, pages 351–360, 2001.

[21] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, page 20, 2013.

[22] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[23] David R. Kelley, Michael C. Schatz, Steven L. Salzberg, et al. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):R116, 2010.

[24] Li Song, Liliana Florea, and Ben Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*, 15:509, 2014.

[25] Haixiang Shi, Bertil Schmidt, Weiguo Liu, and Wolfgang Müller-Wittig. A parallel algorithm for error correction in high-throughput short-read data on cuda-enabled graphics hardware. *Journal of Computational Biology*, 17(4):603–615, 2010.

[26] Jan Schröder, Heiko Schröder, Simon J. Puglisi, Ranjan Sinha, and Bertil Schmidt. Shrec: a short-read error correction method. *Bioinformatics*, 25(17):2157–2163, 2009.

[27] Leena Salmela. Correction of sequencing errors in a mixed set of reads. *Bioinformatics*, 26(10):1284–1290, 2010.

[28] Leena Salmela and Jan Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[29] Páll Melsted and Bjarni V Halldórsson. Kmerstream: Streaming algorithms for k-mer abundance estimation. *Bioinformatics*, 30(24):3541–3547, 2014.

[30] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. *ACM Transactions on Database Systems (TODS)*, 31(2):439–484, 2006.

[31] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Transactions on Information Systems (TOIS)*, 24(1):79–110, 2006.

[32] Hyun-Jeong Seok, Gang Qian, Qiang Zhu, Alexander R Oswald, and Sakti Pramanik. Bulk-loading the nd-tree in non-ordered discrete data spaces. *Proc. of The 13th International Conference on Database Systems for Advanced Applications*, pages 156–171, 2008.

[33] Gang Qian, Hyun-Jeong Seok, Qiang Zhu, and Sakti Pramanik. Space-partitioning-based bulk-loading for the nsp-tree in non-ordered discrete data spaces. *Proc. of The 19th International Conference on Database and Expert Systems Applications'08*, pages 404–418, 2008.

[34] Dong Yoon Choi, A. K M Tauhidul Islam, Sakti Pramanik, and Qiang Zhu. A bulk-loading algorithm for the bond-tree index scheme for non-ordered discrete data spaces. *Proc. of The 25th International Conference on Software Engineering and Data Engineering*, pages 123–128, 2016.

[35] Yarong Gu, Xianying Liu, Qiang Zhu, Youchao Dong, C Titus Brown, and Sakti Pramanik. A new method for dna sequencing error verification and correction via an on-disk index tree. *Proc. of The 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 503–504, 2015.

[36] Yarong Gu, Qiang Zhu, Xianying Liu, Youchao Dong, C Titus Brown, and Sakti Pramanik. Using disk based index and box queries for genome sequencing error correction. *Proc. of The 8th International Conference on Bioinformatics and Computational Biology*, pages 69–76, 2016.

[37] Hyun-Jeong Seok, Qiang Zhu, Gang Qian, Sakti Pramanik, and Wen-Chi Hou. Deletion techniques for the nd-tree in non-ordered discrete data spaces. *Proc. of The 18th International ConferenceSoftware Engineering and Data Engineering*, pages 1–6, 2009.

[38] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three next generation sequencing platforms: comparison of ion torrent, pacific biosciences and illumina miseq sequencers. *BMC genomics*, 13(1):341, 2012.

[39] Wei-Chun Kao, Andrew H Chan, and Yun S Song. Echo: a reference-free short-read error correction algorithm. *Genome research*, 21(7):1181–1192, 2011.

[40] Robert C Edgar and Henrik Flyvbjerg. Error filtering, pair assembly and error correction for next-generation sequencing reads. *Bioinformatics*, page btv401, 2015.

[41] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. Bless: bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[42] Juliane C Dohm, Claudio Lottaz, Tatiana Borodina, and Heinz Himmelbauer. Substantial biases in ultra-short read data sets from high-throughput dna sequencing. *Nucleic acids research*, 36(16):e105–e105, 2008.

[43] Francis YL Chin, Henry CM Leung, Wei-Lin Li, and Siu-Ming Yiu. Finding optimal threshold for correction error reads in dna assembling. *BMC bioinformatics*, 10(1):S15, 2009.