2011

# A FAST IMPLEMENTATION FOR CORRECTING ERRORS IN HIGH THROUGHPUT SEQUENCING DATA

Lankesh K. Shivanna

# A FAST IMPLEMENTATION FOR CORRECTING ERRORS IN HIGH THROUGHPUT SEQUENCING DATA

(Spine title: HiTEC2)

(Thesis format: Monograph)

by

Lankesh Shivanna

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO

School of Graduate and Postdoctoral Studies

## CERTIFICATE OF EXAMINATION

| | |
|---|---|
| Supervisor: | Examiners: |
| .................... | .................... |
| Dr. Lucian Ilie | Dr. Jagath Samarabandu |
| | .................... |
| | Dr. Nazim Madhavji |
| | .................... |
| | Dr. Mahmoud El-Sakka |

The thesis by

**Lankesh Kopalu Shivanna**

entitled:

**A FAST IMPLEMENTATION FOR CORRECTING ERRORS IN HIGH THROUGHPUT SEQUENCING DATA**

is accepted in partial fulfillment of the requirements for the degree of Master of Science

| | |
|---|---|
| .............. | ............................ |
| Date | Chair of the Thesis Examination Board |

# Abstract

The impact of the next generation DNA sequencing technologies (NGS) produced a revolution in biological research. New computational tools are needed to deal with the huge amounts of data they output. Significantly shorter length of the reads and higher per-base error rate compared with Sanger technology make things more difficult and still critical problems, such as genome assembly, are not satisfactorily solved. Significant efforts have been spent recently on software programs aimed at increasing the quality of the NGS data by correcting errors. The most accurate program to date is HiTEC and our contribution is providing a completely new implementation, HiTEC2. The new program is many times faster and uses much less space, while correcting more errors in the same number of iterations. We have eliminated the need of the suffix array data structure and the need of installing complicating statistical libraries as well, thus making HiTEC2 not only more efficient but also friendlier.

**Keywords:** HiTEC, error correction, DNA sequencing

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Deoxyribonucleic Acid (DNA) is the fundamental block and the blueprint of biological life from the inception to its growth and till death. All the cells with nucleus in human being's bodies have the same copy of DNA. All the functions performed by a cell are coded by DNA. It is the genetic material found not only in human beings but in all the living organisms. The discovery of DNA has revolutionized the science and medicine perhaps more than any another discovery. What makes DNA so important? There are numerous answers for the question. DNA transfers hereditary information from generation to generation, controls protein synthesis and determines the structure of the cell, meaning the cell should be a nerve cell, blood cells etc. DNA holds all this information using a sequence of four nucleotides to form a chain of nucleotides. The bases include two purines (Adenine and Guanine) and two pyrimidines (Cytosine and Thymine) commonly referred as A, G, C and T. Human DNA has 3 billion bases and 99% of it is common with all the other human beings. To unlock the information present in the DNA, analysis of the sequence of bases in DNA is essential. The technology which is used in determining the sequence of bases in DNA is called *DNA sequencing*.

The applications of DNA sequencing are many and the significance of these applications is huge. Genealogy, the study of ancestry is one of the prominent application of DNA sequencing. We resemble our parents because we have inherited DNA from them. The inheritance is due to the fact that our parent's DNA duplicates itself at cell division (in embryo) and passes on all of its properties to its daughter cells. This is how the information is passed on from generation to generation. Also genealogy helps in finding out the biological parents and other relatives by comparing the DNA. DNA sequencing helps in detecting hereditary diseases and finding a cure for those diseases. In forensic science, DNA sequencing has resulted in breakthroughs in many criminal cases. By comparing the DNA samples found on the crime scene with those extracted from the suspects, many criminal cases have been solved. The applications of DNA sequencing extend to agriculture as well. The results of DNA sequencing have been successfully used to improve the food products and crops by genetically modifying them and making the crops more resistant to diseases. Similar technology can be used in animal farming industry to produce improved breed of stronger build of animals.

The first DNA sequences were obtained in early 1970's based on two dimensional chromatography by academic researchers using very laborious methods. Now DNA sequencing has become easier and orders of magnitude faster. One of the earlier sequencing methods was RNA sequencing of Bacteriophage MS2, by Walter Fiers and his coworkers at University of Ghent, Belgium in 1972. At the same time scientists such as Frederick Sanger, Walter Gilbert and Allan Maxam were trying laborious methods to sequence DNA. Gilbert and Maxam were successful in sequencing 24 base pairs using a method named wandering-spot analysis. After the development of chain termination by Sanger and his coworkers in 1977, the DNA sequencing became easy and reliable.

After the introduction of Sanger's method of DNA sequencing, development of high throughput sequencing protocols and computational analysis methods have made sequencing a routine procedure. In 2007, a non Sanger ultra high throughput second generation sequencing technology became commercially popular. Third generation of sequencing technology named single molecular sequencing (SMA) were available in 2008 and it could overcome the limitation of first and second generation sequencing techniques. The first generation technology DNA sequencing methods including Sanger chain termination method relied on sequencing DNA based on the fragment length which was accurate but still there were many errors. For example, the first 15-40 bases had poor quality and the quality deteriorates again after 700-900 bases. Another major problem with the first generation techniques is that it consumed lots of space; it needs a place to run the reaction, capillary tube or gel to determine the DNA fragments length. Due to this reason only few hundreds of these reaction can be ran in parallel. Imagine sequencing human DNA with 3 billion base pairs. Sequencing human DNA using first generation techniques with 500 base pairs length of DNA fragments would take very long time.

Second generation sequencing techniques, like Illumina, overcame the restrictions of Sanger method by finding a way to sequence DNA without moving the DNA around. A bit of DNA which has to be sequenced is placed as a little dot, called cluster and do the sequencing in that cluster itself. Due to this behaviour millions of clusters can be fed into the machine at once. In Illumina, calling the first base takes long because it has to make sure the other DNA fragments are not gone out of sync. Illumina can produce a maximum of 100 base pairs length DNA fragments compared to 500-100 base pairs of Sanger method. 454 sequencing is another second generation sequencing method which uses nucleotides which produces a flash when a polymerase is added. This method gives out reads of length almost equal to that of Sanger method. However it produces much fewer reads. The significant problem common to all these techniques is higher error rate than Sanger method. The main advantages of the new methods are much higher speed and lower costs. Sequencing the genomes of many species became possible. Even after making the DNA sequence cheaper, the second generation sequencing still cost around 10 to 20 thousand to sequence a human genome. The low length reads is a huge problem because the machine takes longer time to figure out from which part of the DNA the read came from. Some of the third generation sequencers Pacific Biosciences, Oxford Nanopore and Life Sciences Qdot technology are trying to sequence single molecules of DNA in real-time.

The DNA sequencing machines, no matter which generation, produce huge amounts of data as output. The output data size has been exponentially increasing with the new generation DNA sequencing methods. There has been significant development in the area of softwares, which process these data. The two most widely studied problems are gene assembly and gene mapping. Gene assembly softwares try to construct the original DNA sample using the output reads of the sequencing machines. Gene mapping softwares try to map the particular read to a part of DNA, to know which part of the DNA it resembles. To be successful in both gene assembly and gene mapping, one of the important aspect is that reads have fewer errors. Some of the softwares have attempted to correct these erroneous reads or discard them. In this thesis we will focus on the software which attempt to correct these errors.

The simplest way to handle error in reads is to discard all the reads with an error and use only reads without errors for further processing. Assembly tools such as the spectral alignment based preprocessing step in Euler assembly program [CP09] and pre-filter reads of SHARCGS [Doh07] use this method. Discarding the erroneous reads may seem uneasy solution to the error correction problem but, following this approach leads to fewer reads to use. In this thesis we focus on correcting these errors and provide error free reads for any further processing. The success of error correction lies in having a good coverage of the specified DNA sample. The main idea is, if you sequence a DNA sample multiple times the correct value prevails. This idea has been used in multiple variations to provide better and better error correction softwares. SHREC [SSP+09] uses suffix trees with weights in order to detect and correct the errors. CUDA [SSLW10] provides a successful hardware architecture using CUDA-enabled graphics in order to correct the sequencing errors in parallel. Reptile [YDA10] is another error correction method using k-mers mentioned in the spectral alignment problems to correct the sequencing errors. HiTEC [IFI10] took a step forward and used the statistical approach in figuring out the best parameters to suite the input data set and use these parameters with suffix array and LCP(Longest Common Prefix) to maximize the error correction. This helped HiTEC to be the most successful error correction software available to date.

In Chapter 2, some of the popular sequencing techniques are explained in breif. Chapter 3 will explain in detail about the error correction algorithms used in recent times. HiTEC2 software is built using C++ programming language. In HiTEC2, we took all the positive concepts and built a software which corrects the same amount of errors in less than one third the time and using one fourth the space compared to HiTEC. HiTEC2 uses 2-bit binary encoding and binary operation in achieving the above results. This is explained in Chapter 4 in detail. We have tested HiTEC2 on many simulated and real data sets. The comparison is made only with HiTEC, because we know that it the most accurate error correction software till date. During the period of testing, we have observed that the HiTEC2 is as robust as HiTEC as HiTEC2 inherits the statistical approach from HiTEC. This helped HiTEC2 to operate on data sets with wide range of read lengths and data size. Due to 2-bit encoding of the input data set, removal of suffix array approach and use of hash table approach has made HiTEC2 5 to 7 times faster[1]

[1]Table 5.7 and 5.8

and 2 to 3 times less space[2] consuming than any other error correction software. We have built a software which is faster and less space consuming without hampering the accuracy of error correction. Nevertheless, we are planning to improve the time and space of out software by providing a parallel implementation, the details of which are given in Chapter 6.

[2]Table 5.6

# Chapter 2

# DNA Sequencing

In 1843, Gregor Mendel, "father of genetics", performed experiments on thousands of pea plants [Hen01, Mey00], the results of which led to increased interest in the study of Genetics. Mendal used two pea plants, one with white flowers and one with purple flowers, cross fertilized two true breeding pea plants and observed the offsprings characteristics: they both had purple flowers. Then, he let the offspring self fertilize and he saw that some flowers were white and majority of them were purple. He continued his experiments and came to a conclusion that "*units of information*", one from each parent, existed in the offsprings and one trait was dominant (purple) because it covered the other trait (white), thus laying the foundation of genetics and heredity. It took almost a century after that to have a breakthrough in the field and the breakthrough was given by an army medical officer named Frederick Griffith [Jos85]. Who, by trying to find a vaccine against pneumonia, made an accidental breakthrough in the world of heredity. He pointed out that DNA was the molecule of inheritance which was proved by Oswald Avery some years later. Around 1940 Erwin Chargaff noticed a pattern in the amounts of the four bases, adenine, guanine, cytosine, and thymine. In his experiments he took samples of DNA of different cells and found out that the amount of adenine was almost equal to the amount of thymine and that the amount of guanine was almost equal to the amount of cytosine. The rule A=T and G=C was named as Chargaff's Rule [EE48]. Scientists all around the world were indulged in finding more and more information about this DNA, little was known about its structure. Two scientists Maurice Wilkins and Rosalind Franklin were the first to obtain very good x-ray diffraction of DNA fibers by crystallizing the DNA [FG48, Eve07]. The

DNA in the X-ray was shaped like an X. Using these X-ray photos, James Watson and Francis Crick[WC53] in 1951 came up with the double helix, the structure that is almost always associated with DNA.

## 2.1 Biological molecules

DNA is a nucleic acid that contains the genetic instructions used in the development and functioning of all the known living organisms. The nucleic acid's structure or the sequence is the compositions of the atoms and the chemical bonding between them. DNA is an unbranched polymer, the sequence of the nucleotides in the molecules will represent the structure of the nucleic acid. The sequence of the nucleotides has the capacity to represent information. There are four types of nucleotide bases adenine, guanine, cytosine, and thymine. The technology or the methods used to determine the sequence of these nucleotides in a DNA fragment is called *DNA sequencing*. No method can sequence an entire DNA molecule. They all produce short DNA sequence, called *reads*.

### 2.1.1 Applications of DNA sequencing

**Medicine:** Identify, diagnose and potentially develop treatments for genetic diseases and cancers.

**Forensics:** To match the physical evidence from the crime scene to a particular individual, determine the paternity of an individual etc.

**Agriculture:** Specific genes of the bacteria have been used in some food plants to increase their resistance against insects and as a result, the productivity and the nutritional value of the plants is increased.

## 2.2 Sanger method

The sequencing of DNA molecules began in the 1970s with two group of scientists, one led by Americans Maxam and Gilbert, used a chemical cleavage protocol for DNA sequencing. The

English, led by Sanger, designed a procedure similar to the natural DNA replication [Rus02]. This method was later on famously called as *Sanger* or *Chain Termination* method [San80]. Most DNA sequencing that occured in medical and research laboratories in the past several decades was performed using sequencers employing variations of the Sanger method.

Sanger's method is based on the use of dideoxynucleotides in addition to the normal nucleotides found in DNA. Dideoxynucleotides are the same as nucleotides with one exception: they contain a hydrogen group on the 3′ carbon instead of a hydroxyl group. This extra hydrogen group nucleotide when integrated into a sequence, prevents the addition of further nucleotides. This occurs because a phosphodiester bond cannot form between the dideoxynucleotide and the next incoming nucleotide due to the presence of the hydrogen group in the 3′ end and thus the DNA chain is terminated. So the name *Chain Termination method.*

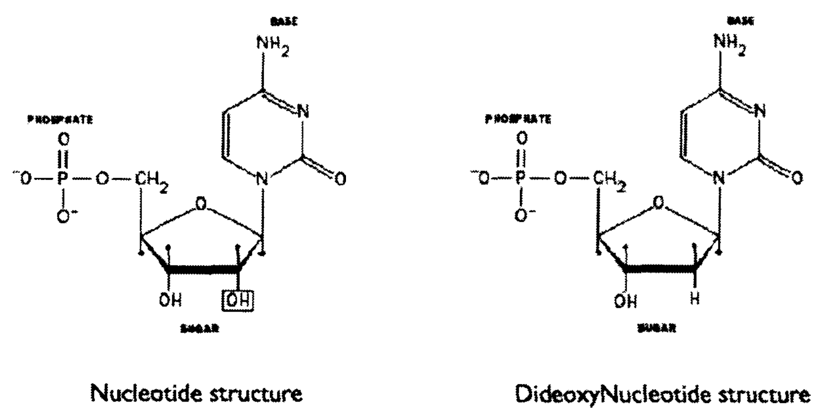


Figure 2.1: Structure of a Nucleotide and Dideoxynucleotide used for chain termination. The figure clearly shows the hydroxyl group required for chain termination

First the DNA is extracted and isolated from the host. Then the DNA is subjected to heat so that one strand can be stripped off and a fluorescent label is attached to one end of the single stranded DNA. The output of this method will have a single stranded DNA. The method requires four test tubes each filled with all four nucleotides, DNA ligase and DNA polymerase. Each one of the G, A, T, C dideoxynucliotides are filled individually in the four different test tubes.

As the DNA is synthesized, nucleotides are added on to the growing chain by the DNA polymerase. However, on occasion a dideoxynucleotide is incorporated into the chain in place of a normal nucleotide. This results in a chain-terminating event. For example if we looked at only the "T" tube, we might find a mixture of the following products:

| | |
|---|---|
| Genome | ATCGGGGATGCTGAGCTAGGAATG |
| Chain 1 | AT |
| Chain 2 | ATCGGGGAT |
| Chain 3 | ATCGGGGATGCT |
| Chain 4 | ATCGGGGATGCTGAGCT |
| Chain 5 | ATCGGGGATGCTGAGCTAGGAAT |

Table 2.1: Example of the contents in the T-tube of Sanger method

The best part about this method is that the chain starts at a specific location and always ends up with the same base. The natural replication will enable this method to have the similar end bases at different positions. This makes each DNA sequence to be of different length. The contents of each of the four tubes are run in separate lanes on a polyacrylmide gel in order to separate the different sized bands from one another. After the contents have been run across the gel, the gel is then exposed to some kind of electric field. Due to the nature of nucleotides being negatively charged, the electric field will push the chains of DNA towards the positive end. The chains being variable in length, constitutes different molecular weights. Each chain of DNA will move with specific speed. That is, lower length chain will move faster and reach the positive end earlier than the other longer chain of DNA. If all of the reactions from the four tubes are combined on one gel, the actual DNA sequence in the 5′ to 3′ direction can be determined by reading the banding pattern from the bottom of the gel up. The DNA sequence read will be the reverse complement of the original DNA. Figure 2.2 shows the DNA chains in the gel properly aligned with respect to their lengths.

### 2.2.1 Shotgun sequencing

Shotgun sequencing is a product of chain termination method [Sta79, And81]. This method uses the chain termination method along with gene assembly technique to get a sequence whose length is more than the one produced by the chain termination method. In this method the DNA is broken into small pieces( 2, 10 and 50 base pairs). The broken DNA pieces are then sequenced using chain termination method to obtain a higher length DNA read. Numerous

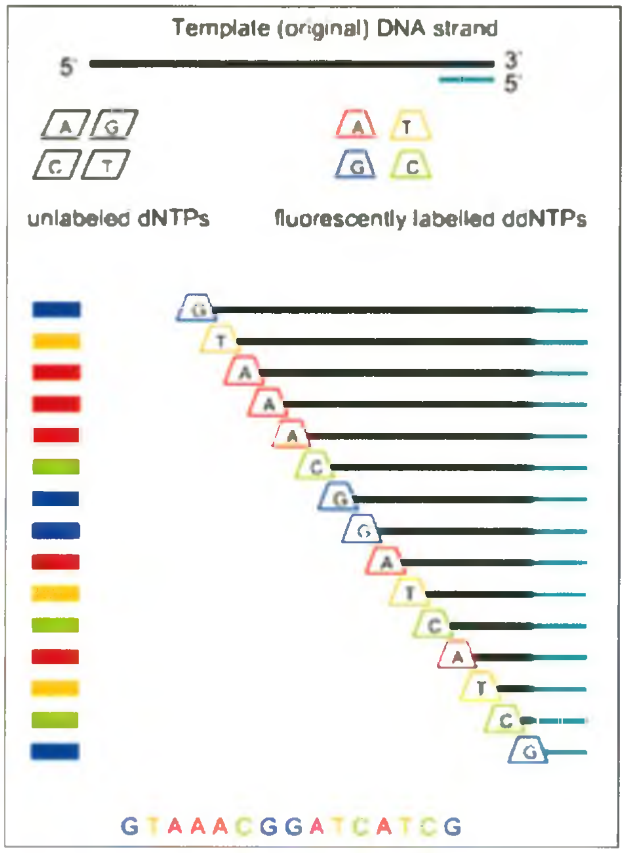

Template (original) DNA strand
5'
3'
5'
A G C T
unlabeled dNTPs
A T G C
fluorescently labelled ddNTPs
G T A A A C G G A T C A T C G

Figure 2.2: Sanger method

overlapping reads are obtained by more cycles of fragmentation and sequencing. The computer algorithms then use the overlapping ends to assemble the reads and build a continuous sequence.The table 2.2 shows a simple example.

The example shown in Table 2.2 is an ideal case and very simple. The practical sequencing is much complex. Because the assembly algorithms use a lot of information which is ambiguous due to sequencing errors. Then there are repetitive sequences, which are the worst enemy of the assembly algorithms. As the sequences considered are small in length, there is a chance that they appear in a different part of the genome.

## 2.3 High throughput sequencing methods

Sanger's shotgun sequencing and Maxam-Gilbert sequencing were first few methods available for DNA sequencing. These methods and Dye-Terminator sequencing method with some other

| Strand | Sequence |
|---|---|
| Original sequence | AGCTTAGGCTCGTTTAGGCAAAT |
| Sequence 1 | AGCTTAGGCTCGTTTAGG . . . . . |
| Sequence 2 | . . . . . . . . . . . . . . . . . . . . . . CAAAT |
| Sequence 3 | AGCTTAGG. . . . . . . . . . . . . . . . . . |
| Sequence 4 | .. . . . . . . CTCGTTTAGGCAAAT |
| Reconstruction | AGCTTAGGCTCGTTTAGGCAAAT |

Table 2.2: Example showing the overlapping of reads in shotgun sequencing

initial techniques were named as first generation DNA sequencing methods. The high demand of low cost sequencing techniques which produces thousands and millions of sequences at once gave birth to second generation DNA sequencing techniques. The second generation DNA sequencing methods used technologies that parallelize the sequencing process to produce high output with low cost. The latter part of the Section 1.3 will explore some of the second generation DNA sequencing techniques.

### 2.3.1 Maxam–Gilbert sequencing

Maxam and Gilbert came up in 1976 with an idea for sequencing DNA based on chemical modification of DNA and subsequent cleavage at specific bases. This method was published two years before the initial Sanger and Coulson plus-minus method of sequencing [Jos85, EE48]. Due to the ready use of the purified DNA this method became famous right away. However with the improvement of chain termination made by Sanger to his initial method, the Maxam Gilbert method lost its charm. The downfall of this method is caused by the technical complexity, extensive use of hazardous chemicals, and difficulties with scaling up [FG48]. The method for sequencing is as follows. First prepare a homogeneous single strand of DNA and then add a radioactive label at one 5' end of the DNA. In each of the four reactions, G, A+G, C, C+T, the DNA breaks at small proportion of one or two or some small number of nucleotides. The concentration of the chemicals are modified in such a way as to modify an average of one modification per DNA molecule. The DNA will be cut on one end using the 5′ radioactive label information. Then the four different DNA fragments are subjected to an electrophoresis reactions side by side in denaturing acrylamide gels for size separation, similar to that of the chain termination method. Then, to visualize the format, the gel is exposed to X-ray film for autoradiography. This process yields a series of dark bands each corresponding to a radio labeled DNA fragment, from which the sequence may be inferred.

### 2.3.2 Dye-Terminator sequencing

Dye-Terminator sequencing is very similar to that of the Sanger chain termination method. Instead of having four different reactions, there will be only one reaction but the four dideoxynucleotide chain terminators will be labelled with a unique fluorescent dye, each of which will emit light at different wavelengths. This method having greater expediency and speed has been used extensively in automated sequencing. The only thing we need to take care of is the dye labelling fluorescence band should be distinct and far apart to make a confident base call.

The next generation DNA sequencing platforms produce sequencing reads with increased depth of coverage[1] but reduced read length and lower per-base accuracy than data than Sanger method.

### 2.3.3 PyroSequencing

PyroSequencing is based on the "*sequencing by synthesis*" principle and one of the first of many next generation sequencing techniques. The technique was developed by Pal Nyren and Mostafa Ronaghi at the Royal Institute of Technology in Stockholm in 1996 [MMP98, MKP+96, Pal07]. This method involves taking a single stranded DNA and synthesizing its complementary strand enzymatically [Mar08]. The experiment detects the activity of DNA polymerase with chemiluminescent enzyme, in building a complementary strand base by base. The template of DNA will be fixed and the solution A, C, G, T nucleotides will be added and removed sequentially followed by analyzing the light produced when an nucleotide solution complements the first unpaired base of template. This allows to determine the sequence of nucleotides. The drawback of this method is that it produces the DNA sequencing of lengths 300-500 compared to 800-1000 of Sanger method and also increase in sequencing errors.

[1]Coverage is defined as the average number of reads representing a given nucleotide in the original genome sequence. It can be calculated from the length of the original genome (G), the number of reads(N), and the average read length(L) as $c = N\frac{L}{|G|}$

### 2.3.4 Illumina Method

The Illumina method of sequencing debuted in 2006 and has been used to sequence the first African, Asian and cancer patient genomes. Fragments are bound to a slide and grown in clusters to provide a stronger fluorescent signal. The process starts with a double stranded DNA. The DNA sample is broken into smaller pieces and each of two different types of *adaptor* (short DNA sequences) are attached to the DNA fragments on both ends. Then a slide is prepared, where a lawn of primers are attached to the slide and the DNA fragments are put onto this slide. The amplification starts once the DNA fragments are put onto the slide and the DNA bends over and finds a complementary primer on the surface. From the primer on the surface a complementary strand is obtained as result of *amplification* process. The strands are then split apart, and the replication process is repeated to create more copies of the replicated complementary DNA strand. This process forms a dense clusters of DNA strands which is termed as *channels*. The channel contains both straight and complementary strands; to make the sequencing much effective one type of strands are removed. To the slides, A, C, G, T nucleotides and DNA polymerase are added. Due to the reaction, the nucleotides go and stick to the single stranded DNA. As the bases are incorporated, a laser is used to activate fluorescence and the color is read. A computing machine monitors each cluster and notes each color as a new base is added from which it works out a sequence from many clusters [Mar08]. One of the major drawbacks of the Illumina method is that reads are shorter than other sequencing methods. It is less suitable for sequencing an organism for the first time. One of the major advantage of this method is that it is faster and cheaper. In comparison with Sanger method which requires a year to read one gigabase at $0.1 per 1000 bases, the Illumina take just over half a day to read one gigabase for $0.001 per 1000 bases. Figure 2.3 shows the amplification process and creation of channels in Illumina technique.

### 2.3.5 SOLiD sequencing

SOLiD (Sequencing by Oligonucleotide Ligation and Detection) is developed by Life Technologies and is a next-generation sequencing technology [FPM+08]. This has been commercially available since 2008. SOLiD sequences the DNA by ligation process. The single

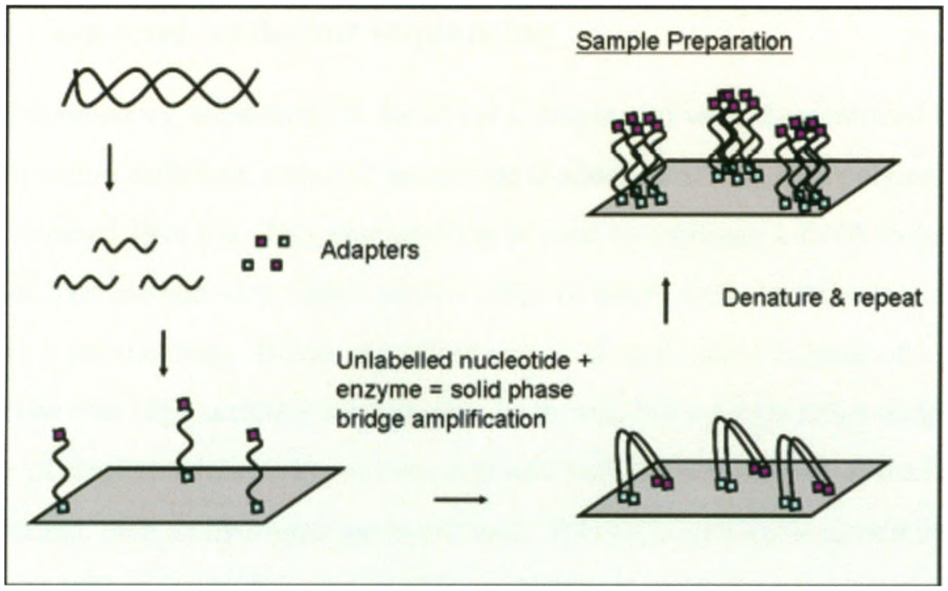

Sample Preparation
Adapters
Denature & repeat
Unlabelled nucleotide + enzyme = solid phase bridge amplification

Figure 2.3: Illumina sequencing method

stranded DNA is shredded into many small DNA pieces of specific size and adapters are ligated at both ends of the DNA pieces to construct a fragment library. So the library consists of millions of DNA molecules which represent the entire target sequence. Each molecule is then clonal amplified into beads in an emulsion PCR reaction. The sample is then enriched on the beads to replicate the DNA piece. The beads are then covalently attached to glass slides and then put in contact with the large pool of dibase probes which are fluorescent labelled with 4 dyes and DNA polymerase. Each of the 4 dyes represent 4 of the 16 possible dinucleotide sequences. The complementary probe hybridizes to the template sequences and is ligated. After the measurement of the fluorescence the die is stripped of leaving the 5′ end of the DNA for further reactions. The step can be repeated to as many cycles as needed for appropriate read lengths. After these cycles the synthesized strand is removed and new primer is hybridized with an offset of one base and the ligation cycles are repeated. This will give a much needed dual measurement of each base. The accuracy is increased with the number of cycles of above operations. The unique 2 base encoding process is designed to increase the accuracy and reduce measurement errors.

### 2.3.6 Ion semiconductor sequencing

Ion semiconductor sequencing is based on a simple and well characterized biochemical process. In nature each time a base of nucleotide is added to a DNA by a polymerase an hydrogen ion is released [Rus11]. This phenomenon is used to sequence a DNA in ion semiconductor method. The method uses a high density array of micro wells to sequence multiple pieces of DNA in a parallel way. Below the micro arrays of wells there is layer of ion sensitive layer and below that high sensitive ion sensors. Each well has a single DNA template and a micro sample of single nucleotide is put into each well with the polymerase. If the base ligates to the DNA strand, then an hydrogen ion is released. This hydrogen ion is sensed by the layer below the micro well and voltage is recorded to see if there is single base are consecutive bases of the induced nucleotide. If the hydrogen ion is not released then the voltage will be null and base call is not made. One of the major advantages of this method is the parallelism of sequencing. But in practice this method does not produce higher read lengths.

# Chapter 3

# Error Correction Algorithms

The continuous improvement in high throughput sequencing technique has open doors for many new applications. But sequencing errors still remain a major problem. There are many techniques proposed to solve the error correction problem for the uniform and non uniform datasets. But the chance of improvement is still there. In this section we will discuss some of the basic concepts of the error correction problem and some of the techniques used to solve it.

## 3.1 Basic Idea of Error Correction

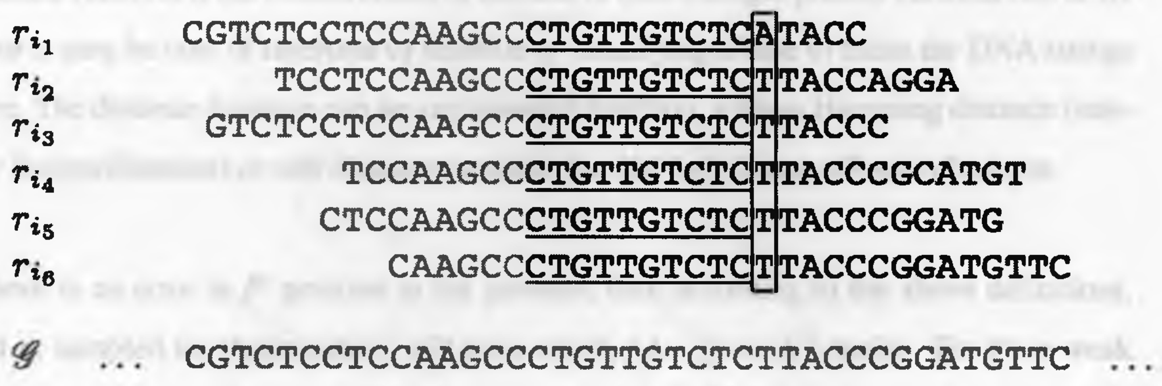
To define error in DNA sequencing, lets take an example. Consider a genome $G$ which is a string of length $|G| = L$[1] over the alphabet $\sum$ = {A,G,C,T}. The probability of occurrence of each letter is 0.25. Also assume that there are $n$ reads of length $l$ produced from the same genome with the per base error rate being $p$. Now if we have had to add a base in front of a random read $r_i$, there are four possibilities. We call the position to be erroneous if that read with the concatenated base as a string is not a substring of the genome $G$. The basic idea of correcting errors in DNA sequencing is as follows: If a genome is replicated and sequenced multiple times the correct base value prevails.

To clearly present the concepts with examples lets put some notations in place. Consider $n$ reads $r_1, r_2, .....r_n$, each of read length $l$. $R$ is a set of reads $\{r_1, \bar{r_1}, r_2, \bar{r_2}......r_n, \bar{r_n}\}$. $\bar{r}$ is the reverse

[1]Length is the number of bases in the string.

complement[2] of read r. The $i^{th}$ letter of $G$ is denoted $G[i]$.

Assume a read $r_i$ is sampled from position $j$ of the genome and there is an error at position $k$. Also assume that the $w$ positions before the position $k$ are error free. That is $r_i = xuay$, where $x, u, y \in \sum^{\star}$, $|x| = k - w - 1$, $|u| = w$, and $a \in \sum$. The set of all strings over $\sum$ is denoted by $\sum^{\star}$. This means that the letter $a$ should be letter $b = G[j + k - 1]$ which appears in the genome. As we know, the genome is sequenced multiple times for the same base. This means that the correct string $ub$ appears in many more reads as a substring than $ua$. The proof of erroneous base $a$ in string $ua$ is derived from the fact that $u$ is followed more often by $b$ than by $a$. The string $u$ is said to be witnessing the error $a$ and implying that it should be changed to $b$. For $a \in \sum$, the support of $u$ for $a$, supp($u$, $a$), is the number of occurrences of the string $ua$ in $R$. In Figure 3.1 from [IFI10] the support and the witness concept is illustrated in detail.



```
r_i1        CGTCTCCTCCAAGCCCTGTTGTCTCATACC
r_i2            TCCTCCAAGCCCTGTTGTCTCTTACCAGGA
r_i3         GTCTCCTCCAAGCCCTGTTGTCTCTTACCC
r_i4                  TCCAAGCCCTGTTGTCTCTTACCCGCATGT
r_i5                 CTCCAAGCCCTGTTGTCTCTTACCCGGATG
r_i6                    CAAGCCCTGTTGTCTCTTACCCGGATGTTC

𝒢   ...  CGTCTCCTCCAAGCCCTGTTGTCTCTTACCCGGATGTTC  ...
```

Figure 3.1: An example of an error covered by six reads; the genome region where the reads came from is shown at the bottom. The letter (inside the frame) following the witness $u$ = *CTGTTGTCTC* (underlined) should be T and not A. The support values are supp($u$, T) = 5 and supp($u$, A) = 1.

## 3.2 Euler SR alignment

Euler SR alignment is used before the assembly process of the shotgun technique for error correction to get the most out of assembly algorithms to produce contigs of higher length. The spectral alignment problem [CP09] is very simple in concept and can be implemented in many

[2]Reverse complement of DNA sequence is the complement of DNA sequence when read from end to the start of the sequence

different ways. Below are few of the definitions used further to understand the concept.

*Definition* 1: An $k$-tuple (i.e., a DNA string of length $k$) is called *solid* with respect to $R$ and $m$, if it is a substring of at least $m$ reads in R and *weak* otherwise.

*Definition* 2: The *spectrum* of R with respect to $m$ and $k$, denoted as $T_{m.l}(R)$, is the set of all solid $k$-tuples with respect to R and $m$.

The spectral alignment problem [CP09] is stated as: Given a DNA string $s$ and spectrum $T_{m.l}(R)$, find a string $s^{\star}$ in the set of all $T_{m.l}(R)$-strings that minimizes the distance function $d(s, s^{\star})$.
The distance function is the measurement of number of base changes present between two DNA strings or it may be cost of insertion or deletion or modifying a base to make the DNA strings the same. The distance function can be any standard function, such as Hamming distance (suitable for Soleza/Illumina) or edit distance (suitable for 454 Life Science/Roche) function.

If there is an error at $j^{th}$ position in the genome, then according to the above definitions, the read $r_i$, sampled for that position, will have $min(k, j, l - j)$ weak $k$-tuples. For these weak $k$-tuples a nearest strong $k$-tuple will be found and replaced. If there are errors in read $r_i$, it is not always $min(k, j, l - j)$ tuples which are going to be weak. This is because a read $r_i$ can have more than one error. To add to the list there are weak tuples to be represented as strong and strong tuples being represented as weak, due to high error rates in the sequencing technique. The first issue of read $r_i$ having more error than one is tackled by increasing the distance. But doing so will see a huge change in the time complexity. The other issue can be minimized by selecting carefully the values of the multiplying factor $m$ and length of the tuple $k$.

## 3.3 SHREC

The main idea of the SHREC (SHort Read Error Correction) method [SSP+09] involves building a generalized suffix tree $ST(R_s)$ of all the reads and their reverse complements.

**Suffix Tree**

The Suffix tree is a data structure where a string is represented in a way to perform operations on strings very fast. In a suffix tree each edge is given a string value. By traversing from the root node to any leaf and collecting all the string on the way, a unique suffix from the original string is obtained. This is one of the fundamental concepts of suffix tree. Also each edge should have a non empty string associated to it and all internal nodes should have at least two children (except the root). The concept was first introduced by Weiner [Wei73] in 1973 as position tree. Then the concept was significantly simplified by McCreignt in 1976. Ukkonen after that gave a beautiful online-construction of suffix trees, now known as Ukkonen's algorithm [Ukk95]. Figure 3.2 shows the suffix tree for the string BANANA$.

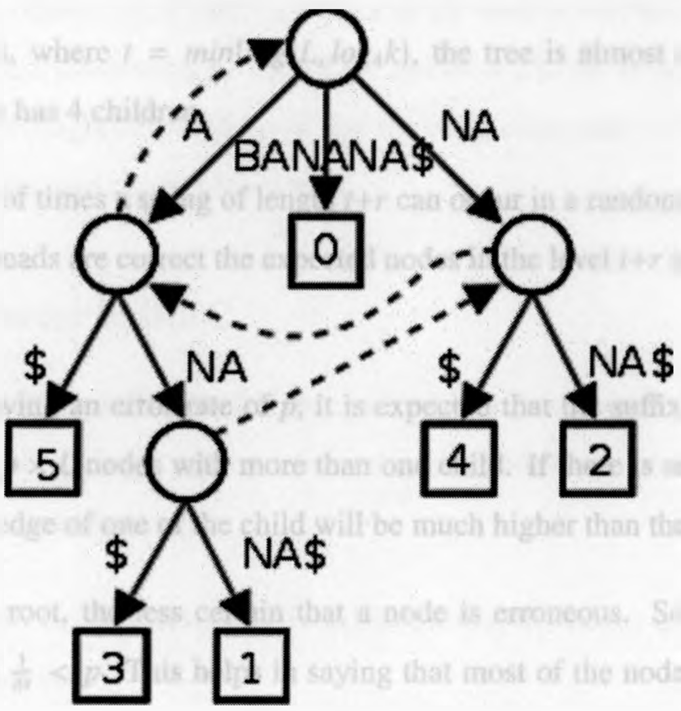


Figure 3.2: Example of Suffix Tree

### 3.3.1 Suffix Tree in SHREC:

The suffix tree in SHREC is built for all the reads and their reverse complements with each read and its complement terminated with a unique value from 1,2,... 2k.

$$R_{st} = r_1\ 1\ \bar{r_1}\ 2\ r_2\ 3\ \bar{r_2}\ 4\ ....r_n\ 2k\ \bar{r_n}$$

Each edge in the $ST(R_{st})$ is uniquely labelled with a single string to ensure the path from the root to each suffix is unique. The string formed by concatenating the characters found on the path from the root to node $v$ is called the *path-label* of node $v$. The level of node $v$ is the string length of the path-label of node $v$. Level zero is the top level of the $R_{st}$ and level $n$ is the bottom level. The suffix tree built by the SHREC algorithm has one more important characteristic to it, the weight of the edge. The weight of the edge is defined as the number of leaves in the subtree below the edge. In turn the weight of the node is $v$ is the number of times the path label $v$ appears as substring in the reads from $R_{st}$.

The generalized suffix tree used in SHREC has some properties listed below

- In the top $t$ levels, where $t = min\{log_4L, log_4k\}$, the tree is almost complete. That is almost every node has 4 children.

- The total number of times a string of length $t$+$r$ can occur in a random string of length $L$ is $\frac{L}{(t+r)^4}$. So if the reads are correct the expected nodes in the level $t$+$r$ should not be more than $\frac{1}{r^4}$.

- With the reads having an error rate of $p$, it is expected that the suffix tree at each level will have at least $p \times L$ nodes with more than one child. If there is sufficient coverage, the weight of the edge of one of the child will be much higher than the other one.

- The closer to the root, the less certain that a node is erroneous. So a parameter $q$ is defined, such that $\frac{1}{4^q} < p$. This helps in saying that most of the nodes below level $t$+$q$ have errors.

- Nodes below level $s$ do not have sufficient weight to distinguish between correct and erroneous nodes. (See Figure 3.3 from [SSP+09].)

SHREC differentiates between the erroneous nodes and correct nodes in the generalized suffix tree. Looking at the properties of the generalized suffix tree constructed in SHREC, the error correction algorithm is very simple. The coverage as we know should be sufficient, it is one of the main deciding factor in differentiating between the erroneous nodes and correct

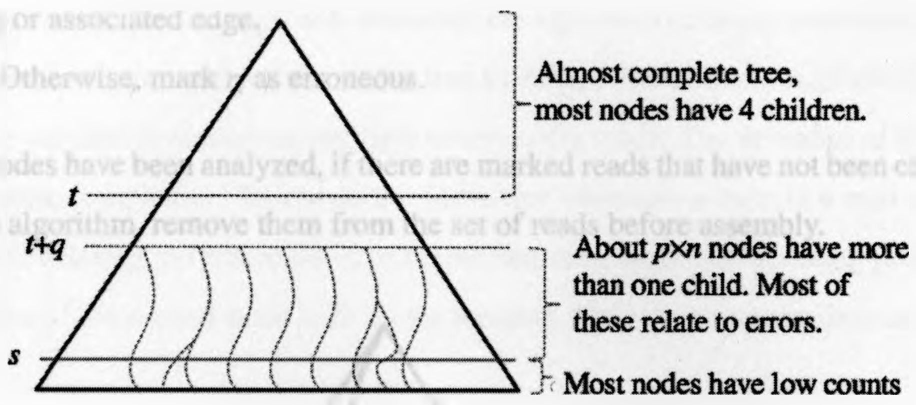

Almost complete tree, most nodes have 4 children.
t
t+q
About p×n nodes have more than one child. Most of these relate to errors.
s
Most nodes have low counts

Figure 3.3: Structure of suffix tree ST( $R_{st}$)

nodes. As shown in Figure 3.4, if there is a error in the node it will have more than one child and the weight of edge connecting to one of the children will be considerably high than the others. This is how the algorithm will know that the child connected with the higher weighted edge is the correct node and the rest are wrong.

### 3.3.2 Algorithm for SHREC

Construct the generalized suffix tree for the reads and the reverse complements with all the weights of edges, nodes and map appropriate read information to the nodes. Traverse the suffix tree as mentioned in the below steps.

1: Perform a depth-first traversal of ST($R_{st}$) inspecting the nodes from level $s$ up to level $t+q$ for potential errors.

2: Identify all nodes $w$ with at least two children where one of the children $w$ has a smaller than expected weight.

3: For each identified node $w$ find the set of reads R($w$) belonging to the suffixes in the sub-tree below $w$.

4: For each read $r_i \in$ R($w$) examine if correction to a sibling of $w$ fits the suffix.

a. If so, calculate error-position in $r_i$ and correct the nucleotide to the edge label of siblings or associated edge.

b. Otherwise, mark $r_i$ as erroneous.

5: After all nodes have been analyzed, if there are marked reads that have not been corrected during the algorithm, remove them from the set of reads before assembly.

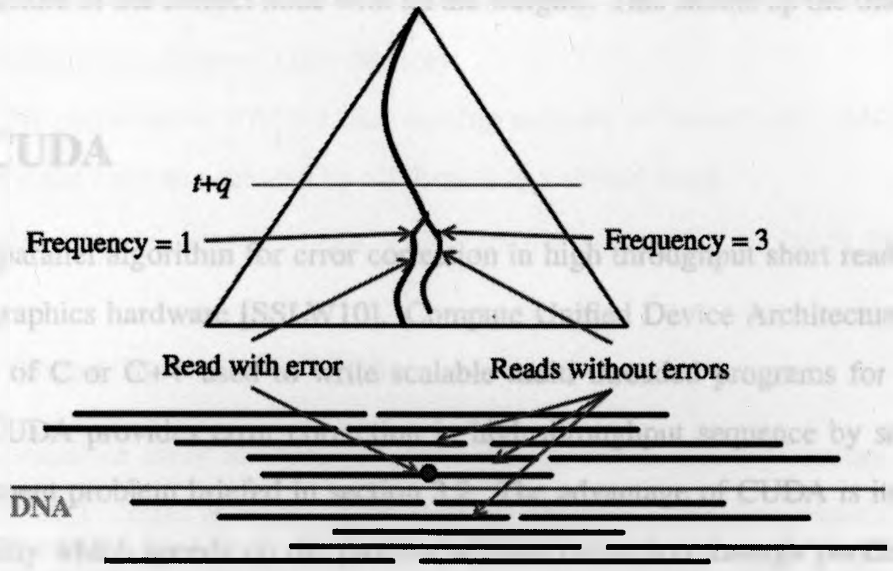


Figure 3.4: Typical error scenario in SHREC

There are three different approaches to run the above algorithms based on the requirement of the user.

**Identify - only approach:** The simple method to handle errors in the reads is to discard erroneous reads and keep only those reads which are correct. The SHREC can be run to detect the erroneous nodes in the suffix tree and delete all the reads which have that path label. This approach is not suited for practical applications because there will be less reads available which do not contain any errors.

**Static approach:** In this approach, whenever the algorithm finds an erroneous node it corrects the error and marks the read so that the same error is not corrected. Which means to say if more than one error occurs in a read, only one error is corrected and still the read is erroneous.

**Dynamic approach:** In this approach whenever the algorithm finds an erroneous node, it corrects the error and updated the suffix tree to reflect this correction. Following this approach can lead in correcting multiple errors in the reads. The downfall of this approach is the time complexity. To update the suffix tree whenever a base in a read is correcting involves deleting the whole sub tree below that error node and updating proper sub tree structure of the correct node with all the weights. This shoots up the time complexity.

## 3.4 CUDA

This is a parallel algorithm for error correction in high throughput short read data on CUDA enabled graphics hardware [SSLW10]. Compute Unified Device Architecture (CUDA) is an extension of C or C++ used to write scalable multi threaded programs for CUDA enabled GPUs. CUDA provides error correction in high throughput sequence by solving the spectral alignment problem briefed in section 3.2. The advantage of CUDA is its multi threaded functionality which speeds up the process of error correction through parallel programming approach.

To implement the multi threaded functionality certain architecture should be laid first. The CUDA programs contains certain sequential part called a *kernel.* The kernel is written in a scalar C-code. It lays out the function of the single thread and is invoked as a part of concurrently executing threads. The threads are hierarchically organized to form thread blocks and the set of thread blocks are grouped as *grids*. Each grid is uniquely associated with a ID which is the set of two values *(threadIdx,blockIdx).* Each thread has it own space and parameters to operate on. The size and the parameter list are predefined. When there are multiple threads operating concurrently, there is always a need of communication between the threads. Threads in the same block can communicate using the shared memory space called *per-block shared memory (PBSM).* When there is a need for communication between thread blocks, a different hierarchy of memory structure is needed. Here is the list of different memory spaces used in CUDA:

*Readable and writable global memory* is relatively large (typically around 1GB), but has high latency, low bandwidth, and is not cached.

*Readable and writable per-thread local memory* is of limited size (16 KB per thread) and is not cached. Access to local memory is as expensive as access to global memory.

*Read-only constant memory* is of limited size (totally 64 KB) and is cached. Reading from constant memory can be as fast as reading from a register.

*Read-only texture memory* is large and is cached. Texture memory can be read from kernels using texture fetching device functions. Reading from texture memory is generally faster than reading from global or local memory.

*Readable and writable PBSM* is fast on-chip memory of limited size (16KB per block). Shared memory can only be accessed by all threads in a thread block.

*Readable and writable per-thread registers* is the fastest memory but is very limited.

### 3.4.1 Hardware model of CUDA

The scalable processor array in Telsa architecture supports CUDA applications (see Figure 3.5 from [SSP+09]). Telsa is made up of array of *Streaming microprocessors* (SM). Each SM is made of typically 8 *Streaming processors* (SP). Each of these eight processors share a common per block share memory of 16KB. Typically 32 threads, called as a *wrap*, are executed concurrently in a single SM by single instruction multiple thread fashion. Due to the parallel implementation the performance is directly dependent on the data independency. That is, the more the data is independent, the less the thread needs to synchronize and therefore less time complexity. The GPUs with NVIDIA's Tesla unified computing architecture are well suited to execute the CUDA programs. Examples of CUDA- enabled GPUs include the Tesla 800/1000, GeForce 8/9/200 and Quadro FX 3000/4000/5000 series.

### 3.4.2 Bloom data structure:

In section 3.2 the spectral alignment problem is stated. One of the requirement for the spectral alignment is to maintain a list of solid tuples. When maintaining the solid tuple it is important to have a data structure to quickly access the solid tuple in the list. For this purpose the CUDA employs the probabilistic hashing scheme based on the space efficient bloom filter data structure [Blo70].

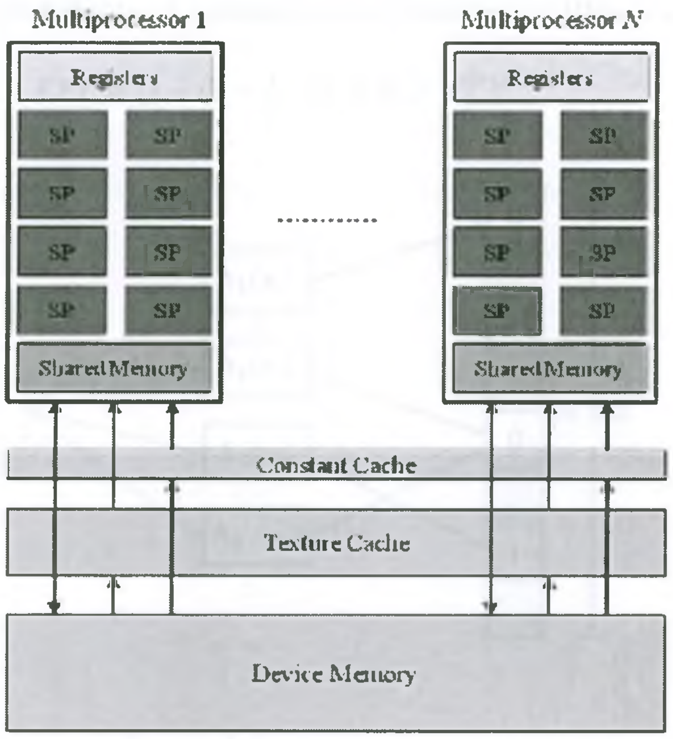


Figure 3.5: Hardware model Telsa for CUDA enabled GPU's

The Bloom data structure, conceived by Burton Howard Bloom in 1970, is a space efficient probabilistic data structure that is used to test whether an element is a member of a set. False positive are possible, but false negatives are not. To simplify the previous statement, the query of an *l* tuple being solid may will always result in yes. But sometime a query of an *l* tuple being weak may result in solid tuple. This particular drawback is tolerated because of the space efficiency of this data structure.

Bloom filter is a bit array of *m* bits, initially all set to 0. There should be *k* different hash function defined which will be the tuple length in case of SAP(Spectral Alignment Problem), each will map some elements of the one of the *m* array position with a uniform random distribution. To add an element into the filter, the tuple is fed to the *k* hash functions and a set of *k* array positions are obtained. Then all those *k* array positions are set to 1. Only strong tuples are added to the Bloom filter. To query an element from the filter, the tuple is again fed to *k* hash function to retrieve the *k* array position. If all the *k* array positions are 1 then the tuple is solid. The false positives arises from the fact that the *k* bits can be set to 1 from any number of

$n$ keys. The false positive probability of a bloom filter is provided by Bloom is

$$FPP \approx (1 - (1 - \tfrac{1}{m})^{kn})^k = (1 - e^{\frac{-kn}{n}})^k$$

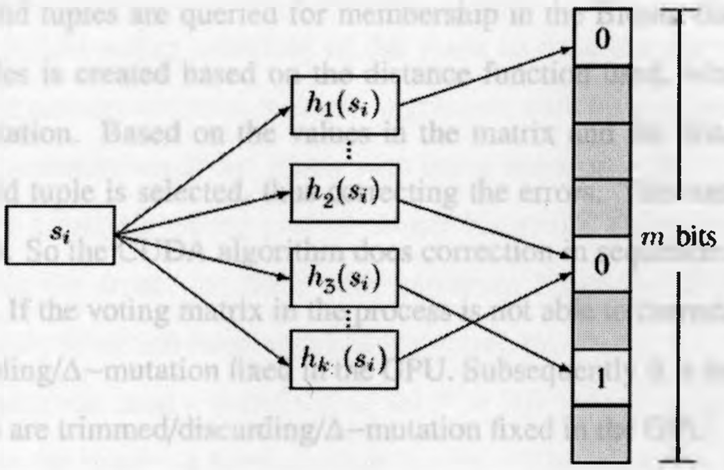


Figure 3.6: Bloom filter

The FPP formula tells that, in order to maintain a fixed false positive probability, the length of the bloom filter must grow linearly with the number of elements being filtered. In practical implementation of this method the parameter values are $k = 8$ and $m = 64 \times n$ which give FPP $= 3.63e^{-8}$.

### 3.4.3 Major steps of the algorithm

1. *Spectrum counting*: The important requirement of the SAP is to first distinguish the tuples into solid and weak tuples. For this requirement $m$ (multiplicity factor to determine whether a tuple is solid or not) Bloom filters are required. The process starts with starting all the Bloom filters bit array set to zero. While starting to process the tuples linearly, the first Bloom filter is used and if the tuple is not present in that table, the entry is added. If the entry is already present in the table then the entry is added to the next Bloom filter. This process continues for all the tuples available. Now to determine the solid tuples, each tuple again is linearly processed and queried in the Bloom filters in descending order. If the result of the Bloom filter is YES in all the $m$ bloom filter queries, then the tuple is solid, otherwise the tuple is weak. This operation is done in CPU not in parallel CUDA.

2. *Parallel error correction*: The Bloom filter is loaded into the texture memory for parallel error correction. The error correction for SAP is done by converting the weak tuples into solid tuples, which has minimum distance $\Delta$. Whenever a weak tuple is encountered, all its $\Delta-$mutation solid tuples are queried for membership in the Bloom filter. A voting matrix of these solid tuples is created based on the distance function used, which in turn is based on the type of mutation. Based on the values in the matrix and the distance function used, an appropriate solid tuple is selected, thus correcting the errors. This method is not suitable for large value of $\Delta$. So the CUDA algorithm does correction in sequences. First the correction is done for $\Delta = 1$. If the voting matrix in the process is not able to correct the error, the reads are trimmed/discarding/$\Delta-$mutation fixed in the GPU. Subsequently $\Delta$ is increased and targeted to the reads which are trimmed/discarding/$\Delta-$mutation fixed in the GPU.

Overall, the steps of our CUDA implementation for error correction with up to two errors per read are as follows.

```
1: Pre-computation spectral counting on the CPU.
2: Data transfer from CPU to GPU: Transfer bloom filter bit-vector and read data to the
   allocated texture and global memory on the GPU.
3: Execute CUDA kernel: Parallel error correction step with $\Delta = 1$.
4: Data transfer from GPU to CPU: Transfer set of error-free/corrected/trimmed/discarded
   reads to the CPU.
5: Data transfer from CPU to GPU: Transfer reads that are neither error-free nor corrected
   to the allocated global memory on the GPU.
6: Execute CUDA kernel: Parallel error correction step with $\Delta = 2$.
7: Data transfer from GPU to CPU. Transfer set of corrected/trimmed/discarded reads to
   the CPU.
```

## 3.5 Reptile

Reptile (Representative tiling for short read error correction) is a scalable short read error correction method that uses alternative decomposition of the erroneous reads and contextual information of the neighbouring substring of the reads to conclude appropriate corrections [YDA10]. Reptile creates approximate multiple alignments with the possibility of substitutions in the absence of location information. Along with using contextual information in correcting the errors it uses the $k$-mer Hamming graphs. $k$-mer Hamming graphs help Reptile retrieve all the candidates for the erroneous reads with minimum Hamming distance. Reptile also incorporates the quality score information while correcting the reads.
A $k$-spectrum for particular read $r$ is defined as set $r^k = \{r[i : i + k - 1] | 0 \leq i < l - k + 1\}$, where $r[i : j]$ denotes the substring from position $i$ to $j$ in $r$. The $k$-spectrum of the set $R$ which is the set of all the reads and their reverse complements is given by $R^k = \bigcup_{i=1}^{n} r_i^k$. Let $\alpha$ and $\beta$ be two strings such that $\alpha[(|\alpha| - c) : (|\alpha| - 1)] = \beta[0 : (c - 1)]$ for some $0 \leq c < min(|\alpha|, |\beta|)$. A *c-concatenation* of two sub string of reads $\alpha$ and $\beta$, denoted as $\alpha||_c\beta$ results in a string $\gamma$ of length $|\alpha| + |\beta| - c$ such that $\gamma[0 : (|\alpha| - 1)] = \alpha$ and $\gamma[(|\gamma| - |\beta|) : (|\gamma| - 1)] = \beta$. The Hamming distance $\text{hd}(\alpha_1, \alpha_2)$ between two strings $\alpha_1$ and $\alpha_2$ is the number of positions at which they differ. For a $k$-mer $\alpha_i \in R^k$, the $d$-neighbourhood is $N_i^d = \{\alpha_j \in R^k | hd(\alpha_i, \alpha_j) \leq d\}$. The complete $d$-neighbourhood $N_{ci}^d = \{\alpha_j | hd(\alpha_i, \alpha_j) \leq d\}$ of a $k$-mer contains of all $k$-mers within Hamming distance $d$, whether or not they occur in the set $R_k$. A tile $t$ of a read $r$ is defined as $\alpha_1||_c\alpha_2$ $(0 \leq c < k)$ only if $t$ is a substring of $r$, and $|\alpha_1| = |\alpha_2| = k$. A $d$ mutant tile is defined as $t' = \alpha'||_c\beta'$ of $t = \alpha||_c\beta$ if $hd(\alpha, \alpha')] \leq d$ and $hd(\beta, \beta') \leq d$. $T_r = (t_1, t_2, ..., t_m)$ is a tiling of read $r$ if $r = t_1||_{c_1}t_2...||_{c_{m-1}}t_m$ such that $t_i$ $(1 \leq i \leq m)$ is a tile of $r$ and $l_i \geq 1$ $(1 \leq i \leq m)$.

### 3.5.1 Major steps of Reptile Algorithm

1: Information extraction.

a. Derive the $k$-spectrum $R_k$ of R.

b. Derive Hamming graph.

c. Compute tile occurrences.

2: Individual read error correction.

a. Place an initial tile t at the beginning of the read.

b. Identify $d$-mutant tiles of $t$.

c. Correct errors in $t$ as applicable.

d. Adjust tile $t$ placement and go to step 2b, until tile placement choices are exhausted.

Information extraction is to construct the $k$-mers. The $k$-mer spectrum can be constructed by a linear scan of the reads. One of the important things to note here is that Reptile converts all non ACGT bases to A, so that they can be corrected, if wrong, later by the algorithm. The Hamming graph is where the $k$-mer are represented by vertices and each $k$-mer will be connected to its mutated $k$-mer with an edge. The edge is weighed by the distance measured in terms of mutation. To fasten the error correction process, the algorithm should be able to query all the $d$ neighbourhood $k$-mers in a constant amount of time. This requires a faster process of querying the $d$-neighbourhood $k$-mer from the Hamming graph which is provided by Reptile. Reptile successfully does this by a simple recursive approximation. Tiles are $c$-concatenations of consecutive overlapping $k$-mers. The multiplicity of the $k$-mers is calculated by a linear scan of the reads. Along this Reptile also calculates the tile occurrence, where every position of the read has a quality score exceeding some threshold $Q_c$.

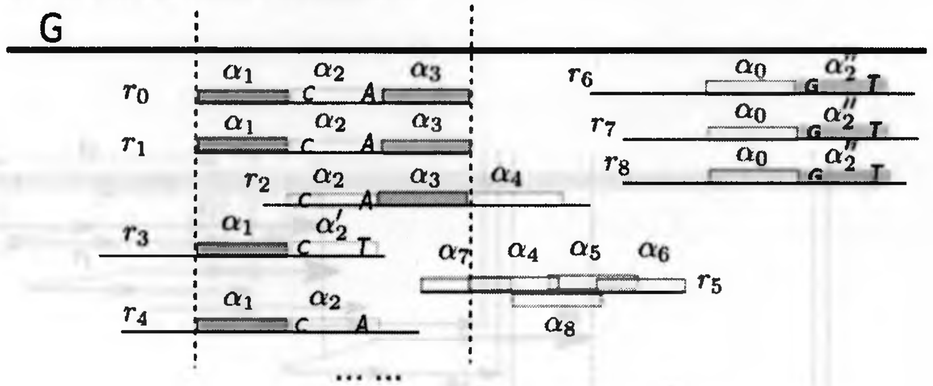


Figure 3.7: Contextual information used in Reptile

Figure 3.7 from [YDA10] shows how Reptile uses the contextual information in correcting errors. G is the target genome, shown as a bold line. The $r_i$'s ($0 \leq i \leq 8$) represent reads, shown as thin lines; $\alpha_j$ ($0 \leq j \leq 8$), $\alpha$ and $\alpha'$ are $k$mer instances in the reads, shown as rectangles. Every read is drawn aligned to its origin of sequencing position on the target genome. The bases at two positions in the $k$-mers $\alpha_2$, $\alpha'$ , and $\alpha''$ are shown. All other positions in these $k$-mers match across all three variants. Without knowing the alignment, it is unclear if $\alpha_2'$ should be corrected to $\alpha_2$ or $\alpha_2''$, since both hd($\alpha_2'$ , $\alpha_2$) = hd($\alpha_2'$ , $\alpha_2''$) = 1 and $\alpha_2$ and $\alpha_2''$ have a similar higher frequency. However, the contextual information of $\alpha_2'$ available from read $r_3$ (in this case, $\alpha_1$) uniquely identifies $\alpha_2$ as the right correction.

Error correction in the Reptile is done by choosing a tile $T_r$ from the erroneous read and replace it with its $d$-mutant tile $T_s$ which is error free. As $T_s$ is the correct tile it should be available in huge number than $T_r$ considering the uniform coverage. The mentioned technique works fine if there are very few errors in the reads. In the typical reads the errors tend to accumulate over the 3′ end of the DNA. Because of that, there would be a $d$ mutant $T_s'$ of $T_r$ which will lower the maximum number of mutations per $k$mer to below $d$ such that $T_s'$ with high probability tilings is one of the $d$ mutant tilings of $T_r'$. In order to solve this problem Reptile places a tile $T_r$ on $r$ and attempt to correct $T_r$ via comparisons with its $d$mutant tiles. If $T_r$ is validated or correct, then the algorithm moves to the next tile. But if there is a cluster of errors, for which there will be no $d$-mutant tile with high probability. Instead of moving to the next tile altogether, the algorithm jumps to the tile where it can find a $d$ mutant tile. By doing this step by step each error in the read $r$ is corrected.

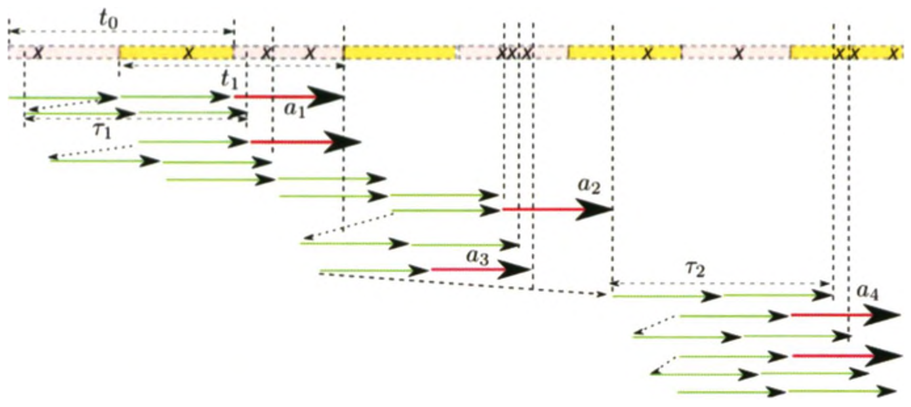


Figure 3.8: Read correction using the tile placement in Reptile

For error correction using tiles some parameters are used with respect to each tile $t$ in read R. $O_c$ is defined as the number of times the tile $t$ occur in read R. $O_g$ is the number of times the tile $t$ occur in read R where each of the base has quality score exceeding a threshold value $Q_c$. If $O_c$ or $O_g$ value is greater than a threshold value $C_q$, the tile is considered to be correct.If $O_c$ or $O_g$ value is less than a upper threshold value $C_m$, the tile is considered to be valid only if, there are no $d$-mutant tiles against it.

The tile placement during the read correction is one of the important factors in the Reptile algorithms. Figure 3.8 shows in detail how the placement of tiles is done during the read correction. Initially a two tiles $t_0$ and $t_1$ are chosen. Having single error in each of the tile, after the correction the tile placement is moved by jumping a distance of one tile at once. But in the next tile sequencing there are more than one error. So the tile placement is readjusted to see that there will be only one error in the tile. If there are two errors in a tile, then it is not able to correct. To find $d$ mutants the time complexity increases exponentially with increase in $d$. Therefore it is better for the algorithm to stick to the minimum value of $d$. The read correction in the Figure 3.8 is explained using the $d$'s value as 1.

The parameter value of $\mathrm{Q_c}$, $\mathrm{O_c}$, $\mathrm{O_g}$, $\mathrm{C_q}$ and $\mathrm{C_m}$ can affect the performance of the Reptile. Therefore these parameters have to be chosen very carefully and much after statistical and practical analysis. The parameter values in Reptile are chosen after careful analysis of the input data. The $\mathrm{Q_c}$ is choose such that 15% to 20% of the bases have quality scores below $\mathrm{Q_c}$, $\mathrm{C_g}$ is chosen such that a small part(1% to 3%) of the tiles have its occurrence value above $\mathrm{C_g}$, $\mathrm{C_m}$ is chosen such that a large part(4% to 6%) of the tiles have its occurrence value more than $\mathrm{C_m}$. As the $\mathrm{C_m}$ value decrease more errors are corrected, but there is high probability of false error correction. Increasing $\mathrm{C_g}$ increases the quality of error correction. The value of $k$ is set to $log_4G$. Hamming distance is defaulted to one. These parameters are changed with respect to input data.

## 3.6 HiTEC

The basic idea used in the HiTEC (High Throughput Error Correction) [IFI10], error correction method is to construct the suffix array of the string $R_s$, where $R_s = r_1\$\bar{r}_1\$r_2\$\bar{r}_2\$....r_n\$\bar{r}_n$. and use it to then determine and correct a wrong base, with support of a witness string $u$. The idea is very simple and is explained in detail as the basic error correction in section 3.1. In the next few sections the details about how the support is determined and the underlying data structure for the witnesses are explained.

### 3.6.1 Suffix array

The suffix array is a text index that was built mainly to compensate the memory and time complexity of suffix tree. It was developed by Myers and Manber [MM93]. The suffix array is an array of numbers holding the starting position of the suffixes sorted in the lexicographical order. Consider for example the string S = ACTAACACTGG. The alphabet set is $\sum = \{A,C,T,G\}$ the four bases of the DNA. In Table 3.1, the $i$ column is the lexicographical index order and SA[$i$] is the corresponding value of the starting position of the suffix in the string $s$ in lexicographical order. The LCP gives the *Longest common prefix* between the two consecutive suffix array elements, that is, between SA[$i$] and SA[$i-1$]. If $|S| = m$, then the SA can be computed in O($m$) time and space by any of the algorithms of Karkkainen and Sanders (2003) [KS03], Kim [Kim05], Ko and Aluru (2005) [YDA10]. The LCP array can be computed also in O($m$) time and space by the algorithm of [Tor01]. HiTEC has used the libdivsufsort library of Yuta Mori[3] in the program.

| $i$ | SA[$i$] | $\text{suf}_{\text{SA}[i]}$ | LCP[$i$] |
|---|---|---|---|
| 1 | 4 | AACACTG | 0 |
| 2 | 5 | $\underline{\text{A}}$CACTG | 1 |
| 3 | 1 | $\underline{\text{AC}}$TAACACTGG | 2 |
| 4 | 7 | $\underline{\text{ACT}}$G | 3 |
| 5 | 6 | CACTG | 0 |
| 6 | 2 | $\underline{\text{C}}$TAACACTG | 1 |
| 7 | 8 | $\underline{\text{CT}}$G | 2 |
| 8 | 10 | G | 0 |
| 9 | 3 | TAACACTG | 0 |
| 10 | 9 | $\underline{\text{A}}$ACACTG | 1 |

Table 3.1: Suffix array of string ACTAACACTGG

[3]libdivsufsort: A lightweight suffix sorting library, http://code.google.com/p/libdivsufsort/.

## 3.6.2 Statistical analysis

The novelity brought by HiTEC was a thorough statistical analysis that enabled correction of more errors than all the other programs. Considering *u* to be the support of length *w*, the cluster of *u* is defined as the set of all the positions where the witness *u* appears in the string R with a leading base value other than \$

$$clust(u) = \sum_{a \in \Sigma} supp(u, a).$$

The cluster of *u* is very easy to find in SA, because all the suffixes are lexicographically ordered. LCP makes it even easier to find out the cluster of *u*. The occurrence of the witness string *u* of length *w* is random in the genome. To detect an erroneous base it is very important that the witness *u* does not appear anywhere else in the genome. HiTEC models this randomness using a *Bernoulli model* and derives *w*, the length of the witness *u*. The witness length *w* derived using the Bernoulli model will be balanced in such a way that there is low probability of that witness to appear some where in the genome and have enough occurrence to successfully detect an erroneous base. There are two cases to consider here. First the witness *u* is correct and the base following it is also correct. The probability of any substring of length *l* in the genome and the read sequenced for that exact substring having a correct witness of length *w* with a correct base following it is

$$q_c = \tfrac{l-w}{L}(1-p)^{w+1}$$

If $R_c$ is the number of reads satisfying the mentioned condition, then

$$Prob(R_c = k) = \begin{pmatrix} n \\ k \end{pmatrix} q_c^k (1 - q_c)^{n-k}$$

Thus the expected number of pairs (*u*, *a*), both *u* and *a* correct, given that supp(*u*, *a*) = *k*, is

$$W_c(k) = \begin{pmatrix} n \\ k \end{pmatrix} q_c^k (1 - q_c)^{n-k} L$$

Second case is the witness *u* is correct and the base following it is wrong. Giving the same reason as that in case one the probability of the second case to occur is

$$q_e = \tfrac{l-w}{L}\tfrac{p}{3}(1-p)^{w}$$

The number of reads $R_e$ has the probability distribution

$$Prob(R_e = k) = \begin{pmatrix} n \\ k \end{pmatrix} q_e^k (1 - q_e)^{n-k}$$

Thus the expected number of pairs $(u, a)$, where $u$ is correct and $a$ is wrong, given that supp$(u,a)$ $= k$, is

$$W_e(k) = \begin{pmatrix} n \\ k \end{pmatrix} q_e^k (1 - q_e)^{n-k} L$$

One may argue that a third case exists, when an error is present in the witness $u$. If there are errors in the witness, those occurrence will anyway be few. The threshold value $T$ is derived from $k$ in the interval where $W_c(k)$ and $W_e(k)$ are very small. As the error rate decreases the interval increases. So $T$ should remain good when some of the errors are corrected. The formula for $T$ is

$$T = min(\{k | W_c(k) > W_e(k)\}) + 2$$

One more parameter which plays an important role in correcting most of the errors is the witness length $w$. Witness length can not be just randomly picked. Because if the witness length is small, the chances that the same witness appears somewhere else in the genome are high. The consequence will be that some of the erroneous reads are validated as correct reads. If the witness length is large then the support for the witness will be very small to detect the erroneous base. HiTEC uses a statistical approach to calculate the witness length $w$. Again, in order to successfully predict an appropriate value for the witness length we need to consider two cases. First the number of uncorrectable reads. The uncorrectable reads are those where the witness $w$ when fitted any where in the read has errors in it. To estimate this we need to determine a way of placing $k$ errors in the read of length $l$, so that the witness placed anywhere in the read will have at least one error. The function which reflects the required behaviour is $f_w(k, l)$.

$$f_w(k, l) = \begin{cases} \begin{pmatrix} l \\ k \end{pmatrix}, & if l < w, \\ 0, & if k < \lfloor \frac{l}{w} \rfloor, \\ \sum_{i=1}^{w} f_w(k-1, l-i), & otherwise \end{cases}$$

Using this function, the number of uncorrectable reads with the witness length $w$ is $f_w(k, l)p^k(1-p)^{l-k}$.

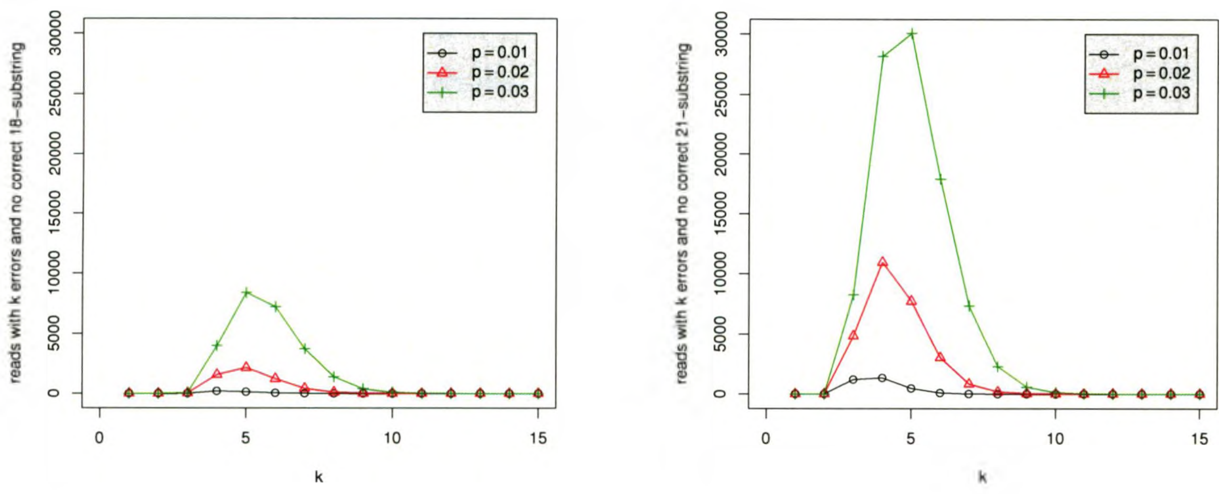


Figure 3.9: The number of reads with a given number of errors and no error-free interval of length w for L = n = 4.2 mil. and l = 70. The right plot uses w = 21 and the left w = 18.

Figure 3.9 from [IFI10] plots the total number of uncorrected reads for witness lengths of 21 and 18 for a genome size of 4MB. The important point we can derive from the graph is that, as the witness length decreases the number of uncorrectable read count plummets. The downfall is, the smaller the witness length the higher the chances of that string appearing somewhere else in the genome which causes correct bases to be changed to wrong ones. This brings us to the second case, where we need to calculate the number of reads which are destroyed because of the above mentioned scenario. That is the witness being wrong, so that the correct bases are deemed wrong and changed. The probability of the witness $u$ having error and posing as string $v$ which is present in the genome followed by a different base is given by

$$q_w = (1 - (1 - p)^w)(1 - p)(1 - (1 - \tfrac{1}{4^w})^L)\tfrac{3}{4}$$

The total number of destructible reads using the above mentioned probability is

$$D(w) = (1 - (1 - q_w)^{l-w})(1 - p)^l n$$

Clearly we can see that both cases mentioned are going to hurt the error corrections process badly. HiTEC takes a common ground by deriving the witness value

$$w_m = \arg\min_w(U(w) + D(w))$$

Figure 3.10 shows the values of $U(w) + D(w)$ for various witness lengths. It is theoretically shown that, to get the highest accuracy the best witness length is $w_m$. However in practical it turns to be combination of witness lengths around $w_m$ and smallest value $w$ for which no correct reads are changed will yield the best results:

$$w_M = min(\{w|D(w) < 0.0001E_e\})$$

HiTEC uses the following witness lengths in sequence for 9 iterations in the software:

$$w_{seq} = w_m + 1, w_M + 1, w_M + 1, w_m, w_m, w_M, w_m - 1, w_M - 1, w_M - 1$$

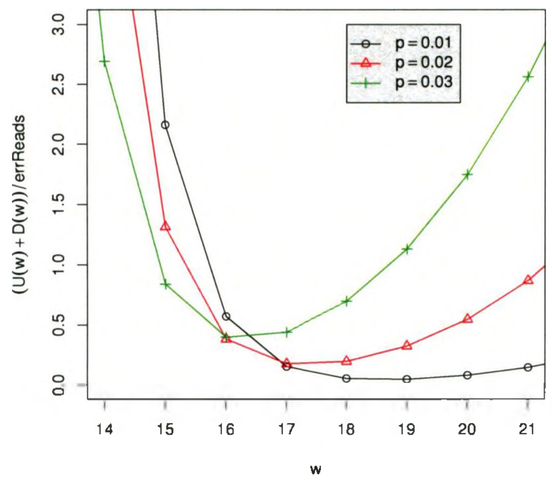


Figure 3.10: The values of $U(w) + D(w)$ as percentages of the total number of erroneous reads, $E_e$, for $L = n =$ 4.2 mil., $l = 70$, and $p = 0.03$.

The software runs in iterations and each iteration picks up different values of witness lengths. The number of iterations ran by HiTEC software is maximum of 9 and the only reason the program stops before iteration 9 is when the number of bases corrected by the program is below some threshold value the threshold value being 0.01% of the total number of bases.

## 3.6.3 Algorithm

The pseudo code for HiTEC is given in algorithm 1. The inputs HiTEC algorithm accepts are the error rate $p$, genome length $L$ and input reads. The rest of the parameters are calculated as mentioned in the section 3.6.2 using the statistical approach. The support for a witness is calculated online instead of LCP because the online algorithm due to cache effect will reduce the complexity of space and time. If there is enough support for a witness then the erroneous base is changed. If the algorithm is ambiguous whether the base is erroneous or not, the successive bases are considered in determining the appropriate decisions. The algorithm runs for a maximum of 9 iteration or can end before that if the total number of bases changed in the particular iteration is less than the 0.01% of the total bases in the read file. To provide best time and space complexities, HiTEC also splits the high coverage input read files into reads of coverage 70 each.

**Algorithm 1** HiTEC

```
given: n reads $r_1, r_2, ...., r_n$ (of length $l$ each), $L$ and $p$
return: Corrected reads in the same input format
1: compute $w_M$ and $w_m$
2: compute $T$
3: $i \leftarrow 1$
4: repeat
5:    $c \leftarrow 0$
6:    $w \leftarrow w_{seq}[i]$
7:    Construct R and compute SA and LCP
8:    Compute the clusters in SA for all witnesses of length w
9:    for each witness u with $clust(u) \leq T + 1$ do
10:      $Corr \leftarrow \{a \mid supp(u,a) \geq T\}$
11:      $Err \leftarrow \{a \mid supp(u,a) \leq T - 1\}$
12:      for each $a \in Err$ do
13:         if $| Corr |= 1$ then
14:            correct a to $b \in Corr$
15:            $c \leftarrow c + 1$
16:         end if
17:         if $| Corr | \geq 2$ then
18:            for each $b \in Corr$ do
19:               if ua, ub followed by same two letters then
20:                  correct a and b
21:                  $c \leftarrow c + 1$
22:               end if
23:            end for
24:         end if
25:         $i \leftarrow i + 1$
26:      end for
27:   end for
28: until $((\frac{c}{ln} < 0.0001)$ or $(i > 9))$
29: return all $r_j$'s from R
```

# Chapter 4

# A fast implementation of HiTEC

When we analyse HiTEC solution for error correction in detail, the important aspect we notice is that the suffix array uses a high amount of space and time to construct it. The main goal of this thesis is to reduce this space and time. The purpos of the suffix array in HiTEC is to get the witnesses and the witness counts. In the fast implementation of HiTEC we propose, the witnesses and witness counts are calculated using 2-bit encoding of the bases and are stored in an hash table (array of witness). To speed up the process of finding an element in the hash table, linear probing technique is used. Using the mentioned techiques, the error correction algorithm uses significantly less space and less time in correcting errors.

## 4.1 Hashing

Most of the computer science algorithms use some type of search mechanism to fulfill their purpose. It is necessary to have an efficient search mechanism where the results can be obtained in constant amount of time, that is O(1) time complexity and with minimum storage requirements. To understand the above requirement in detail, we will consider an example. Consider a school that has 100 students in it. Each of them has a student *Id* ranging from 1 to 100 and we have to store them in an array. An array of size 100 will be allocated and each one of the students record can be identified in the array using the index in a constant time that is O(1) time complexity with a space requirement of 100. If we notice the above solution in detail we can conclude that there is a one to one correspondence between the element key and the array index.

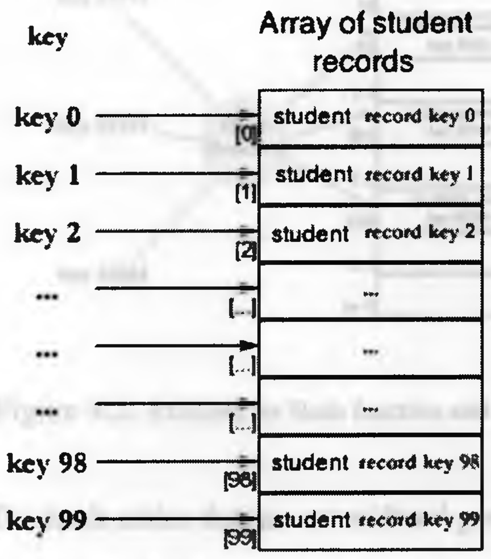


Figure 4.1: Example for Hash table

However in reality establishing this perfect one to one relationship is quite a challenge. Now consider the same school with the same 100 students but the student *Id* ranges from 0 to 99999, because the school decided to use 5 digit number as a primary key. Now it is not a good decision to have an array of 100000 elements, in which only 100 will hold a value. Instead we can trim the first 3 digits and using the last two digits to map it to a two digit indexed array. For example 45678 will be mapped to array element with index 78 and 76545 will be mapped to array element with index 45. Doing this would mean that the elements are not stored according to the key value as per previous design. Therefore there is a necessity of converting the five digits Id into a two-digit array index. The function used for the conversion is called *hash function* and the array is called the *hash table*.

### 4.1.1 Hash function and hash tables

A hash table is a data structure consisting of two parts, the key and the value, where the key points to the value. The easiest way to visualize the hash table is by visualizing the array data structure. The array index acts as the key and the data the location holds is a value. In this hash table any time can be searched in a constant amount of time that is O(1), amortized time

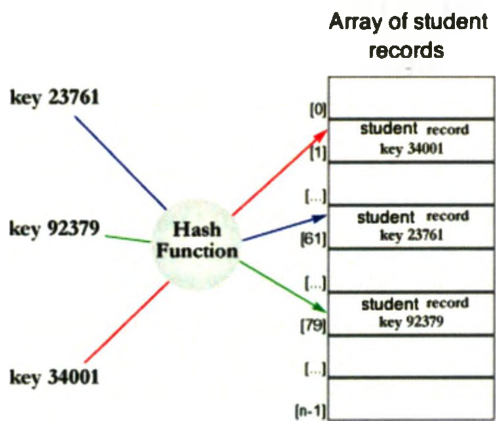


Figure 4.2: Example for Hash function and Hash table

using a hash function. The hash tables that are considered good also exhibits property such as constant time O(1) for insertion and deletion of elements into the hash table [Knu88].

A hash function is a mathematical function which when applied to a key will produce a value that can be used as an address to the value. The intention here is to spread the elements uniformly and relatively random. Unfortunately reaching a solution to the hash function is not quite simple. Considering the school example and the mechanism of using the last two digits of the five digits to map the keys. Student *Id* 45678 will be mapped to the array element with index 78 and 76545 will be mapped to array element with index 45. What about 86745? This will also be mapped to the location 45, which contains the data of student with student id 76545. This phenomenon is called a collision. In plain terms collision is a condition resulting when a hash function produces the same value for two or more keys. Collisions are important part of the hash table and a major factor considered while building a hash function.

The main question is can we design a hash function that doesn't have any collision? The answer varies with the requirements. Sometimes a hash table without collision can be designed and those are called perfect hash functions. But in most of the practical scenarios it is impossible to create a perfect hash function. So, we settle for the next best that is good hash functions that have minimum collision by spreading the elements uniformly over the address space. Designing a hash function may be very complex due to the collision issues. Below are some of the steps used while designing the hash scheme for HiTEC2.

1. Set the hash table size to a prime number. (Eventually this prime number is used in the

hash function). This helps in uniformly distributing the key over the address space. The explanation is quite simple; we will use this prime number that will be labeled as TABLE_SPACE in our hash function to get an address.

Due to the nature of maths, if the constant used as modulus in the hash function and the input variable are co-prime, then the collisions will be minimized and that minimizes the clustering in the hash tables. If the hash table size is not a prime then the multiple common factors are obviously greater than one for most of the inputs. When the inputs are millions in number and the hash function has a modulus operation, it is better to have prime number to do the modulus of the input key, so that we are confident that the result of Greatest Common Factor (input, hash table size) = 1. Appendix, shows the list of the primes used in HiTEC2. We use a list of primes because the size of the hash table is set based on the input file size, so that the memory used will be minimum.

2. Transformation of inputs: In HiTEC2 there is a unique two-bit representation of the 4 alphabets in DNA sequencing. So a string can be converted into a series of bits, which in turn will be an integer. The remainder of this integer when divided by the TABLE_SPACE gives the key in the hash table. The detailed explanation of this transformation is given in the Section 3.3.1

3. Collision handling: Even when TABLE_SPACE is a prime number, there will still be many collisions. There should be a mechanism in place to handle the collisions. HiTEC2 uses one of the most common and efficient solutions called Open Addressing with Linear Probing (see below).

### 4.1.2 Open Addressing

Open addressing is a method of collision resolution in hash tables, where all records are stored in the array itself (as opposed to, e.g., separate chains). The solution to the collisions is given by a mechanism called *probing*. Probing means searching alternative locations in the array until the target record is found or an empty spot is found where the target record inserted [TLA90]. There are many types of probing; the most important ones are linear, quadratic and double probing. In linear probing the interval between probes is fixed and it is usually 1. Quadratic probing increases the interval by values of a quadratic polynomial. Double probing invokes another hash function to determine the probing sequence. The two important factors which helps us to determine which probing mechanism is best for our requirements is *cache performance* and *sensitivity to clustering*. In HiTEC2 linear probing is used to get the advantage of cache performance as the program deals with huge data and the sensitivity of clustering is anyways handled by allocated enough space for the hash table.

### 4.1.3 Linear probing

The simplest probing method in open addressing is linear probing. The simplicity comes from the fact that the sequence of probing is consecutive. The probing is very fast due to cache effect the probing will be faster. With all these advantages there is a disadvantage as well. To get a clear understanding here is an example. Let the size of the hash table be 10. Let the hash function be simple modulus function, that is H(k) = k mod 10. Initially all the elements in the array are empty. Insert the keys 15, 17, 8 (see Figure 4.3). When we insert 25, we compute P = H(25) = 25 mod 10 = 5. Position 5 is already occupied by 15. So we must look elsewhere for a position in which to store 25. Using linear probing, the position we would try next is 5+1 mod 10 = 6. Position 6 is empty, so 25 is inserted there.

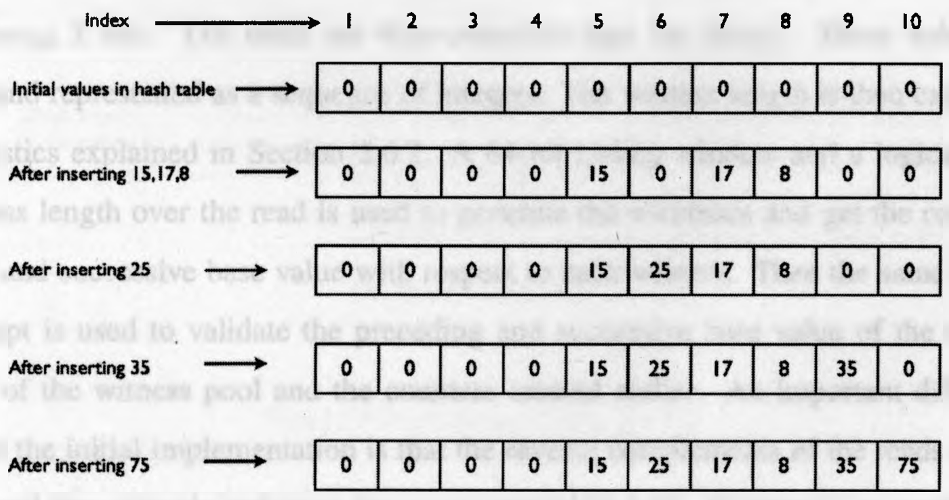

index 1 2 3 4 5 6 7 8 9 10
Initial values in hash table 0 0 0 0 0 0 0 0 0 0
After inserting 15,17,8 0 0 0 0 15 0 17 8 0 0
After inserting 25 0 0 0 0 15 25 17 8 0 0
After inserting 35 0 0 0 0 15 25 17 8 35 0
After inserting 75 0 0 0 0 15 25 17 8 35 75

Figure 4.3: Example for Linear probing

Now suppose we insert 35. We first try position 5 =35 mod 10. It is occupied. Using linear probing we next try positions 6 =(5+1) mod 10, 7 = (6+1) mod 10, 8 = (7+1) mod 10. Finally position 9 = (8+1) mod 10 is tried and it is empty so 35 is inserted there. Now suppose we insert 75. 5, 6, 7, 8, 9 are tried unsuccessfully. Finally we try position 0 = (9+1) mod 10; it is empty so 75 is stored there. The potential problem of linear probing is evident from the above example. With the increase in clusters in the table, finding the value in the table takes more and more time. In HiTEC2 the experimental results show that allocating space twice the amount of space required to store the target number of keys for hash table will keep the clusters small. Allocating twice the space required for the hash table will spread the keys over the whole hash table, which result in less collision, making the retrival of the value faster.

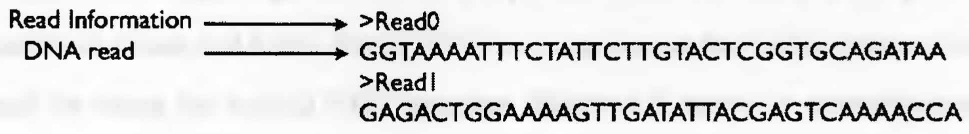
## 4.2 Implementation

The basic idea of error correction in HiTEC has been already explained in Section 3.1. HiTEC2 uses the same basic error correction method but the approach used to implement it is different than that of any other error correction techniques. HiTEC2 encodes each of the letters A, C, G and T using 2 bits. The reads are thus converted into bit strings. These strings are concatenated and represented as a sequence of integers. The witness length is then calculated using the statistics explained in Section 2.6.2. A 64-bit sliding window and a logical AND mask of witness length over the read is used to generate the witnesses and get the counts of the preceding and successive base value with respect to each witness. Then the same sliding window concept is used to validate the preceding and successive base value of the witness with the help of the witness pool and the counters created earlier. An important difference with respect to the initial implementation is that the reverse complements of the reads are not explicitely stored thus already reducing the space needed in half. However, we need clever manipulation techniques to make use of reverse complements directly from the reads. The details are explained in detail in the next few sections.

### 4.2.1 2-bit encoding of the reads

The DNA alphabet consists of 4 letters A, C, T, and G. The DNA sequence is a long sequence of these 4 distinct values. The DNA is sequenced and obtained in the form of reads with a specific read length. There are two kinds of read files one can pass as an input to HiTEC2, *fasta* and *fastq*.

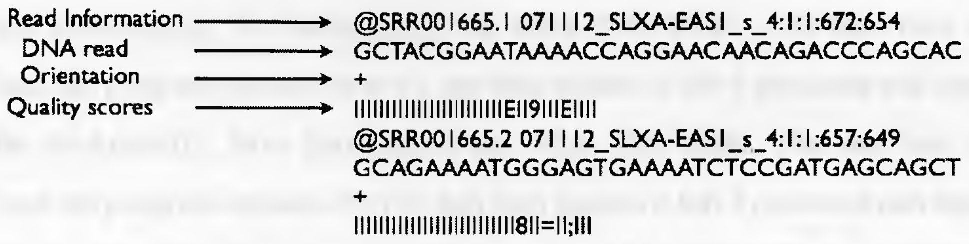
Each read in a fasta file typically occupies two lines. The first line always starts with > and is followed by the information about the DNA hosts id and the read number. The second line is the DNA sequence of the specific length. Figure 4.4 will show an example of fasta file.



```
>Read0
GGTAAAATTTCTATTCTTGTACTCGGTGCAGATAA
>Read1
GAGACTGGAAAAGTTGATATTACGAGTCAAAACCA
```

Figure 4.4: Example of fasta read file

In fastq files, each read occupies 4 lines. First line always starts with @ and is followed by the information about the DNA hosts id and the read number. Second line is the DNA sequence. The third line starts with either + or -, depending on whether the DNA read in the second line is from 3' to 5' or 5' to 3' end orientation. The fourth line is the quality score of each of the DNA base. The quality score of a base defines the probability of correctness of that base. Figure 4.5 will show an example of fastq file.



```
@SRR001665.1 071112_SLXA-EAS1_s_4:1:1:672:654
GCTACGGAATAAAACCAGGAACAACAGACCCAGCAC
+
IIIIIIIIIIIIIIIIIIIIIIIIEII9IIIEIIII
@SRR001665.2 071112_SLXA-EAS1_s_4:1:1:657:649
GCAGAAAATGGGAGTGAAAATCTCCGATGAGCAGCT
+
IIIIIIIIIIIIIIIIIIIIIIIIII8II=II;III
```

Figure 4.5: Example of fastq read file

The two bit encoding of the DNA is simple: base A is represented as 00, base C as 01, base G as 10 and base T as 11 and any ambiguous base N is converted to A and is represented as 00. Encoding the DNA bases with these specific values has valid reasons behind it. As we know the

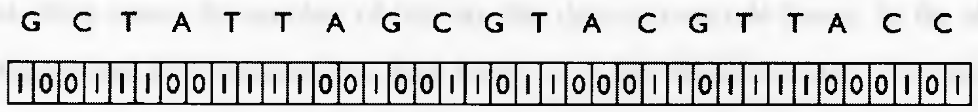


Figure 4.6: Example of read and its binary encoded form

complement of base A is base T and complement of base C is base G and vice versa. The DNA bases are encoding in such a way that the binary encoding of the bases when complemented yields the binary values of the complement bases. For example base A is encoded as 00 and when 00 is complement we get 11, which in turn represents base T, which is complement of base A. This makes it easier and faster for HiTEC2 to complement the whole witness value of whichever length by using the logical NOT operator. Figure 4.6 shows an encoding example in detail.

In HiTEC2 the whole read file is read into a character array and based on the type of file the loops are executed to extract only the DNA sequence in the read data. Each read while parsed from starting till the end is encoded base by base into an 8-bit integer array. The encoding is straightforward but placing them in the array is a bit tricky. Let us consider and example to illustrate how this is done (see also Figure 4.7). Consider a read with read length 10 and the DNA sequence GTCGTACGTC. Initially the 8 bit integer array *binArray*[ ] will be initialized to 0 and the index starts with 0. When our program reads the DNA read sequence, it first encounters G and encodes it to 10. Our program then pushes it left 6 positions and logically OR's it with the binArray[0]. So binArray[0] has value 1000 0000. The next base in the sequence is T and the program encodes it to 11 and then pushes it left 4 positions and logically OR's it with the binArray[0]. Now binArray[0] has value 1011 0000. The next base in the sequence is C and the program encodes it to 01 and then pushes it left 2 positions and logically OR's it with the binArray[0]. Now binArray[0] has value 1011 0100. The next base in the sequence is G and the program encodes it to 10 and logically OR's it with the binArray[0]. Now binArray[0] has value 1011 0110. At this point the 8-bit integer value in binArray[0] holds 4 bases and now the index value of the array is incremented by 1. The process is repeated again and again till the bases in the read are exhausted. If the read does not fit exactly into the multiples of 8 bit integer, the rest of the bits of that integer will be defaulted to 0 and a variable named offset stores the number of bits are that does not encode bases. In the above example the offset would be 4 because the read length is 10 and when encoded will take 20-bit space. 20 bits require 3 bytes where the last byte has 4 nonencoding bits (the least significant ones).

In the Figure 4.8 there are two examples. One shows the DNA read of length 10 encoded and put into 3 bytes with the bits in the cells colored red are nonencoding bits and offset is set to 4. In the other example a DNA read of length 12 is encoded and put into 3 bytes that fits exactly and the offset is set to 0. The example illustrates why we choose an 8-bit integer array to store the encoded binary values of the read. The answer is to have a low maximum offset; always almost 6, so only up to 6 bits of space get wasted for each read. If we chose 32 or 64 bits, a significant amount of space could be wasted.

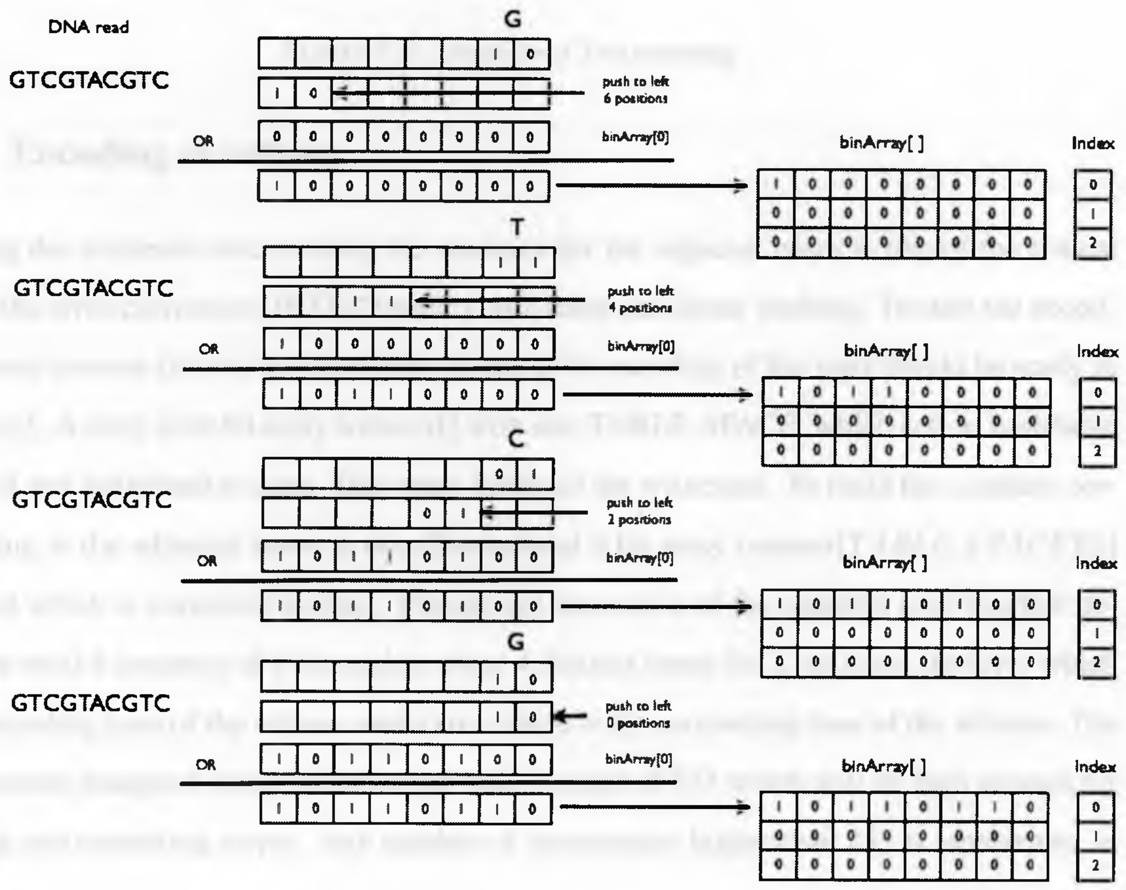


Figure 4.7: 2-bit encoding of reads

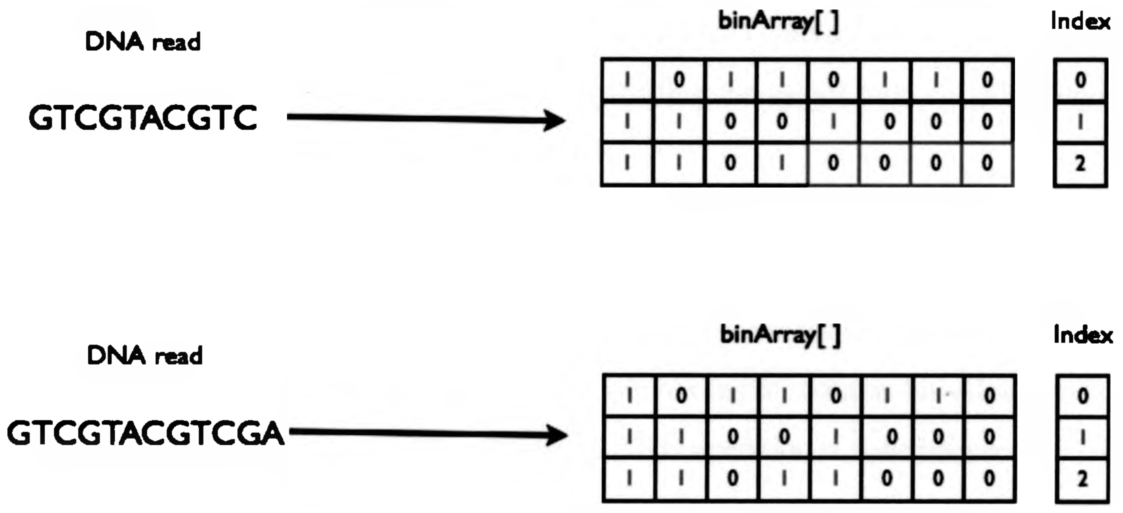


Figure 4.8: Examples of 2 bit encoding

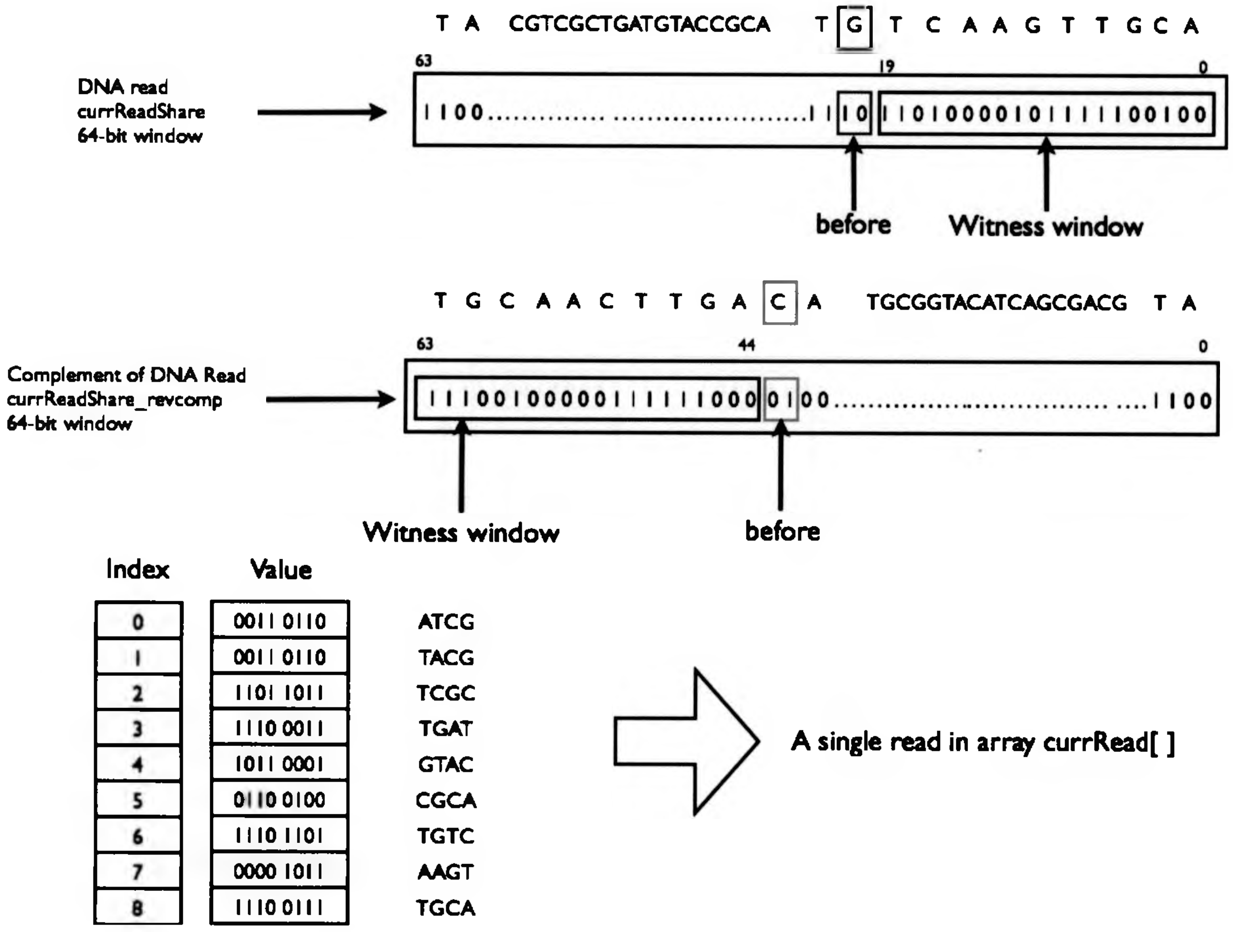
### 4.2.2 Encoding of witness

Encoding the witnesses and creating the counters for the adjacent bases is one of the critical steps in the error correction. HiTEC2 uses a hash table and linear probing. To start the encoding witness process (shown as Algorithm 3), the 2-bit encoding of the read should be ready in *binArray*[ ]. A sixty-four bit array *witness*[ ] with size TABLE_SPACE, which acts as hash table is created and initialized to zero. This array holds all the witnesses. To build the counters corresponding to the adjacent bases, a two-dimensional 8 bit array *counter*[*TABLE_SPACE*][8] is created which is initialized to zero. The second dimension of the counters is of length 8 because we need 8 counters of 8 bit each to store 4 distinct bases for 2 positions, *before*, which is the preceding base of the witness and *after* which is the succeeding base of the witness. The 8−bit counter unsigned integer counter can hold a count of 255 which will be high enough for detecting and correcting errors. Any number of occurrences higher than 255 is represented as 255.

*binArray*[ ] has all the reads data in binary, from which a single read data is copied into an array, called *currRead*[ ]. This can be easily done if we know the read length, that is by copying the data from *binArray*[ ] from index 0 to read length into *curRead*[ ] (Algorithm 2

step 2). The last 64 bits of the *curRead*[ ] are loaded onto a 64 bit integer (*currReadShare*), while another 64–bit integer is loaded with the reverse complement of the *currReadShare*, as shown in the Figure 4.9. (The procedure for loading the values onto *currReadShare* and *currReadShare_revcomp* is given in Algorithm 4)
Then a witness widow is created which is twice the witness length, because each base is encoded into two bits (Algorithm 2 step 3). This witness window is then placed over to the rightmost end of the *currReadShare*. The bits value in the mask are obtained by doing a logical AND operation with the witness mask. The witness mask contains bit value 1 inside the witness window and 0 elsewhere. When a logical AND operation is done between *currReadShare* and the witness mask only the value inside the witness window is obtained as output (Algorithm 2 step 17). At the same time the witness window will be placed to the leftmost end of the reverse complement of currReadShare and get the value of the reverse complement of the witness using the same masking technique (Algorithm 2 step 18).
Now these two integer values of the witness and its reverse complement are compared to see which is smallest (step 19). This is done so that both the counts of straight and the reverse complement witness can be stored in the same place, thus reducing the memory for counters in half. The smallest value is then subjected to the modular hash function, modulus of TABLE_SPACE, to get the index value in the array *witness*[ ] to store the witness integer value, if that index value already holds a witness value then the linear probing is done to get a place to store the witness value (Algorithm 5).

As shown in Figure 4.9, the witness window helps in extracting the witness value; *after* and *before* bases need to be extracted from *currReadShare* to know for which base the witness string is a *witness*. They are extracted using the same masking procedure, with a mask of only two bits at the proper position (Algorithm 3 step 10 and 14). The *after* base A, C, G and T counts are stored in *counters*[ ][0..3] and the *before* base A, C, G and T counts are stored in *counters*[ ][4..7] respectively. Before incrementing the corresponding counter, the value of the counter is checked for less than 255 condition. If a counter with value 255 is incremented, then the value of counter will remain unchanged (because the counter is 8 bit can not hold higher values). One important thing to note down here is that we are storing both the straight and the reverse complement witness counts in the same place, So the *after* and *before* are relative to



| Index | Value | |
|---|---|---|
| 0 | 0011 0110 | ATCG |
| 1 | 0011 0110 | TACG |
| 2 | 1101 1011 | TCGC |
| 3 | 1110 0011 | TGAT |
| 4 | 1011 0001 | GTAC |
| 5 | 0110 0100 | CGCA |
| 6 | 1110 1101 | TGTC |
| 7 | 0000 1011 | AAGT |
| 8 | 1110 0111 | TGCA |

Figure 4.9: An example of read with sequence ATCGTACGTCGCTGATGTACCGCATGTCAAGTTGCA of length 36 is stored in *currRead*[ ] array. The *currReadShare* and *currReadShare_revcomp* both 64 bit variable holding a share of 64 bit of the *currRead*[ ]. The witness window of length 20 bit is shown and the *after* and *before* bases are shown in green and blue box respectively.

the witness or witness_revcomp based on which is the *witnessMinValue* (Algorithm 6).

In the next iteration the 64 bit window is shifted 2 bits left in *currReadShare* and 2 bits right in reverse complement of *currReadShare* as shown in Figure 4.10. The iteration will stop when all the witnesses are exhausted in the read, that is total of $(readLenInBits - witLenInBits + 1)$ witness in each read. After processing every 4 bases, the algorithm check if there exists an extra read information in the *currRead*[ ] array, if yes then the next byte of information is added to the most significant byte of the *currReadShare*. The same applies for the *currReadShare_revcomp*, but here the reverse complement of the next byte is added to the least significant byte (Algorithm 7).

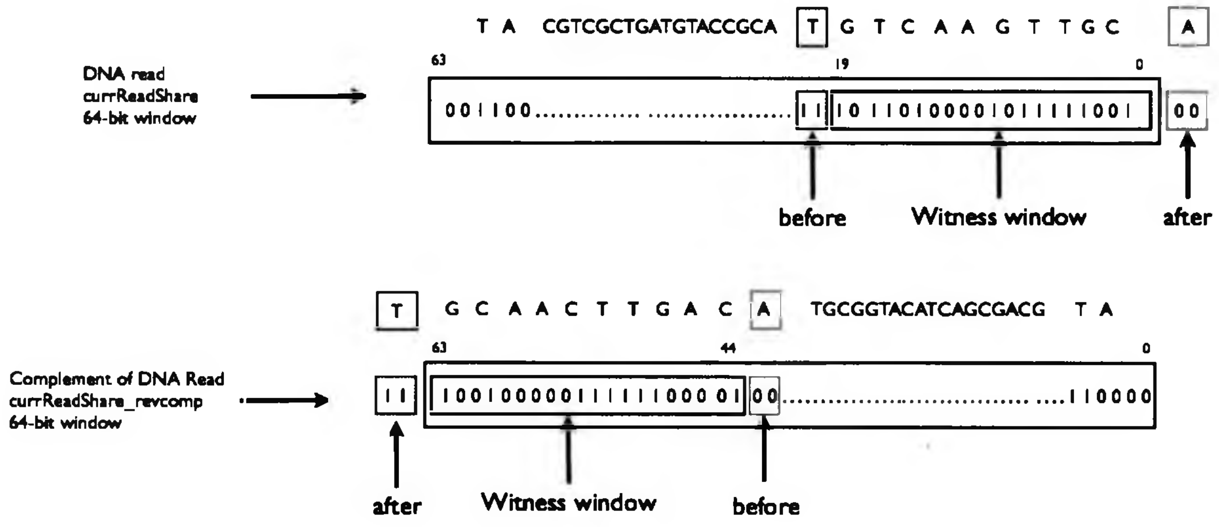


Figure 4.10: The 64-bit window is shifted 2 bits left in *currReadShare* and 2 bits right in *currReadShare_revcomp*. The *after* and *before* bases is marked in green and blue box respectively. Based on the minimum value of the witness the proper *after* and *before* base counters are incremented. When the 64-bit window is shifted the zeroes will be added to the vacant bits. This after processing every 4 bases will be replaced by the next consecutive byte

## 4.3 Detecting and Correcting Errors

Once the witness pool and the counters are calculated, the detection and correction process can start. The 64 bit sliding window, extraction of the witness and its reverse complement, computation of *after* and *before* bases are the same as in Algorithm 2. The *witness*[ ] and *counters*[ ][ ] are used to detect and correct errors. Initially the *after* and *before* bases are assumed to be correct unless the *counters*[ ][ ] prove it wrong. Algorithm 9 does this job for HiTEC2. If the count of the current base value is not higher than the threshold, the *counters*[ ][ ] are parsed to find whether the count of another base is higher than the current one and above the threshold (refer section 2). If yes then the erroneous base will be replaced with the correct base in both currReadShare and currReadShare_revcomp. A important thing to note down here is the avalanche phenomenon. That is once an erroneous base is corrected, the next witness contains the corrected base in the witness. This helps in correcting more than one error in one read if present, in a single iteration. This avalanche phenomenon is an important factor in correcting the errors in as few iterations as possible.

Once the errors in a reads are corrected, the read is put back in to *binReads*[ ]. This process

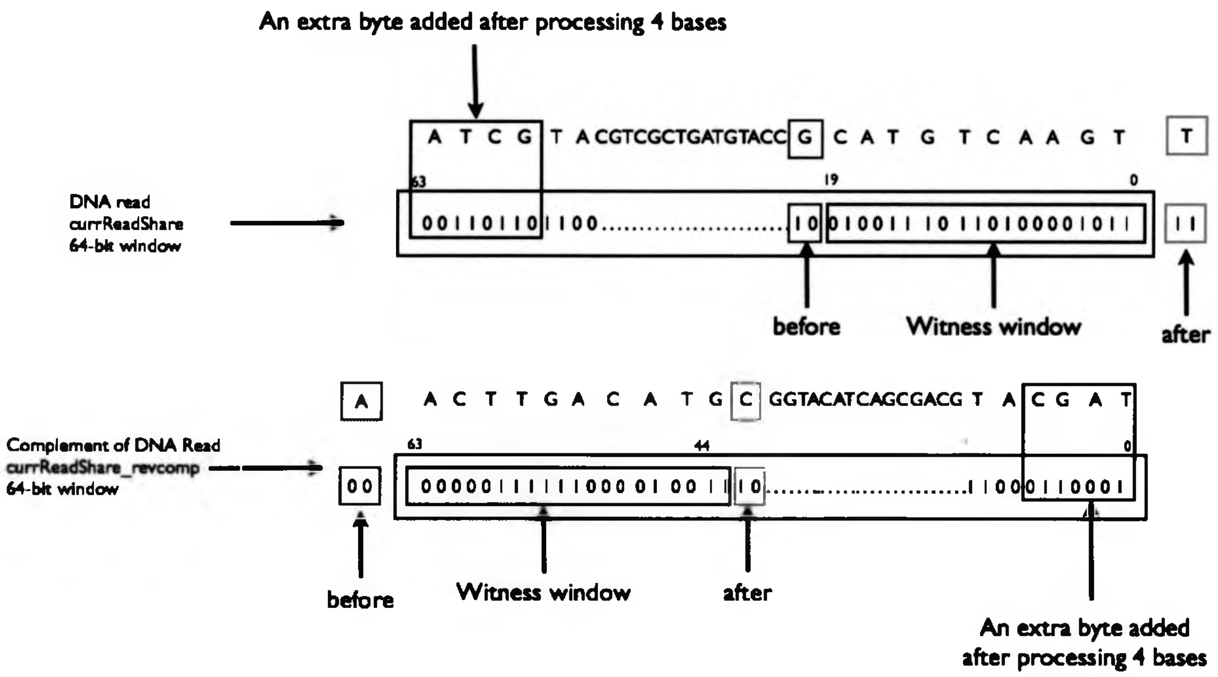


Figure 4.11: If a read length is more than 64 bits the next byte is added after processing every 4 bases

goes on for a number of iterations, till the number of bases changed in one iteration due to correction fall below 0.01 percent of the total number of bases or till iteration 9, whichever is reached first. This condition is necessary because after this particular condition is reached, the algorithm is will correct very few errors in one iteration. Moving forward with another iteration means a lot of time consumption with very very less correction. So the satisfactory line is drew with this condition. Also for better management of time and space the reads with coverage more than 140 are split into individual reads of coverage size 70. This is one of the practical approaches that help HiTEC reaching the highest accuracy in less time with less space than any other error correction method. Another important aspect of the new implementation is that time is saved by reusing the witness length. In the original implementation, witnesses are changed and used for correction in each iteration. If we keep the same length then we do not have to recount the witnesses. Noting that this is not possible in the original implementation, due to the use of suffix array which cannot be dynamically updated.

**Algorithm 2** Encoding_witness ($binArray[n], witLenInBits, witness[\,]$ *and* $counters[\,][\,]$)

```
1: for $i = 0$ to $n$ do
2:   $currRead[\,] \leftarrow$ one full read
3:   $currRead\_revcomp[\,] \leftarrow$ reverse complement of $currRead[\,]$ {use Algorithm 3}
4:   $currReadShare \leftarrow 0$
5:   $currReadShare\_revcomp \leftarrow 0$
6:   $i \leftarrow i + readLenInBytes$
7:   Load_CurrReadShare($currRead[\,], currRead\_revcomp[\,], readLenInBytes$)
8:   for $j = 0 \rightarrow readLenInBits - witLenInBits$ do
9:     if $j \neq 0$ then
10:      $after \leftarrow currReadShare \,\&\, 3$
11:      $currReadShare \leftarrow currReadShare \gg 2$
12:    end if
13:    if $j \neq readLenInBits - witLenInBits$ then
14:      $before \leftarrow (currReadShare \gg witLenInBits) \,\&\, 3$
15:    end if
16:    $j \leftarrow j + 2$
17:    $witness \leftarrow currReadShare \,\&\, witnessMask$
18:    $witness\_revcomp \leftarrow (currReadShare\_revcomp \,\&\, (witnessMask \ll 64 - witLenInBits)) \gg (64 - witLenInBits)$
19:    $witnessMinValue \leftarrow (witness < witness\_revcomp)\,?\,witness : witness\_revcomp$
20:    Witness_Hash_Value($WitnessMinValue, witness[\,]$, TABLE_SPACE)
21:    Witness_Counters($witnessHashVal, counters[\,][\,], after, before$)
22:    $currReadShare\_revcomp \leftarrow currReadShare\_revcomp \ll 2$
23:    if $j \,\%\, 8 = 0$ AND $j \neq 0$ then
24:      Load_Extra_Byte($currReadShare, currReadShare\_revcomp, readLenInBytes, numOfBytesRead$)
25:    end if
26:  end for
27: end for
```

**Algorithm 3** All_Reverse_Complement_Of8bit

```
1: $tempRevComp \leftarrow 0$
2: $complementOf\_i \leftarrow 0$
3: for $i = 0 \rightarrow 255$ do
4:   $allRevCompOf8Bit[i] \leftarrow 0$
5:   $complementOf\_i \leftarrow \sim i$
6:   $tempRevComp \leftarrow 3 \,\&\, complementOf\_i$
7:   $allRevCompOf8Bit[i] \leftarrow allRevCompOf8Bit[i] \mid (tempRevComp \ll 6)$
8:   $complementOf\_i \leftarrow complementOf\_i \gg 2$
9:   $tempRevComp \leftarrow 3 \,\&\, complementOf\_i$
10:  $allRevCompOf8Bit[i] \leftarrow allRevCompOf8Bit[i] \mid (tempRevComp \ll 4)$
11:  $complementOf\_i \leftarrow complementOf_i \gg 2$
12:  $tempRevComp \leftarrow 3 \,\&\, complementOf\_i$
13:  $allRevCompOf8Bit[i] \leftarrow allRevCompOf8Bit[i] \mid (tempRevComp \ll 2)$
14:  $complementOf\_i \leftarrow complementOf\_i \gg 2$
15:  $tempRevComp \leftarrow 3 \,\&\, complementOf\_i$
16:  $allRevCompOf8Bit[i] \leftarrow allRevCompOf8Bit[i] \mid tempRevComp$
17:  $i \leftarrow i + 1$
18: end for
19: Return $allRevCompOf8Bit[\,]$
```

**Algorithm 4** Load_CurrReadShare ($currRead[\,], currRead\_revcomp[\,], readLenInBytes$)

```
1: $counterForRead \leftarrow readLenInBytes$
2: $counterForRead\_RevComp \leftarrow readLenInBytes$
3: for $k = 0 \rightarrow 8\& < readLenInBytes$ do
4:   $tempReadShare \leftarrow currRead[counterForRead]$
5:   $tempReadShare\_RevComp \leftarrow currRead_RevComp[counterForRead\_RevComp]$
6:   $tempReadShare \leftarrow tempReadShare \ll 8 \times k$
7:   $tempReadShare\_RevComp \leftarrow tempReadShare\_RevComp \ll (56 - 8 \times k)$
8:   $currReadShare \leftarrow currReadShare \mid tempReadShare$
9:   $currReadShare\_RevComp \leftarrow currReadShare\_RevComp \mid tempReadShare\_RevComp$
10:  $counterForRead \leftarrow counterForRead - 1$
11:  $counterForRead\_RevComp \leftarrow counterForRead\_RevComp + 1$
12:  $k \leftarrow k + 1$
13: end for
```

**Algorithm 5** Witness_Hash_Value ($WitnessMinValue, witness[\,]$,TABLE_SPACE)

```
1: $WitnessHashValue \leftarrow witnessMinValue\%$ TABLE_SPACE
2: while ($witness[witnessHashVal] \neq 0$) AND ($witness[witnessHashVal] \neq witnessMinValue$) do
3:   if $witnessHashVal <$ TABLE_SPACE then
4:     $witnessHashVal \leftarrow witnessHashVal + 1$
5:   else
6:     $witnessHashVal \leftarrow 0$
7:   end if
8: end while
9: $witness[witnessHashVal] \leftarrow witnessMinValue$
```

**Algorithm 6** Witness_Counters ($witnessHashValue, witness, after, before$)

```
1: if $witnessHashValue \neq witness$ then
2:   $after \leftarrow (\sim before)\ \&\ 3$
3:   $before \leftarrow (\sim after)\ \&\ 3$
4: end if
5: if $counter[witnessHashVal][before + 3] < 255$ then
6:   $counter[witnessHashVal][after] \leftarrow counter[witnessHashVal][before + 3] + 1$
7: end if
8: if $counter[witnessHashVal][after] < 255$ then
9:   $counter[witnessHashVal][after] \leftarrow counter[witnessHashVal][after] + 1$
10: end if
```

**Algorithm 7** Load_Extra_Byte ($currReadShare, currReadShare\_revcomp, readLenInBytes, numOfByte$

```
1: $extraRead \leftarrow extraByte$
2: $extraRead\_RevComp \leftarrow extraByte\_RevComp$
3: if $numOfBytesRead \leq readLenInBytes$ then
4:   $extraRead \leftarrow extraRead \ll 56$
5:   $currReadShare\_RevComp \leftarrow currReadShare\_RevComp \mid extraRead\_RevComp$
6:   $currReadShare \leftarrow currReadShare \mid extraRead$
7:   $numOfBytesRead \leftarrow numOfBytesRead + 1$
8: end if
```

**Algorithm 8** Correct_Errors (*binReads*[ ], *witness*[ ], *counter*[ ][ ], *c*)

```
1: for $i = 0$ to $n$ do
2:   $currRead[\,] \leftarrow$ one full read
3:   $currRead\_revcomp[\,] \leftarrow$ reverse complement of $currRead[\,]$ {use Algorithm 3}
4:   $currReadShare \leftarrow 0$
5:   $currReadShare\_revcomp \leftarrow 0$
6:   $i \leftarrow i + readLenInBytes$
7:   Load_CurrReadShare($currRead[\,], currRead\_revcomp[\,], readLenInBytes$)
8:   for $j = 0 \rightarrow readLenInBits - witLenInBits$ do
9:     if $j \neq 0$ then
10:      $after \leftarrow currReadShare \;\&\; 3$
11:      $currReadShare \leftarrow currReadShare \ll 2$
12:    end if
13:    if $j \neq readLenInBits - witLenInBits$ then
14:      $before \leftarrow (currReadShare \gg witLenInBits) \;\&\; 3$
15:    end if
16:    $j \leftarrow j + 2$
17:    $witness \leftarrow currReadShare \;\&\; witnessMask$
18:    $witness\_revcomp \leftarrow (currReadShare\_revcomp \;\&\; (witnessMask \ll 64 - witLenInBits)) \gg (64 - witLenInBits)$
19:    $witnessMinValue \leftarrow (witness < witness\_revcomp)\;?\;witness : witness\_revcomp$
20:    $witnessHashVal \leftarrow$ Get_Witness_Hash_Value($WitnessMinValue, witness[]$,TABLE_SPACE)
21:    Base_Correction($witnessHashVal, after, before, c$)
22:    $currReadShare\_revcomp \leftarrow currReadShare\_revcomp \gg 2$
23:    if ($j \,\%\, 8 = 0$ $AND$ $j \neq 0$ then
24:      Load_Extra_Byte($currReadShare, currReadShare\_revcomp, readLenInBytes, numOfBytesRead$)
25:    end if
26:  end for
27: end for
```

**Algorithm 9** Base_Correction ($witnessHashVal, after, before, c$)

```
1: $corrected \leftarrow 0$
2: $after\_count \leftarrow counters[witnessHashVal][after]$
3: for $j = 0 \rightarrow 4$ do
4:     if $counters[witnessHashVal][j] > after\_count$ AND $counters[witnessHashVal][j] > T$ then
5:         $before \leftarrow j$
6:         $before\_count \leftarrow counters[witnessHashVal][j]$
7:         $corrected \leftarrow 1$
8:         $j \leftarrow j + 1$
9:     end if
10: end for
11: if $corrected = 1$ then
12:     Correct the base value with the correct value($j$) in both $currReadShare$ and $currReadShare\_revcomp$
13:     $c \leftarrow c + 1$
14: end if
15: $corrected \leftarrow 0$
16: $before\_count \leftarrow counters[witnessHashVal][before + 4]$
17: for $j = 4 \rightarrow 8$ do
18:     if $counters[witnessHashVal][j] > before\_count$ AND $counters[witnessHashVal][j] > T$ then
19:         $before \leftarrow j - 4$
20:         $before\_count \leftarrow counters[witnessHashVal][j]$
21:         $corrected \leftarrow 1$
22:         $j \leftarrow j + 1$
23:     end if
24: end for
25: if $corrected = 1$ then
26:     Correct the base value with the correct value($j - 4$) in both $currReadShare$ and $currReadShare\_revcomp$
27:     $c \leftarrow c + 1$
28: end if
```

**Algorithm 10** HiTEC2

```
given: $n\ reads\ r_1, r_2, ...., r_n\ (of\ length\ l\ each), L\ and\ p$
return: $Corrected\ reads\ in\ the\ same\ input\ format$
 1: compute $w_M$ and $w_m$
 2: compute $T$
 3: $witnessMask \leftarrow 2^{witLenInBits+1} - 1$
 4: for $i = 0 \rightarrow$ TABLE_SPACE do
 5:     $witness[i] \leftarrow 0$ , $i \leftarrow i + 1$
 6:     for $j = 0 \rightarrow 8$ do
 7:         $counter[i][j] \leftarrow 0, j \leftarrow j + 1$
 8:     end for
 9: end for
10: $i \leftarrow 1$
11: $w \leftarrow w_m$
12: Construct $binReads$[ ] from the original input reads
13: Choose appropriate TABLE_SPACE from the list of prime numbers
14: Encode_Witness($binReads, w \times 2$)
15: repeat
16:     $c \leftarrow 0$
17:     Correct_Errors($binReads$[ ], $witness$[ ], $counters$[ ][ ], $c$)
18:     $i \leftarrow i + 1$
19: until $((\frac{c}{ln} < 0.0001)\ or\ (i > 9))$
20: Fill the $charReads$[ ] by decoding $binReads$[ ]
21: return $charReads$[ ]
```

# Chapter 5

# Experiments

We present in this chapter several experiments using the simulated and real data sets in order to compare the new implementation, called HiTEC2 with the original one, HiTEC. The comparison is done with respect to all aspects, accuracy, time and space. The *accuracy* is defined as the ratio between the number of corrected reads and the number of initially erroneous reads. If $err_{bef}$ is the number of erroneous reads before and $err_{aft}$ is the number of errors after. This can also be represented in terms of TP, TN, FP, FN(true/false positive/negative): $err_{bef}$ = TP+FN and $err_{aft}$ = FP+FN. Then the accuracy is the ratio of

$$accuracy = \frac{err_{bef}-err_{aft}}{err_{bef}} = \frac{TP-FP}{TP+FN}$$

## 5.1 Accuracy

We have compared the accuracy of HiTEC2 with HiTEC on several data sets consider in [IFI10]. Table 5.1 lists several bacterial genomes downloaded from GenBank under the accession numbers specified. The IDs mentioned in the parenthesis in Table 5.1 of the bacterial genomes are referred in the successive table. The data sets were generated by uniformly sampling the reads with given coverage, length and per base error rate from the genomes listed in Table 5.1. Table 5.4 contains data sets with different coverage and read lengths taken from the longest genome considered. The first real data set was used also by [Her08] and [SSP+09]; it is available from *www.genomic.ch/edena.php*. Both the first and the second real data sets were used by [Doh07] and [SSLW10]. The second data set is available from *sharcgs.molgen.mpg.de/download.shtml*. The third one is available from *clcbio.com/index.php?id=1290*, the CLCbio web site, as an example of NGS data. Table 5.5 contains several real data sets of Illumina reads. The per base error rate taken from [IFI10] is calculated by counting the number of mismatches from the output file of the RMAP.

| Reference genome (ID) | Accession no. | Len.(bp) |
|---|---|---|
| Saccharomyces Cerevisiae, Chr. 5 (S.cer5) | NC_001137 | 576,869 |
| Saccharomyces Cerevisiae, Chr. 7 (S.cer7) | NC_001139 | 1,090,946 |
| Haemophilus Influenzae (H.inf) | NC_007146 | 1,914,490 |
| Escherichia coli str.K-12 substr.MG1655 (E.coli) | NC_000913 | 4,639,675 |
| Escherichia coli str.K-12 substr.DH10B (E.coli2) | NC_010473 | 4,686,137 |
| Staphylococcus aureus (S.aureus) | NC_003923 | 2,820,462 |
| Helicobacter acinonychis (H.acinonychis) | NC_008229 | 1,553,927 |

Table 5.1: The genomes used for comparison.

Even if the idea of correcting errors of HiTEC2 is inherited from HiTEC, it is applied somewhat differently, aiming for a significant reduction in time. Since the witness length doesn't change in HiTEC2, the results can be different. To have a fair comparison both programs were ran for same number of iterations. Table 5.2, 5.3, 5.4 contains results for the simulated data and Table 5.5 for real data. The conclusion we can derive from these results is, HiTEC2 has the same accuracy that of HiTEC. HiTEC2, though the technique of error correction is different from HiTEC, some of the merits of HiTEC still holds good for HiTEC2. Such as the accuracy for the simulated data is not affected by the change in coverage or error rate and both perform better for low coverage data sets compared to other techniques. One more important thing to note here is both HiTEC and HiTEC2 was ran for a maximum of 6 iterations. HiTEC2 unlike HiTEC uses only one witness length for all iterations rather than using different witness length for each iteration.The conclusion we can arrive after comparing the HiTEC and HiTEC2 in terms of accuracy is that HiTEC2 performs with similar or better accuracy.

## 5.2 Time and Space

The novelty is provided mainly in Table 5.6, 5.7, 5.8 where a significant decrease in the space and time complexities of HiTEC2 as seen compared to HiTEC. This is expected due to the hash table and smart use of binary operations. The results in Table 5.6, 5.7, 5.8 were performed for the data sets in Table 5.1 on SHARCNET ORCA cluster high performance computers: *www.sharcnet.ca*. The each node of the ORCA cluster has AMD Opteron Processors, Model 6174 (Magny-Cours: 2.2 GHz, 12MB Level 3 Cache, 80 W), 32 GB of RAM and 160 GB of local storage. HiTEC has the lowest space complexities compared to other techniques [IFI10], but HiTEC2 significantly surpasses HiTEC in both space and time complexities.

| Data set | | Accuracy | |
|---|---|---|---|
| Genome | err.(%) | HiTEC | HiTEC2 |
| S.cer5 | 1 | 99.79 | 99.58 |
| S.cer7 | 1 | 99.74 | 99.60 |
| H.inf | 1 | 99.73 | 99.55 |
| E.coli | 1 | 99.22 | 99.58 |

Table 5.2: Accuracy comparison for the data sets of [SSP+09]. The read length is 70bp and coverage is 70 for all data sets.

| Data set | | Accuracy | |
|---|---|---|---|
| Genome | err.(%) | HiTEC | HiTEC2 |
| S.cer5 | 1 | 96.27 | 97.02 |
| S.cer7 | 1 | 95.76 | 96.63 |
| H.inf | 1 | 96.39 | 95.57 |
| E.coli | 1 | 94.41 | 95.663 |

Table 5.3: Accuracy comparison for the data sets of [SSLW10]. The read length is 35bp and coverage is 70 for all data sets.

| Data set | | | | Accuracy | |
|---|---|---|---|---|---|
| Genome | read len. | covrg. | err.(%) | HiTEC | HiTEC2 |
| E.coli | 70 | 35 | 1 | 99.25 | 99.58 |
| E.coli | 50 | 50 | 1 | 97.88 | 99.04 |
| E.coli | 50 | 35 | 1 | 97.91 | 99.03 |
| E.coli | 35 | 50 | 1 | 91.10 | 95.63 |

Table 5.4: Accuracy comparison between HiTEC and HiTEC2 for a variety of read lengths, coverage levels, and error rates sampled from the E.coli genome.

| Data set | | | | Accuracy | |
|---|---|---|---|---|---|
| Genome | read len. | covrg. | err.(%) | HiTEC | HiTEC2 |
| S.aureus | 35 | 42.5 | 1.00 | 90.23 | 94.87 |
| H.acinonychis | 36 | 94 | 1.60 | 89.15 | 91.66 |
| E.coli2 | 35 | 17.8 | 0.38 | 73.185 | 86.94 |
| E.coli | 100 | 86 | 0.50 | 77.62 | 80.60 |

Table 5.5: Accuracy comparison for several real sets of Illumina reads.

| Data set | | Space (MB) | |
|---|---|---|---|
| Genome | err.(%) | HiTEC | HiTEC2 |
| S.cer5 | 1 | 778 | 253 |
| S.cer7 | 1 | 1471 | 505 |
| H.inf | 1 | 2582 | 805 |
| E.coli | 1 | 7200 | 1710 |
| S.aureus | 1 | 2100 | 415 |
| E.coli 2 | 1 | 1400 | 322 |
| H.acinonychis | 1 | 2600 | 607 |

Table 5.6: Space comparison between HiTEC and HiTEC2. The read length is 70bp and coverage is 70 for all data sets.

| Data set | | | | Time(seconds) | |
|---|---|---|---|---|---|
| Genome | read len. | covrg. | err.(%) | HiTEC | HiTEC2 |
| S.aureus | 35 | 42.5 | 1.00 | 1114 | 145 |
| H.acinonychis | 36 | 94 | 1.60 | 1370 | 282 |
| E.coli2 | 35 | 17.8 | 0.38 | 720 | 138 |
| E.coli | 100 | 86 | 0.50 | 4840 | 1128 |

Table 5.7: Time comparison between HiTEC and HiTEC2 for several real sets of Illumina reads.

| Data set | | | | Time(seconds) | |
|---|---|---|---|---|---|
| Genome | read len. | covrg. | err.(%) | HiTEC | HiTEC2 |
| S.cer5 | 70 | 70 | 1 | 377 | 72 |
| S.cer5 | 70 | 70 | 1 | 754 | 133 |
| H.inf | 70 | 70 | 1 | 1395 | 273 |
| E.coli | 70 | 70 | 1 | 3762 | 564 |
| S.cer5 | 35 | 70 | 1 | 258 | 71 |
| S.cer5 | 35 | 70 | 1 | 666 | 109 |
| H.inf | 35 | 70 | 1 | 1177 | 302 |
| E.coli | 35 | 70 | 1 | 3231 | 1408 |
| E.coli | 70 | 35 | 1 | 1793 | 255 |
| E.coli | 50 | 35 | 1 | 1848 | 335 |
| E.coli | 50 | 50 | 1 | 2467 | 358 |
| E.coli | 35 | 50 | 1 | 2199 | 465 |

Table 5.8: Time comparison between HiTEC and HiTEC2 for a variety of read lengths, coverage levels, and error rates for various genomes.

# Chapter 6

# Conclusion and Future Research

The main goal of this thesis is to provide a better version of HiTEC in terms of time and space complexity without the cost of accuracy for correcting errors of high throughput sequencing technologies. The extensive experiments we have conducted shows that our algorithm has significantly improved the time and space complexities and the capability of algorithm to adjust with the input data remain the same as HiTEC. The results published in the experiments section with respect to real data sets are from Illmina and the algorithm is expected to behave similarly for any type of reads for which the errors consist mainly of substitutions.

Our algorithm's accuracy will increase with the increase in the reads lengths and, according to [Xia10], the read length is going to grow with the 3rd generation of sequencing technologies, such as single molecule sequencing or nanopore sequencing. So we hope our program will be competitive for future changes in the sequencing technologies. While we think of some features to add to HiTEC2, the immediate one we can think of is parallel implementation. This feature would help the program to handle more massive data output. Another feature we think of adding to HiTEC2 is efficient use of quality scores in improving the accuracy of the error correction.

# Bibliography

[And81] S Anderson. Shotgun dna sequencing using cloned dnase i-generated fragments. *Nucleic Acids Research*, 9(13):3015–3042, 1981.

[Blo70] B Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun, ACM*, 13:422–426, 1970.

[CP09] M Chaisson and P Pevzner. A short-read fragment assembly of bacterial genomes. *Bioinformatics*, 18:324–330, 2009.

[Doh07] J Dohm. Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res*, 17:1697—-1706, 2007.

[EE48] V Ernst and C Erwin. The separation and quantitative estimation of purines and pyrimidines in minute amounts. *Biol. Chem*, 176:703–712, 1948.

[Eve07] T Everson. *The gene: a historical perspective*. Greenwood Publishing Group, 88 Post Road West, P.O. Box 5007, Westport, CT 06881-5007, 2007.

[FG48] R E Franklin and R G Gosling. Molecular configuration in sodium thymonucleate. *Nature*, 171:740–741, 1948.

[FPM+08] Y Fu, H Peckham, S McLaughlin, J Ni, M Rhodes, J Malek, K McKernan, and A Blanchard. Solid sequencing and 2-base encoding. *Applied Biosystems*, 2008.

[Hen01] R M Henig. *The Monk in the Garden : The Lost and Found Genius of Gregor Mendel, The Father of Genetics*. Houghton Mifflin, 222 Berkeley St number 11, Boston, MA 02116-3760, United States, 2001.

[Her08] D Hernandez. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Res*, 18:802–809, 2008.

[IFI10] L Ilie, F Fazayeli, and S Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2010.

[Jos85] L Joshua. The transforming principle: Discovering that genes are made of dna by maclyn mccarty. *Genetics*, 64:2–3,173–178, 1985.

[Kim05] D K Kim. Constructing suffix arrays in linear time. *Discrete Algorithms*, 3(2-4):126—-142, 2005.

[Knu88] D Knuth. *The Art of Computer Programming*, volume 3. Pearson Education, Inc., San Francisco, 1988.

[KS03] J Karkkainen and P Sander. Simple linear work suffix array construction, in proc. of icalp'03. *Lecture Notes in Comput. Sci. 2719, Springer-Verlag, Berlin, Heidelberg*, pages 943—-955, 2003.

[Mar08] E Mardis. Next-generation dna sequencing methods. *Annu Rev Genomics Hum Genet*, 9:387–402, 2008.

[Mey00] B Meyer. A constraint-based framework for diagrammatic reasoning. *Applied Artificial Intelligence*, 14:327–344, 2000.

[MKP+96] R Mostafa, S Karamohamed, B Pettersson, U Mathias, and N Pal. Real-time dna sequencing using detection of pyrophosphate release. *Analytical Biochemistry*, 242(1):84, 1996.

[MM93] U Manber and G Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935—-948, 1993.

[MMP98] R Mostafa, U Mathias, and N Pal. A sequencing method based on real-time pyrophosphate. *Science*, 281(5375):363, 1998.

[Pal07] N Pal. The history of pyrosequencing. *Methods Mol Biology*, 373:1–14, 2007.

[Rus02] P Russell. *iGenetics*. Pearson Education, Inc., San Francisco, 2002.

[Rus11] N Rusk. Torrents of sequence. *Nat Meth*, 8(1):44–88, 2011.

[San80] F Sanger. Nobel lecture: Determination of nucleotide sequences in dna. *Nobelprize.org, retrieved*, 1980.

[SSLW10] H Shi, B Shcmidt, W Liu, and W Wittig. A parallel algorithm for error correction in high-throughput short-read data on cuda-enabled graphics hardware. *Jouranal of Computational Biology*, 17(4):601–617, 2010.

[SSP$^+$09] J Schroder, H Schroder, S Puglisi, R Sinha, and B Schmidt. Shrec: a short-read error correction method. *Genome*, 25(17):2157–2162, 2009.

[Sta79] R Staden. A strategy of dna sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601–2611, 1979.

[TLA90] A Tenenbaum, Y Langsam, and M Augestein. *Data Structures Using C*, volume 26(20). Prentice Hall, Inc., A Pearson Education Company, Upper Saddle River, New Jersey 07458, 1990.

[Tor01] K Toru. Linear-time longest-common-prefix computation in suffix arrays and its applications. *Proc. of CPM'01, Lecture Notes in Comput. Sci. 2089, Springer-Verlag, Berlin*, pages 181—-192, 2001.

[Ukk95] E Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249—-260, 1995.

[WC53] J Watson and F H Crick. Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid. *Nature*, 171(4356):737—-738, 1953.

[Wei73] P Weiner. Linear pattern matching algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1—-11, 1973.

[Xia10] Z XiaoGuang. The next-generation sequencing technology: A technology review and future perspective. *Science China*, 53:44—-57, 2010.

[YDA10] X Yang, K Dorman, and S Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics*, 26(20):2526–2533, 2010.

# Appendix A

# Appendix

The list of primes used in HiTEC2 as the hash table size.

| | | | | |
|---|---|---|---|---|
| 1114523 | 1180043 | 1245227 | 1310759 | 1376447 |
| 1442087 | 1507379 | 1573667 | 1638899 | 1704023 |
| 1769627 | 1835027 | 1900667 | 1966127 | 2031839 |
| 2228483 | 2359559 | 2490707 | 2621447 | 2752679 |
| 2883767 | 3015527 | 3145739 | 3277283 | 3408323 |
| 3539267 | 3670259 | 3801143 | 3932483 | 4063559 |
| 4456643 | 4718699 | 4980827 | 5243003 | 5505239 |
| 5767187 | 6029603 | 6291563 | 6553979 | 6816527 |
| 7079159 | 7340639 | 7602359 | 7864799 | 8126747 |
| 8913119 | 9437399 | 9962207 | 10485767 | 11010383 |
| 11534819 | 12059123 | 12583007 | 13107923 | 13631819 |
| 14156543 | 14680067 | 15204467 | 15729647 | 16253423 |
| 17825999 | 18874379 | 19923227 | 20971799 | 22020227 |
| 23069447 | 24117683 | 25166423 | 26214743 | 27264047 |
| 28312007 | 29360147 | 30410483 | 31457627 | 32505983 |
| 35651783 | 37749983 | 39845987 | 41943347 | 44040383 |

| | | | | |
|---|---|---|---|---|
| 46137887 | 48234623 | 50331707 | 52429067 | 54526019 |
| 56623367 | 58720307 | 60817763 | 62915459 | 65012279 |
| 71303567 | 75497999 | 79691867 | 83886983 | 88080527 |
| 92275307 | 96470447 | 100663439 | 104858387 | 109052183 |
| 113246699 | 117440699 | 121635467 | 125829239 | 130023683 |
| 142606379 | 150994979 | 159383759 | 167772239 | 176160779 |
| 184549559 | 192938003 | 201327359 | 209715719 | 218104427 |
| 226493747 | 234882239 | 243269639 | 251659139 | 260047367 |
| 285215507 | 301989959 | 318767927 | 335544323 | 352321643 |
| 369100463 | 385876703 | 402654059 | 419432243 | 436208447 |
| 452986103 | 469762067 | 486539519 | 503316623 | 520094747 |
| 570425399 | 603979919 | 637534763 | 671089283 | 704643287 |
| 738198347 | 771752363 | 805307963 | 838861103 | 872415239 |
| 905971007 | 939525143 | 973079279 | 1006633283 | 1040187419 |
| 1140852767 | 1207960679 | 1275069143 | 1342177379 | 1409288183 |
| 1476395699 | 1543504343 | 1610613119 | 1677721667 | 1744830587 |
| 1811940419 | 1879049087 | 1946157419 | 2013265967 | 2080375127 |
| 2281701827 | 2415920939 | 2550137039 | 2684355383 | 2818572539 |
| 2952791147 | 3087008663 | 3221226167 | 3355444187 | 3489661079 |
| 3623878823 | 3758096939 | 3892314659 | 4026532187 | 4160749883 |
| 4563403379 | 4831838783 | 5100273923 | 5368709219 | 5637144743 |
| 5905580687 | 6174015503 | 6442452119 | 6710886467 | 6979322123 |
| 7247758307 | 7516193123 | 7784629079 | 8053065599 | 8321499203 |
| 9126806147 | 9663676523 | 10200548819 | 10737418883 | 11274289319 |
| 11811160139 | 12348031523 | 12884902223 | 13421772839 | 13958645543 |
| 14495515943 | 15032386163 | 15569257247 | 16106127887 | 16642998803 |
| 18253612127 | 19327353083 | 20401094843 | 21474837719 | 22548578579 |

| | | | | |
|---|---|---|---|---|
| 23622320927 | 24696062387 | 25769803799 | 26843546243 | 27917287907 |
| 28991030759 | 30064772327 | 31138513067 | 32212254947 | 33285996803 |
| 36507222923 | 38654706323 | 40802189423 | 42949673423 | 45097157927 |
| 47244640319 | 49392124247 | 51539607599 | 53687092307 | 55834576979 |
| 57982058579 | 60129542339 | 62277026327 | 64424509847 | 66571993199 |
| 73014444299 | 77309412407 | 81604379243 | 85899346727 | 90194314103 |
| 94489281203 | 98784255863 | 103079215439 | 107374183703 | 111669150239 |
| 115964117999 | 120259085183 | 124554051983 | 128849019059 | 133143986399 |
| 146028888179 | 154618823603 | 163208757527 | 171798693719 | 180388628579 |
| 188978561207 | 197568495647 | 206158430447 | 214748365067 | 223338303719 |
| 231928234787 | 240518168603 | 249108103547 | 257698038539 | 266287975727 |
| 292057776239 | 309237645803 | 326417515547 | 343597385507 | 360777253763 |
| 377957124803 | 395136991499 | 412316861267 | 429496730879 | 446676599987 |
| 463856468987 | 481036337207 | 498216206387 | 515396078039 | 532575944723 |
| 584115552323 | 618475290887 | 652835029643 | 687194768879 | 721554506879 |
| 755914244627 | 790273985219 | 824633721383 | 858993459587 | 893353198763 |
| 927712936643 | 962072674643 | 996432414899 | 1030792152539 | 1065151889507 |
| 1168231105859 | 1236950582039 | 1305670059983 | 1374389535587 | 1443109012607 |
| 1511828491883 | 1580547965639 | 1649267441747 | 1717986918839 | 1786706397767 |
| 1855425872459 | 1924145348627 | 1992864827099 | 2061584304323 | 2130303780503 |
| 2336462210183 | 2473901164367 | 2611340118887 | 2748779070239 | 2886218024939 |
| 3023656976507 | 3161095931639 | 3298534883999 | 3435973836983 | 3573412791647 |
| 3710851743923 | 3848290698467 | 3985729653707 | 4123168604483 | 4260607557707 |
| 4672924419707 | 4947802331663 | 5222680234139 | 5497558138979 | 5772436047947 |
| 6047313952943 | 6322191860339 | 6597069767699 | 6871947674003 | 7146825580703 |
| 7421703488567 | 7696581395627 | 7971459304163 | 8246337210659 | 8521215117407 |
| 9345848837267 | 9895604651243 | 10445360463947 | 10995116279639 | 11544872100683 |

| | | | |
|---|---|---|---|
| 12094627906847 | 12644383722779 | 13194139536659 | 13743895350023 |
| 14293651161443 | 14843406975659 | 15393162789503 | 1594291860434 |
| 16492674420863 | 17042430234443 | 18691697672867 | 19791209300867 |
| 20890720927823 | 21990232555703 | 23089744183799 | 24189255814847 |
| 25288767440099 | 26388279068903 | 27487790694887 | 28587302323787 |
| 29686813951463 | 30786325577867 | 31885837205567 | 32985348833687 |
| 34084860462083 | 37383395344739 | 39582418600883 | 41781441856823 |
| 43980465111383 | 46179488367203 | 48378511622303 | 50577534878987 |
| 52776558134423 | 54975581392583 | 57174604644503 | 59373627900407 |
| 61572651156383 | 63771674412287 | 65970697666967 | 68169720924167 |
| 74766790688867 | 79164837200927 | 83562883712027 | 87960930223163 |
| 92358976733483 | 96757023247427 | 101155069756823 | 105553116266999 |
| 109951162779203 | 114349209290003 | 118747255800179 | 123145302311783 |
| 127543348823027 | 131941395333479 | 136339441846019 | 149533581378263 |
| 158329674402959 | 167125767424739 | 175921860444599 | 184717953466703 |
| 193514046490343 | 202310139514283 | 211106232536699 | 219902325558107 |
| 228698418578879 | 237494511600287 | 246290604623279 | 255086697645023 |
| 263882790666959 | 272678883689987 | 299067162755363 | 316659348799919 |
| 334251534845303 | 351843720890723 | 369435906934019 | 387028092977819 |
| 404620279022447 | 422212465067447 | 439804651111103 | 457396837157483 |
| 474989023199423 | 492581209246163 | 510173395291199 | 527765581341227 |
| 545357767379483 | 598134325510343 | 633318697599023 | 668503069688723 |
| 703687441776707 | 738871813866287 | 774056185954967 | 809240558043419 |
| 844424930134187 | 879609302222207 | 914793674313899 | 949978046398607 |
| 985162418489267 | 1020346790579903 | 1055531162666507 | 1090715534754863 |