Shahjalal University of Science and Technology
Department of Computing Science and Engineering

NanoMapper: An Efficient Read Mapping Algorithm Using Gapped Minimizer and FM-indexing for Oxford Nanopore Long Noisy Reads

Enamul Hassan
Reg. No.: 2011331051
4^{th} year, 2^{nd} Semester

Md. Khairullah Gaurab
Reg. No.: 2011331063
4^{th} year, 2^{nd} Semester

Department of Computer Science and Engineering

Supervisor

Md. Eamin Rahman
Assistant Professor
Department of Computer Science and Engineering

29th October, 2016

NanoMapper: An Efficient Read Mapping Algorithm Using Gapped Minimizer and FM-indexing for Oxford Nanopore Long Noisy Reads

A Thesis submitted to the

Department of Computing Science and Engineering

Shahjalal University of Science and Technology

Sylhet - 3114, Bangladesh

in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

By

ENAMUL HASSAN

Reg. No.: 2011331051

4^{th} year, 2^{nd} Semester

MD. KHAIRULLAH GAURAB

Reg. No.: 2011331063

4^{th} year, 2^{nd} Semester

Department of Computer Science and Engineering

Supervisor

MD. EAMIN RAHMAN

Assistant Professor

Department of Computer Science and Engineering

29th October, 2016

Recommendation Letter from Thesis Supervisor

The thesis

entitled " NanoMapper: An Efficient Read Mapping Algorithm Using Gapped Minimizer and FM-indexing for Oxford Nanopore Long Noisy Reads"

submitted by the students

1. Enamul Hassan
2. Md. Khairullah Gaurab

is a record of research work carried out under my supervision and I, hereby, approve that the report be submitted in partial fulfillment of the requirements for the award of their Bachelor Degrees.

Signature of the Supervisor:

Name of the Supervisor: Md. Eamin Rahman

Date: 29th October, 2016

Certificate of Acceptance of the Thesis

The thesis

entitled "NanoMapper: An Efficient Read Mapping Algorithm Using Gapped Minimizer and FM-indexing for Oxford Nanopore Long Noisy Reads"

submitted by the students

1. Enamul Hassan
2. Md. Khairullah Gaurab

on 29th October, 2016

is, hereby, accepted as the partial fulfillment of the requirements for the award of their Bachelor Degrees.

Head of the Dept.	Chairman, Exam. Committee	Supervisor

Abstract

There are several modern technologies developed to make the DNA sequencing easier and cheaper. Among them, Oxford Nanopore Technology (ONT) produces the longest read sequences and the lengths of the reads are growing day by day. But the challenge of handling such long read sequences is, the standard of these is very low comparing to other technologies. As error does not come following any specific pattern, it is hard to trace them. Several approaches are taken to defend this challenge. Tools like NanoBLASTer, GraphMap, BLAST, LAST, BWA-SW, Bowtie are developed to align these sequences. Every tool faced some challenges, but facing the desired challenges introduce some new challenges on the other side. So, no one tool is perfect for every work. NanoBLASTer is a new aligner tool which does a trade off and perform better from different aspects. But the naive mapping portion of this tool lead to skip long insertion or deletion. To minimize this problem along with maximizing the efficiency, a new mapper tool named NanoMapper is developed by us. Mapping is a key part of alignment. If this part could make efficient, then all further steps would be efficient. In NanoMapper, gapped minimizer is introduced as the first step so that it reduce the options of K-mers in the read and in the second step, the result of the query is retrieved using BWT FM-index. The memory requirement of the FM-index is tremendously low. Mapper tool like minimap takes few times more memory to index the whole genome of a reference sequence. There are some other tools which deals with mismatches only and does not play with insertion or deletion. NanoBLASTer as well as those tools are the main target of NanoMapper. It is shown that, it would help the NanoBLASTer and similar tools to align better with long insertions and deletions. NanoMapper is run against the well-known *Escherichia coli (E. coli)* reference sequence and a randomly generated synthetic reference.

Keywords: Gapped Minimizer, Mapper, Noisy Read Sequence, Suffix Array, Next Generation Sequencing (NGS), Oxford Nanopore Technology (ONT), Suffix Tree, Suffix Trie, Gapped K-mer, Full-text Minute-space(FM)-index, Burrows-Wheeler Transform (BWT), Variant Calling.

Acknowledgements

We would like to thank the Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet 3114, Bangladesh, for supporting this research. We are very thankful to our honorable teacher Md. Ruhul Amin Shajib for his outstanding directions and providing erroneous reads as well as reference data of *E. Coli*.

We are also grateful to anonymous authors of previous works for their co-operation and support. We want to give an special thanks to the *JabRef Development Team* for supporting us to manage references with such an interesting tool called *JabRef* [**?**].

Dedication

We would like to dedicate our research to our parents. We are also grateful to anonymous authors of previous works for their co-operation and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

A well-formed combination of several billions of human cells is called human being. Several components shape a human. Among these components, nucleus is the most overwhelming and intense part.[**?**] As like nucleus in cell, chromosome is the most king like component in nucleus. In the vast majority of the chromosomes, there exists a long two-strand-nucleotide-chain which is called DNA helix. Each strand builds up by consecutive occurrence of only four types of nucleotide named Adenine(A), Guanine(G), Cytosine(C) and Thymine(T). These four nucleotides are often called base and each of them is represented as a one letter symbol given in bracket. A strand in DNA helix is the reverse complement of the other strand if it is seen by fixing a direction. The word *complement* means the compatible base of bonding of a base. The complement of Adenine, Guanine, Cytosine and Thymine are Thymine, Cytosine, Guanine and Adenine respectively. So, if one strand is known, then the other strand is also known. To represent a strand, a long string of A, T, G and C is used in the field of Bioinformatics. A sub-string of this long string is called gene, if the corresponding sequence of nucleotides is responsible for any characteristics of the cell owner[**?**].

Now-a-days, earth knows that for every characteristics of an animal, plant, bacteria or even virus, there is a responsible gene in the cell for that characteristics. Finding out this type of responsible gene needs huge amount of data sequences to study. But sequencing data was very expensive and rare in the time when data reads were collected through Sanger sequencing[**?**, **?**] or

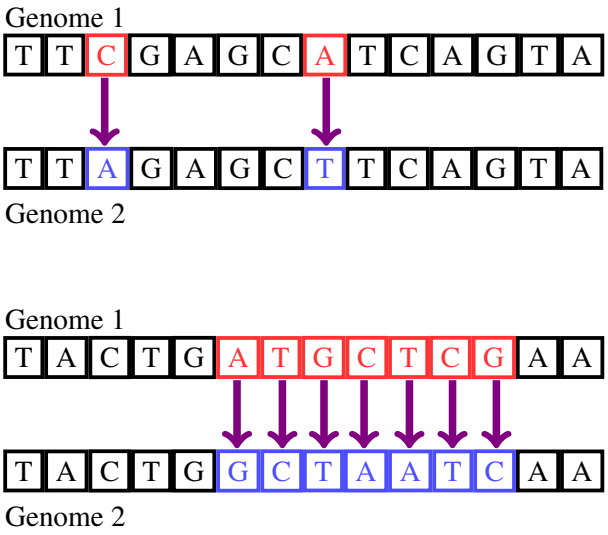

Figure 1.1: First one shows random mutations at two positions and Second one shows segment mutation.

first generation sequencing.

Next Generation Sequencing (NGS)[**?**] brought a tremendous revolution in the field of Bioinformatics. DNA sequencing was very expensive before this revolution. Extracting information from gene sequences become very cheap[**?**] and easy with the next generation sequencing technologies[**?**]. Raw data are kept as fasta or fastq[**?**] file format. The raw data file consist of short sequences, varying length from 30–60 base pair (bp)[**?**] to 3K–20K bp[1] each[**?**], taken from the random positions of the whole genome sequence. But the genome sequence of all members of a single species are not same, rather every individual carries different genome than others[**?**, **?**]. The difference between two genomes may be very few or may be very large. Based on this difference, some minor to major differences may be noticed in several things like behavior, skin, structure, color etc. Every individual is born with mutation in their 20K gene[**?**]. Figure 1.1 visualizes mutation clearly. However, identifying the mutations is a big issue for NGS data as the reads are short and repetitive as well as error occurs in reads. To retrieve the original genome sequence from these reads is a challenge. This challenge is faced by doing alignment[**?**] and keeping in a special format called SAM[**?**].

To avoid repetition, long DNA sequences are needed. Among the existing technologies, Oxford

[1]Reads could be more than 60K bp long in some cases. More information about read length could be found at: http://www.pacb.com/smrt-science/smrt-sequencing/read-lengths/

Nanopore Technology (ONT) generates extremely large sequences having no strong opponent. But the only problem is noise. The more data, the more noise. The length of the reads would increase day by day. As petabytes of sequencing data is adding in the databases daily[**?**], it would be in no use if the data could not be processed efficiently.

Alignment is basically done for pointing structural, functional, evolutionary similar portions[**?**]. An important part of alignment is mapping. An alignment software does mapping first, but the concepts are divided into two parts in the recent past in a sense that, if mapping could be efficient, then the next steps of the process would be efficient. That's why efficient mapping technique in terms of memory, time and placement is become a challenging task.

1.2 Motivation

Mapping as well as aligning has so many applications which makes this an important issue. The more we can map, the more we could find the gene specification and it's works. Below are some important issues discussed briefly.

1.2.1 Variant Calling

By definition, a variant call is an outcome that there is a nucleotide difference vs. some reference at any given particular position in an individual genome sequence or transcriptome.

Variant calling is a very significant procedure for WES or WGS sequencing and for some experiments also for RNA sequence. There are two major classes of variant [**?**]

1. Single Nucleotide variant (SNV)
2. Structural Variant

Application of those variant can be grouped in to three common scenarios for human geneticist using NGS data.[**?**]

1. Identification of causative genes in Mendelian disorders (germline mutations)
2. Identification of candidate genes in complex diseases
3. Identification of constitutional mutations as well as driver and passenger genes in cancer

1.2.1.1 Mendelian Disorder

This relates any disorders or phenotype complication that is resulted by hereditary genetic pass. This study design is only suitable for familial diseases where an proper amount of sample size if available for analysis.

1.2.1.2 Complex Diseases

Finding out candidate genes in complex diseases had been a challenging task for decades. As genetic research of complex phenotypes are based on either 'common disease-common variant' or 'common disease - rare variant' hypothesis[**?**], identification of candidate genes for both common variant and rare variant is important for the remedy and further study on any disease.

1.2.1.3 Somatic Mutations

Somatic mutation detection is very important for many cancer remedy. Even by detecting the mutations, tailored drug can be given to the affected patients tailored to the genetic makeup of their tumor or cancer cell.

Behind all these detections lies a common technique which require Alignment. Alignment is a process where read sequences are aligned against a reference genome sequence of any particular species. But any alignment first needed the reads to be mapped to the reference sequence. Means for each read there must be a position in the reference from where it was taken from.

For long reads which is prone to higher error rates like 15% to 20% and a average length of 10K base pare this read mapping becomes challenging actually. Our goal was to develop a mapping tool for long read which consume a feasible amount of time and gives a good mapping of the reads.

1.2.2 Gene Expression

Basically Gene expression is the physical method in which information from a gene is used in the synthesis of a operational gene product. These outcome products are mostly proteins, but in the non-protein coding genes such as transfer RNA (tRNA) or smaller nuclear RNA (snRNA) genes, the operational product is a functional RNA.

So for the greater understanding of genetic information measuring gene expression level in any particular species or in any particular individual is very important. For finding gene expression level one need to find the number of reads at different genes or target loci of the reference genome.These count of reads to each gene are then used to estimate expression levels.

There are many usage of gene expression in human genetics. Some of are listed below.

1. Classification of human tumors according to the gene level.[**?**]
2. Proper analysis and profiling of breast cancer.[**?**]
3. Ontological analysis for proper biological interpretation for gnomic data and results.[**?**]
4. Proper sub classification of cancer like Myeloid Leukemia .[**?**]

From the above list we can guess how important is to measure the gene expression level in any particular species or individual. And most of the commonly used gene expression measuring tools need to align the read sequences short/long to a particular reference genome sequence. And we know that alignment is dependent on how well the reads are mapped to the reference genome sequence. From the alignment information later count of reads mapped into a particular gene or target loci of the reference genome is the extracted to analyze and estimate the gene expression level for that particular individual.

So proper alignment technique is important for reads those are considered as long reads as well as long reads with higher error rates than short reads. Besides this for long reads with a higher error rates like 11% to 15%, mapping is more challenging. Our goal is to develop a more accurate mapping tool for long reads with higher error rates like PacBio[**?**] sequence reads.

1.2.3 DNA Binding

DNA-binding proteins are a specific type of proteins mainly composed of DNA-binding domains and thats why have a specific or general inclination for either single or double stranded DNA. That means in many cases those proteins are supposed to bind to a specific site or location of the DNA. This site of the DNA called a DNA-binding site. For example

In many cases locating these specific sites can be proven very important for may purposes. Like ChIP-sequencing[**?**] which used Solexa-sequencing[**?**] to find the locations of STAT1 sites

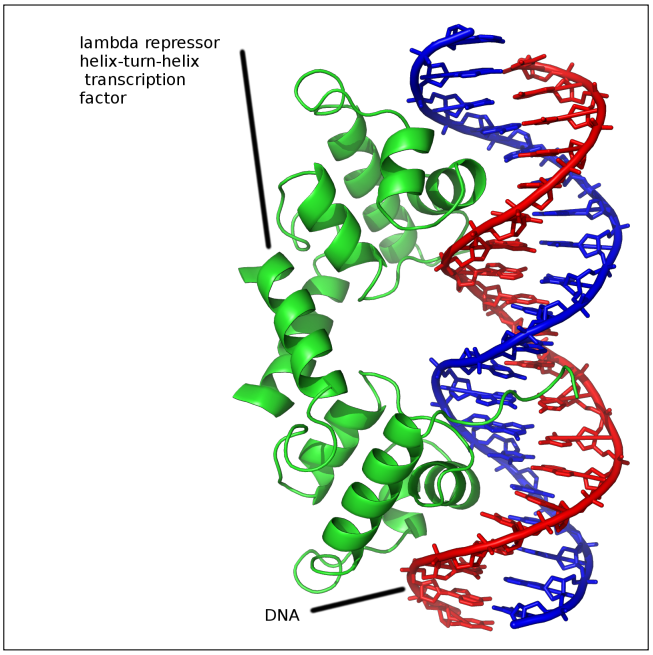

Figure 1.2: Picture of a lambda repressor helix-turn-helix transcription factor binding to its target location in DNA.

in HeLa S3 cells[**?**]. Mapping longs reads rather than short ones to a large genome sequence obviously presents a algorithmic challenge.

Applications where such mapping and resequencing is being done a primary concern always remains how accurately it is done. So our goal is develop a mapping tool that can map reads to the reference accurately and consuming a feasible amount of time.

Chapter 2

Background Study

2.1 Literature Review

After defining the problem specifically, the first thing is to have a good literature review to ensure that we are not going to waste our time by rediscovering anything. In this case, there are more than 95 mapper tools are developed till now according to on of the benchmarking[**?**]. But most of them are aligner. As it is stated before, aligner and mapper are separated in recent years. Before that these two words are used alternatively. On the other hand, all of them are not general purpose tools, rather their dedication goes to some categories like DNA, RNA, bi-sulfite, miRNA etc.[**?**]. Here, some tools are presented which are best matches with our task. Later in this chapter, some basics are covered to understand the work properly.

2.1.1 Minimap

Minimap[**?**] is a new mapper which is known for efficiently mapping ONT[**?**] and SMRT[**?**] reads. This mapper is claimed to be faster than the existing pipelines. Author of this paper also introduced a new mapping format called pairing mapping format(PAF)[**?**]. The primary purpose of the minimap is to act as a read overlapper but also it can be used as a read-to-genome and genome-to-genome mapper also.

Minimap is greatly influenced by the works of BLAST[**?**], BLAT[**?**], DALIGNER[**?**] and MHAP[**?**]. Minimap used the basic idea of sketch[**?**] like MHAP but rather than using fully sketch it uses minimizers as a compact representation. It also stores $\boldsymbol{k}$-mers in a hash table as used in

BLAT and MHAP as well as it uses sorting significantly like DALIGNER.

Minimap can map in four different mode. They are

1. Map two list of long sequences
2. All-vs-All read self mapping (That is later used for miniasm[**?**])
3. Prebuild index before mapping and then map
4. Map any genome sequence against itself

Minimap does all this mapping basically in three steps. They are

First computing the minimizers

Secondly indexing the minimizers properly

Finally Mapping provided the two sequences.

Minimap generally generates a file in PAF format. Details of the format can be obtained from the cited papers above. For our purpose we ran then Minimap with our test sequences in the first mode that is listed above. The we extracted information from the PAF file to compare our results with it. Results are shown in comparison section.

Minimap is comparatively faster and new tool. But we are not going to discuss it more because of the reasons below.

1. The author himself declared that the tool is not tested properly yet and it needs a lot of testing.
2. It is basically tweaked for its assembly tool miniasm[**?**].
3. The tool is multi-threaded and our tool is not yet ready to process thing parallel.

2.1.2 NanoBLASTer

NanoBLASTer[**?**] is a new tool that does the alignment work. The target of this tool is ONT's MinION technologies which generate long noisy reads. This work is the main base of our work as it targets noisy reads like us. It aligns faster than GraphMap[**?**] and have higher sensitivity than LAST[**?**], BLAST[**?**] and BWA-MEM[**?**].

NanoBLASTer has introduced many smart and elegant enhancements to hold higher sensitivity and performance while reads have higher error rates. The basic underlying technique of this tool is mainly "seed-and-extend" technique. Inspired by BLAST[**?**] this tool also uses a technique that is called fixed size exact matching seeds that is then enhanced by a dynamic programming based extension mechanism.

As this tool is specifically designed for MinION[**?**] reads and because such reads can have 10% to 40% base error rate[**?**, **?**], before extension of the seeds NanoBLASTer uses one-dimensional clustering technique to find out longer high scoring segments dubbed as alignment anchors. Those anchors are then scored against their length and similarity and top prioritized anchors are extended into a proper length alignments using a block-wise dynamic programming approach.

As experiments and results are shown in NanoBLASTer[**?**] paper. It can be observed that NanoBLASTer is faster than BLAST, BWA-MEM, and GraphMap. And in regards of specificity NanoBLASTer is many greater than GraphMap. Besides all these there can be improvements done on NanoBLASTer in regards of memory efficiency.

2.1.3 BWA

BWA has two tools separately for long[**?**] and short[**?**] reads. One is BWA-SW or Burrows-Wheeler Aligner's Smith-Waterman Alignment which is for aligning long reads and it is an extended version of their main tool Burrows-Wheeler Alignment tool (BWA) which is mainly to align short reads. BWA is also an enhanced version of their previous tool MAQ[**?**]. But MAQ had some draw backs that it could not handle gapped alignment for the most frequent type of reads called single-end reads. But in case of long reads, inserts and deletes (indels) are frequent. The speed of MAQ is also a big issue when alignment scaled up.

BWA is developed mainly for handling the Illumina/Solexa[**?**] technology's 30-100 BP reads. It produces 5 crore to 20 crore reads in every run of the machine. To map these reads to human genome was a big challenge. Many hash-table based tools are developed to face this challenge like ZOOM[**?**], MAQ[**?**], CloudBurst[**?**], RMAP[**?**], SeqMap[**?**], SHRiMP[**?**]. As they have used hash-table, their memory use is moderate, but they could align very few reads considering the whole human genome. All of them are single threaded and hash the read first then search it in the reference[**?**]. On the other hand, some tools hash the genome.

PASS[?], MOM[?], NovoAlign[?], SOAPv1[?], ReSEQ[?], ProbeMatch[?] are in this category. They could be multi-threaded, but their main bottleneck is they usually take large memory to build an index, specially for the human genome[?]. These tools are dependent on the error rate of the data. More error could consume more. Slider[?] is another type of software which use merge-sorting which includes read sequences and reference subsequences.

None of the above tools cared about the BWT FM-index[?]. But efficiency of the FM-index is suitable for this work. SOAPv2[?] and Bowtie [?] inaugurated the process of fulfillment of the demand. BWA[?] is a continuation of that process. This last tool made the thing efficient by following the technique of backward search[?] and traversing the prefix trie of the genome in top-down fashion[?]. It reduces the memory consumption of the whole comparing to the other tools. It takes linear time to find the count of a read. There is no relation of the complexity with the number of count. That means whatever the length of the genome, the complexity does not depend on it. So, it could be the reference genome of Human, E.Coli or any other species, all needs same time to search the count, the length of the read. However, BWA allows K edit distance with respect to the query read.

In Suffix Trie, all repeated string follow the same path and to align one should not match it with each piece of repeated string. It is an advantage of using Suffix Trie[?] and this makes the FM-index based algorithms efficient. As it stated before, BWA is an implementation of inexact matching and also works for paired-end data.

Burrows-Wheeler Aligner's Smith-Waterman Alignment (BWA-SW)[?] is a tool which is developed dedicatedly for long reads tweaking the BWA. This tool works better than BLAT[?] and as accurate as SSAHA 2[?]. Both are hashing-based software. There are some other tools which developed for having extra care on local alignment such as FASTA[?], BLAST[?], MegaBLAST[?, ?], PatternHunter[?] etc. They also tried to speed up the process of matching against large reference genome. PacBio reads are very large and it would be more larger day by day. In early days of it's testing, it produced longer than 1K bp reads[?]. There are some other techniques used to tune the tools for long reads like bounding search process[?], filtering out bad matches with q-gram filtration[?, ?], spanning the entire read[?, ?, ?] and spaced seed templates[?]. Cuashaw2[?], GEM[?] also performs comparatively well in some cases.

Most of the tools used hash-table based algorithm, but it is not the only option. Between suffix

tree of the reference and a query sequence Smith–Waterman-like dynamic programming that could be applied[**?**]. Here, avoiding the repeated alignment, time could be saved. Later in this chapter, another tool name bowtie2[**?**] would be described which uses the same approach and some cases it performs better.

2.1.4 MUMmer

MUMmer is a versatile and open software for comparing large scale genome. Basically based on the Suffix Tree technique that MUMmer 1.0 first introduced in 1999.

In 1999 the when MUMmer 1.0 development started, several other tools for large-scale genome comparison have also been developed.Such as AVID[**?**], MGA[**?**], BLASTZ[**?**], LAGAN[**?**] and SSAHA[**?**] etc. Most of the above mentioned programs follow an approach which is actually anchor-based. We can divide the whole process into three phase:

1. Firstly, Precomputation of the potential anchors.
2. Secondly, Calculating the colinear sequence of the non-overlapping potential anchors.
3. Finally, Alignment of the gaps in between the anchors .

The conventional approach of computing potential anchors is to find first maximal matches of some length l or longer using a generate and test approach.In the first step, all possible matches of a fixed length $k < l$, called k-mers, are generated using general methods basically based on hashing. Then each of the generated k-mers is checked to examine if it can be extended to a maximal exact match of length at least l.

As there are several disadvantages of hashing approaches. Considering those MUMmer 1.0 was the first tool system to use Suffix Trees to find potential anchors for alignment. Suffix Trees have been studied and researched for almost three or four decades in computer science.

Suffix Tree is elegant data structure for representing all the substrings of a string whether it is a plain text, DNA sequence or a DNA sequence. A suffix tree for any string having a length of n can be represented in space proportional to n. Also fast algorithms have been developed and designed to construct a Suffix Tree of a string in time proportional to n[**?**, **?**]. And given the Suffix Tree of $\boldsymbol{S}$ and another query string $\boldsymbol{Q}$ of length m, there are algorithms to calculate all unique maximal matches between $\boldsymbol{S}$ and $\boldsymbol{Q}$ in time proportional to m.

Above features of Suffix Tree have made it an important data structure for large-scale genome analysis. MUMmer 1.0 first introduced Suffix Tree in such alignment technique.

From the development of MUMmer 1.0 in 1999 there have been three version of MUMmer upto 2016.

1. MUMmer 1.0 [**?**]
2. MUMmer 2.0 [**?**]
3. MUMmer 3.0 [**?**]

MUMmer 3.0 is most recent and feature added version MUMmer. Additional features like

1. New Java viewer, DisplayMUMS, a very new graphical output tool to make images in fig-format or portable document format(pdf). This can also show alignment of a set of contigs to a reference genome/chromosome.
2. New feature to find non-unique matches.
3. Ability to run multi-contig query against a multi-contig reference.

In the algorithm level MUMmer 3.0 is a complete rewrite of the Suffix-Tree code pivoting on compact Suffix-Tree representation of [**?**]. This same compact Suffix-Tree is also used in repeat analysis tool called REPuter[**?**].

In MUMmer 3.0 implementation was improved to a great level so that it enables MUMmer to allow sequences up to 250 Mbp on a personal computer with only a 4 gigabytes (GB) of real memory or RAM. Which comes from the cost of a slightly larger space usage per base pair. As an example anyone can construct the Suffix Tree for human chromosome 2(237.6 Mbp, the largest human chromosome) using 15.4 bytes per base pair.

MUMmer now requires approximately 25% less memory than the last release of 2.1 and it also runs slightly faster.

If compared to the initial release in 1999, the MUMmer 3.0 system is more than two times faster and uses less than half memory. Like MUMmer 2.1, MUMmer 3.0 release also streams query read sequences against the prebuild Suffix Tree of the target genome/reference sequence. So the sum total space requirement of MUMmer 3.0 is the size of the Suffix Tree as well as the size of the reference and the query sequences.

With the development of MUMmer 3.0, researcher got the capability to virtually any two genomes or sets of genomic sequences using computers commonly available todays. Bacterial genomes and small eukaryotes can be aligned on a standard personal desktop computer while larger genomes may require larger or server-class computers.

2.1.5 Bowtie

Like other programs mentioned above Bowtie[**?**] is a very fast and space-efficient that means also memory effective alignment tool mainly for aligning short reads of DNA sequences to a large reference genome sequence. Bowtie basically adopts the main Burrows-Wheeler techniques and then extending with a smart quality-aware backtracking method allowing mismatches in the sequences. Bowtie using the parallel threading power can also achieve better and greater alignment speed.

The underlying technique of Bowtie is basically Burrows-Wheeler index based on full-text minute-space(FM) index[**?**]. This technique has a very little memory footprint. Like only 1.3 gigabytes (GB) for the human genome. As a result Bowtie is allowed to run a typical and widely available desktop computer with 2 GB of RAM or real memory.

The conventional method for searching in an FM index is like the exact-matching algorithm of Ferragina and Manzini[**?**]. But Bowtie does not exactly adopts this algorithm because exact matching does not for the errors caused by sequencing or any sort of genetic variations. Bowtie introduce two smart extensions that make the method applicable to sort read alignments. These extensions are

1. A smart quality-aware backtracking algorithms enabling mismatches and also favors high-quality alignments.

2. Double Indexing, another smart way to avoid the excessive use of backtracking.

Bowtie follows a similar technique to Maq's[**?**] where it is allowed to a small number of mismatches within each read.

The main Bowtie algorithm consists of three main phases. These three phases alternate between using the forward and mirror indices that is describe in [**?**].

1. First phase uses the mirror index and invokes the aligner to locate alignments for cases 1 and 2.

2. Second phase finds partial alignments with mismatches only in the hi-half.

3. Third and final phase attempts to extend those partial alignments into the full alignments.

After the deployment of Bowtie in 2009 for the first time Bowtie has gone through several minor releases. After that Bowtie2 has come in 2012[**?**] with several improvements.Bowtie generally fail to align reads having gaps and therefore miss important information. Whereas Bowtie2 extends the FM-index based technique of Bowtie to perform gapped alignment basically dividing the algorithm broadly in two steps.

1. The initial step is defined as ungapped seed-finding stage which takes the full advantage from the speed and space efficiency of the FM-index.

2. The second stage is defined by gapped extension stage which basically uses dynamic programming technique and takes the full advantages of parallel processing of modern processors.

For every read Bowtie2 works in basic four steps. They are

1. In first step, Bowtie2 gleans seed substring from the read and its reverse complement.

2. In the second step,the gleaned substrings are aligned to a reference genome sequence in a ungapped style using FM-index.

3. In third step, Those seed alignments are prioritized and their positions in the reference genome sequence are then extracted from the FM-index.

4. In fourth and final step, seeds are extended into full alignments.

Bowtie2 came mainly to address the limitation of Bowtie of failing in cases of reads containing gaps. Though Bowtie2 addresses it well another tool BWA-SW[**?**] came first addressing the same problem. Several comparisons can be found here [**?**] that in many cases Bowtie2 works better than BWA-SW.

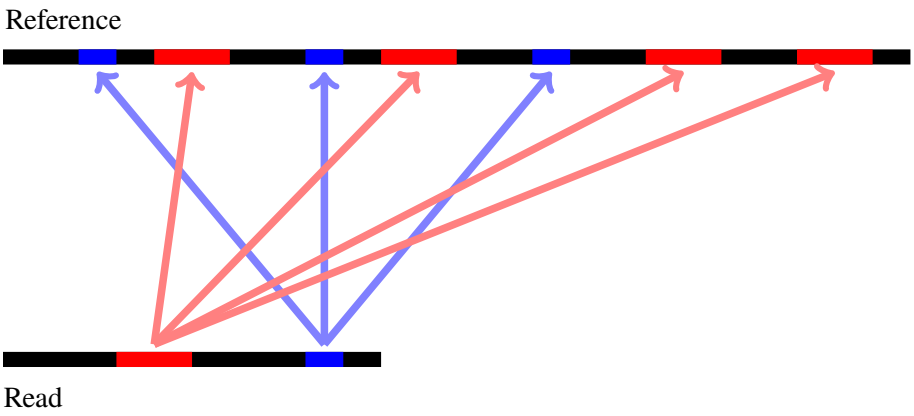

Figure 2.1: Mapping is just indicating the clusters of a large segment of the read in reference. The red large segment is found in four places in the reference where blue large segment is found in three places.

2.2 Basics

In this section, some basic concept about Mapper and Aligner would be covered. Some basic algorithm or data structures would be described as it needed to understand the work and to compare with other tools.

2.2.1 Mapper and Aligner

The terms mapper and aligner often used alternatively[**?**] to indicate a tool which align several portions of a read with several portions of a reference genome for certain reasons[**?**]. But in recent days, these words carry a slightly different meaning.

Mapper seeks for most matched clusters in the reference genome where the read could be fitted best. Based on some parameter it just identify most similar clusters which would help the aligner to align faster and easier. Figure **??** illustrates the concept.

On the other hand, aligner takes those mapping and align the read with the best suitable portion considering some insertion, deletion, mismatch if needed. Figure **??** shows a small demonstration where aligner picks one of the possible alignment and tells the read best fitted here, then stores this result in a specific format[**?**].

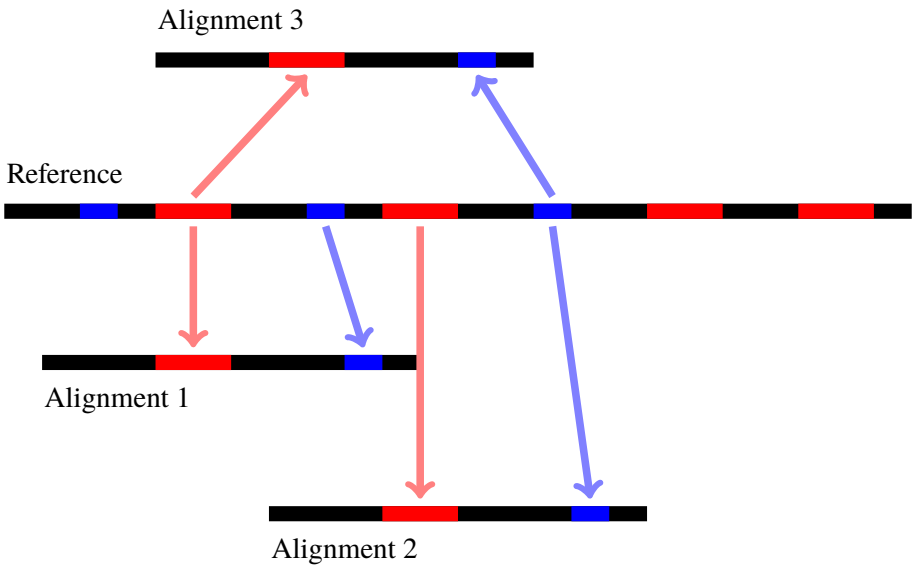

Figure 2.2: Three possible alignment are showed based on the read mapping. The aligner does this type of alignment and picks most probable alignment based on several parameters.

2.2.2 Suffix Trie

Before knowing about Suffix Trie we should have a quick glance at Trie. Sometimes it is considered that the word "Trie" comes from the word "Retrieval". As generally Trie is used for storing string type of data or information and then retrieve. Trie is also known as digital tree, radix tree or prefix tree. By formal definition a Trie is a tree containing a collection of strings with a single node per common prefix. It is the smallest tree such that:

1. Every edge is labeled with a character ***C*** from the alphabet set of the whole string.
2. Every node has at most one edge emanating from that node labeled as ***C***.
3. Every prefix of the string stored can be found along some path starting from the root of the tree.

Suffix Trie is nothing but only a Trie containing all the suffixes of a text ***T***. Let's take a text ***T*** = "ATATACA". All suffixes of the above text would be figure **??**

Now have to just build a Trie for all the suffixes. At first we need to add a special terminal character $ to the end of the ***T***. This is because

ATATACA
TATACA
ATACA
TACA
ACA
CA
A

Figure 2.3: Suffixes of the text "ATATACA".

ATATACA$
TATACA$
ATACA$
TACA$
ACA$
CA$
A$
$

Figure 2.4: Suffixes of the text "ATATACA$".

1. $ is a character that does not appear anywhere in ***T*** except at last and we define it as to be less than other characters. (For DNA sequences ***$<A<C<G<T***)

2. As a result ***$*** guarantees no suffix is a prefix of any other suffix.

So the resulting suffixes of the text ***T$*** will be figure **??** Now Building Trie for the above suffixes will result in such Trie Tree as in .

2.2.2.1 Complexity Analysis of Suffix Trie

As Trie takes a huge memory depending on alphabet size and number of keys to be inserted. Suffix Trie also takes such memory and complexity. Insert and Search in Suffix Trie requires ***O(key_length)***. How ever the memory requirement of Suffix Trie is ***O(alphabet_size*key_length*N)***. Where N is the number of keys in Suffix Trie. For a text of length ***m***, N = m - 1.

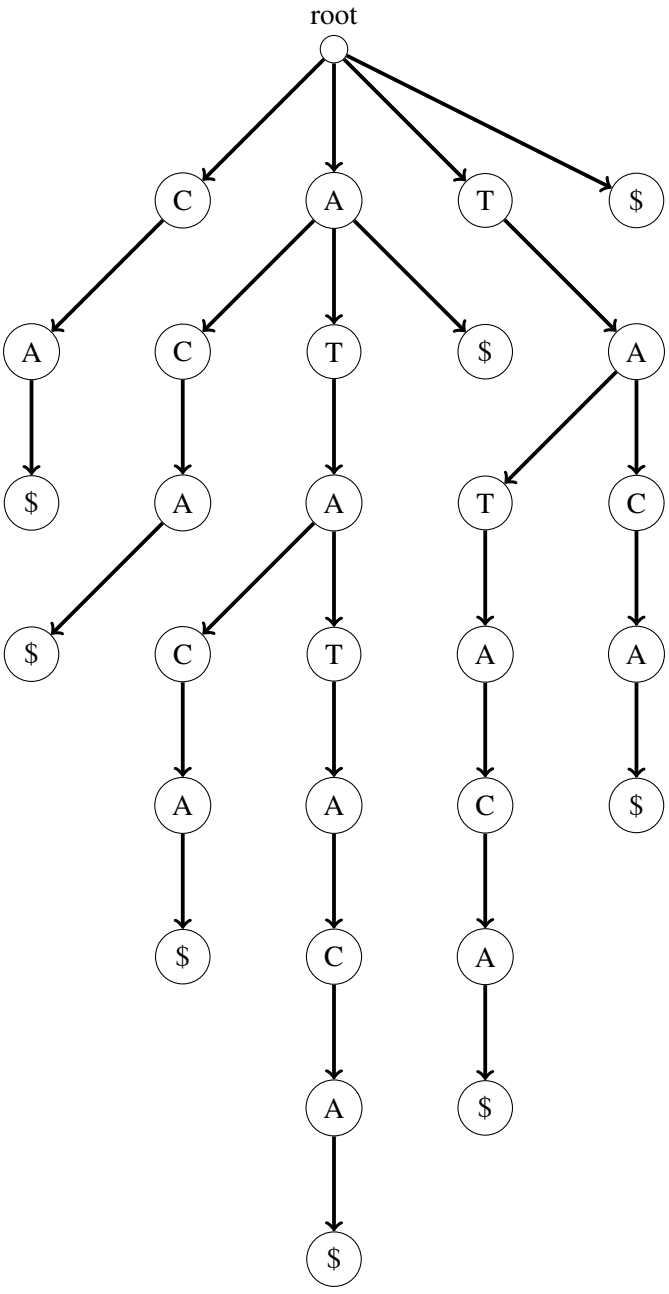

Figure 2.5: Cyclic Left Shift of the First Character of **S**.

2.2.3 Suffix Tree

The name of the title is enough to tell us that Suffix Tree is has to do with the Suffixes of any string/sequence. Suffix Tree has some advantages over Suffix Trie regarding space consumption. Considering the formal definition of we can state that a Suffix Tree for a sequence **S** considering it's length as **m** is:

1. A rooted tree **T** with **m** leaves numbered from 1 to **m**.
2. Every internal node of **T**, except perhaps the root, has at least two children emanating from it.
3. Every edge of **T** is labeled with a nonempty substring of **S**.
4. Every edge emanating from a node must have edge-labels starting with unique characters.
5. For any leaf ***i***, the concatenation of the edge-labels on the path from the root to the leaf ***i*** exactly represents **S[i,m]**, the suffix of **S** that starts at position ***i***.

To construct Suffix Tree, we first have to add the terminal character $ at the end of the text. For example Suffix Tree for the sequence ATATACA$ is in figure **??**[1]

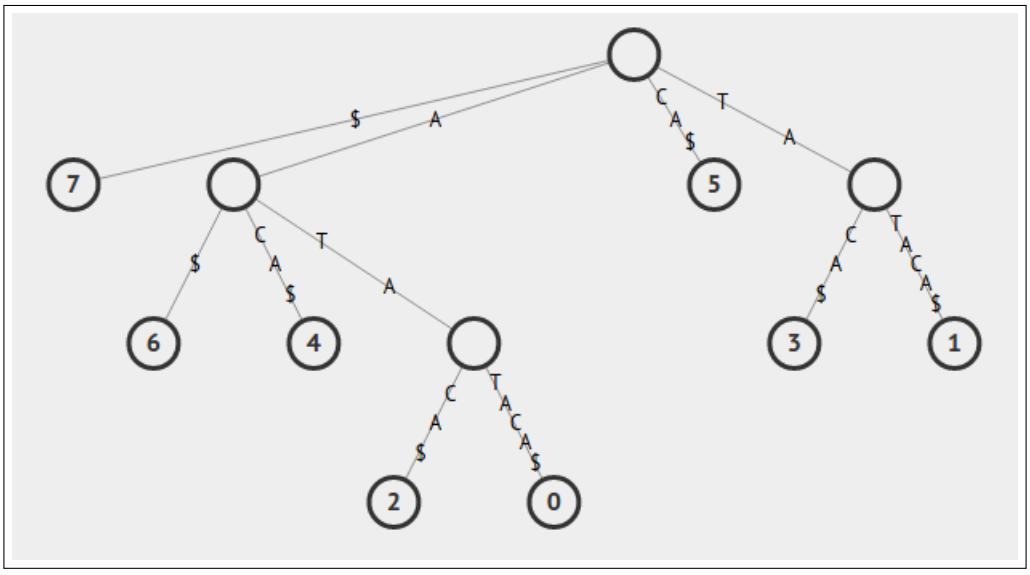

Figure 2.6: Suffix Tree for the sequence ATATACA$.

[1]Generated from https://visualgo.net/suffixtree

2.2.3.1 Analysis

By constructing the Suffix Tree in the previous example we can obviously perceive that this construction algorithm takes ***O(*m^2*)*** time. This can be done in ***O(m)*** time by Ukkonen's Algorithm[**?**] with a space complexity of ***O(m)***.

2.2.3.2 Real life Application of Suffix Tree

For gnomic studies in real life, many real life analysis tools uses Suffix Trees as their core level data structure enabling them fast and accurate analysis capability. Some of them enlisted below.

1. MUMmer for alignment purpose. [**?**, **?**, **?**]
2. REPuter for calculating and finding repeats in the whole genome sequence. [**?**]
3. Identifying sequence motifs.[**?**, **?**]
4. Multiple alignment.[**?**]

2.2.4 Suffix Array

2.2.4.1 Overview

Suffix Array[**?**] is a common data structure which keeps the starting index of sorted suffixes of a string. The definition of suffix is an important thing here. So, a suffix is a substring or consecutive characters from a string from any position to the end of the string. An example would make the thing more clear. So, lets take an example string,

$$S = ATTCGAGCATCAG$$

To keep the track of ending, an end mark like \$ should be appended.

$$S\$ = ATTCGAGCATCAG\$$$

The suffixes are shown in Figure **??**. After sorting lexicographically, the order of the suffixes becomes like Figure **??**. So, Suffix Array of S would be as in Figure **??**.

```
ATTCGAGCATCAG$ --  0
 TTCGAGCATCAG$ --  1
  TCGAGCATCAG$ --  2
   CGAGCATCAG$ --  3
    GAGCATCAG$ --  4
     AGCATCAG$ --  5
      GCATCAG$ --  6
       CATCAG$ --  7
        ATCAG$ --  8
         TCAG$ --  9
          CAG$ -- 10
           AG$ -- 11
            G$ -- 12
             $ -- 13
```

Figure 2.7: Suffixes of S$ in ascending order of their starting index.

```
$              -- 13
AG$            -- 11
AGCATCAG$      --  5
ATCAG$         --  8
ATTCGAGCATCAG$ --  0
CAG$           -- 10
CATCAG$        --  7
CGAGCATCAG$    --  3
G$             -- 12
GAGCATCAG$     --  4
GCATCAG$       --  6
TCAG$          --  9
TCGAGCATCAG$   --  2
TTCGAGCATCAG$  --  1
```

Figure 2.8: Suffixes of S$ sorted in Lexicographical Order with their starting index.

$S\$ =$	A	T	T	C	G	A	G	C	A	T	C	A	G	$
$SA(S) =$	14	12	6	9	0	11	8	4	13	5	7	10	3	2

Figure 2.9: Suffix Array of the string S, SA(S).

2.2.4.2 Building Suffix Array

An approach of building suffix array could have the following steps.

- Bulild a Suffix Tree as mentioned in the previous section.
- Do a Depth First Traversal storing indexes at leaves.

The memory complexity could be reduced following several approaches[**?**] with LCP array where the Suffix Tree should not be constructed. However, traversing n nodes need $O(n)$ time. So, it is the building time complexity of Suffix Array.

2.2.4.3 Querying in Suffix Array

A binary search algorithm[**?**] could be used to query in the suffix array. Say we want to have the answer of following two questions where $P = AT$ would be used as query string.

- Does P occurs in S?
- How many times does P occur in S?

To answer the first question, we would do a binary search. if any match found, then it occurs, otherwise not. Have a look at Figure **??** and see, it occurs. But here is steps in binary search.

- left = 0, right = 13, mid = 6. SA[6] = 8.
- S[8] = 'A', S[8+1] = 'T' which matches with P[0] and P[1]. So, T is a substring of S.

However, to answer the second question, we should have two binary search. One would find left bound and other would find right bound. The answer is just the difference between them.

2.2.4.4 Complexity Analysis

The memory complexity of Suffix Array is $O(n)$ where n is the length of the string. To index human genome it takes 12GB[**?**] where MUMmer[**?**] takes 47GB using Suffix Tree.

In case of searching, both of the answers in the querying section takes$O(mlog_2(n))$ where, n and m are the length of the main string and query string respectively. Here, m for comparison and $log_2(n)$ for binary search. But it could be reduced to $O(m+log_2(n))$ in practice using LCP[**?**,**?**,**?**,**?**] and other techniques[**?**, **?**, **?**].

2.2.5 Burrows-Wheeler Transform (BWT)

2.2.5.1 Overview

Burrows-Wheeler Transformation (BWT) is an interesting concept which is the basic building block of the world's one of the best indexing system called FM-index. It compresses the memory highly. Reordering the characters of a string, BWT transforms it to a more compression-friendly version. The original string could be retrieved from the transformed one through a couple of processing. The next few subsections would brief the compression procedure. A naive concept would be shown at first. Then the trade-offs would be discussed.

2.2.5.2 Burrows-Wheeler Matrix Construction

The first step of BWT is to construct the Burrows-Wheeler Matrix (BWM) of a string. But before that, a end mark should be append to the end of the string which mark should be considered as the lowest value comparing to the any other member of the alphabet set.

Let,

The alphabet set,

$$\sum = \{A, T, G, C\}$$

and a string,

$$S = ATTCGAGCATCAG$$

Let the endmark symbol = \$. So, appending this, the string becomes,

$$S = ATTCGAGCATCAG\$$$

The last thing to let is $n = |S|$, where, $|\cdot|$ means the length of the string.

The matrix is $n \times n$ dimensional. To construct the matrix, S should be considered as the first row. The next rows would be constructed by left shifting the characters of the S in cyclic order. That means, in the last row, the end mark symbol (\$) would be in the first column.

In the example above, $n = 14$. So, the dimension of the matrix would be 14×14.

Let S' be the string after first cyclic left shift operation on S. That means, the A at the beginning of the string would be removed and appended at the end of the string, after the end mark symbol.

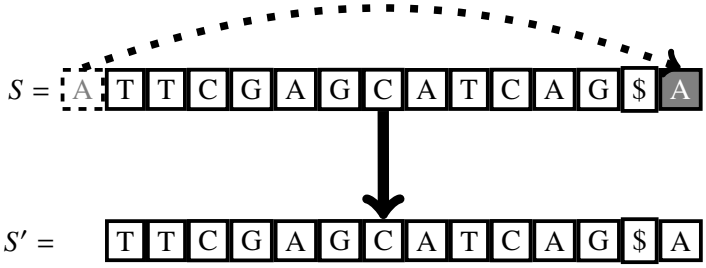

Figure 2.10: Cyclic Left Shift of the First Character of **S**.

Figure **??** would illustrate the process better.

$$S' = TTCGAGCATCAG\$A$$

It is the second row of the $BWM(S)$. The third row would be constructed by doing the same operation on S'. That means the first T in the above string would be moved at the end of the string, just after the character A.

$$S'' = TCGAGCATCAG\$AT$$

In this way, other rows would be constructed. The last row would be $\$ATTCGAGCATCAG$ having the end mark symbol at the beginning as mentioned above.

Assigning the row numbers from 0 to $n-1$, the final matrix becomes,

$$\textbf{Pre-BWM(S)} = \begin{cases} 0 & ATTCGAGCATCAG\$ \\ 1 & TTCGAGCATCAG\$A \\ 2 & TCGAGCATCAG\$AT \\ 3 & CGAGCATCAG\$ATT \\ 4 & GAGCATCAG\$ATTC \\ 5 & AGCATCAG\$ATTCG \\ 6 & GCATCAG\$ATTCGA \\ 7 & CATCAG\$ATTCGAG \\ 8 & ATCAG\$ATTCGAGC \\ 9 & TCAG\$ATTCGAGCA \\ 10 & CAG\$ATTCGAGCAT \\ 11 & AG\$ATTCGAGCATC \\ 12 & G\$ATTCGAGCATCA \\ 13 & \$ATTCGAGCATCAG \end{cases}$$

After preparing the matrix, the rows of it would be sorted lexicographically. That means, if A and B are two strings containing characters $A_0A_1 \cdots A_{n-1}$ and $B_0B_1 \cdots B_{n-1}$ respectively having same length n and there exists an i, for which $A_i < B_i$ and $A_j = B_j$ for $0 \leq j < i < n$, then A would come before B.

So, the sorted matrix,

$$\textbf{BWM(S)} = \begin{cases} 13 & \$ATTCGAGCATCAG \\ 11 & AG\$ATTCGAGCATC \\ 5 & AGCATCAG\$ATTCG \\ 8 & ATCAG\$ATTCGAGC \\ 0 & ATTCGAGCATCAG\$ \\ 10 & CAG\$ATTCGAGCAT \\ 7 & CATCAG\$ATTCGAG \\ 3 & CGAGCATCAG\$ATT \\ 12 & G\$ATTCGAGCATCA \\ 4 & GAGCATCAG\$ATTC \\ 6 & GCATCAG\$ATTCGA \\ 9 & TCAG\$ATTCGAGCA \\ 2 & TCGAGCATCAG\$AT \\ 1 & TTCGAGCATCAG\$A \end{cases}$$

The index of the rows are preserved for certain reasons which would be discussed later. It is a clear property of the BWM that the last row of the previous matrix would go top here.

2.2.5.3 Finding BWT(S)

Picking the last character of every string from top to bottom, a new string would be constructed which is actually the desired $BWT(S)$. To state more clearly, the above matrix could be compared with the matrix below.

$$BWM(S)_{n,n} = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{1,1} & s_{2,2} & \cdots & s_{n,n} \end{pmatrix}$$

Taking the last character from each of the above string in matrix,

$$BWT(S) = s_{1,n}s_{2,n}\cdots s_{n,n}$$

In the actual case,

$$BWT(S) = GCGC\$TGTACAATAA$$

The indexes, we saw before, are same as in Suffix Array. BWM(S) is identical to the Figure **??**. So, the following equation should be hold:

$$BWT[i] = \begin{cases} S[SA[i]-1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases}$$

2.2.5.4 T-Ranking

T-Ranking is not directly ranking among the whole string, rather it keeps the relative position of the same characters. To define more easily, it could be stated that the T-Ranking of a character says how many times the character is found before this position in the string. This is the condition of our string after having T-Ranking.

$$S = A_0T_0T_1C_0G_0A_1G_1C_1A_2T_2C_2A_3G_2$$

2.2.5.5 LF Mapping

LF mapping is one of the basic property of BWM. Now, by left cyclic shifting S, we could generate the BWM again. If we have only the first row and the last row of the matrix with their

T-Ranking, then it is easy to find the actual string.

$$\textbf{BWM(S)} = \begin{cases} \$\ A_0T_0T_1C_0G_0A_1G_1C_1A_2T_2C_2A_3G_2 \\ A_3G_2\$\ A_0T_0T_1C_0G_0A_1G_1C_1A_2T_2C_2 \\ A_1G_1C_1A_2T_2C_2A_3G_2\$\ A_0T_0T_1C_0G_0 \\ A_2T_2C_2A_3G_2\$\ A_0T_0T_1C_0G_0A_1G_1C_1 \\ A_0T_0T_1C_0G_0A_1G_1C_1A_2T_2C_2A_3G_2\$ \\ C_2A_3G_2\$\ A_0T_0T_1C_0G_0A_1G_1C_1A_2T_2 \\ C_1A_2T_2C_2A_3G_2\$\ A_0T_0T_1C_0G_0A_1G_1 \\ C_0G_0A_1G_1C_1A_2T_2C_2A_3G_2\$\ A_0T_0T_1 \\ G_2\$\ A_0T_0T_1C_0G_0A_1G_1C_1A_2T_2C_2A_3 \\ G_0A_1G_1C_1A_2T_2C_2A_3G_2\$\ A_0T_0T_1C_0 \\ G_1C_1A_2T_2C_2A_3G_2\$\ A_0T_0T_1C_0G_0A_1 \\ T_2C_2A_3G_2\$\ A_0T_0T_1C_0G_0A_1G_1C_1A_2 \\ T_1C_0G_0A_1G_1C_1A_2T_2C_2A_3G_2\$\ A_0T_0 \\ T_0T_1C_0G_0A_1G_1C_1A_2T_2C_2A_3G_2\$\ A_0 \end{cases}$$

To make the think easy, the first and last columns would be kept and from this, the string could be found again. This type of keeping Last (L) and First(F) columns' data is called LF Mapping. Figure **??** the LF Mapping of S.

2.2.5.6 Retrieving the string back using LF Mapping

From the mapping in figure **??** the original string could be retrieved by following the steps:

1. Start from the '$' character marking as current character which is also the first character of F and take an empty string.
2. Append the now character in previous string.
3. Find the current character in F. and take the corresponding L as current character.

F	L
$	G_2
A_3	C_2
A_1	G_0
A_2	C_1
A_0	$
C_2	T_2
C_1	G_1
C_0	T_1
G_2	A_3
G_0	C_0
G_1	A_1
T_2	A_2
T_1	T_0
T_0	A_0

Figure 2.11: LF Mapping of the string S.

4. Loop through until current character becomes '$'.

5. Reverse the string to get the original string.

2.2.5.7 Characteristics of BWT

There are a lot of properties of BWT, but it is here to efficiently manipulate biological genome sequences. So, here are some of the important properties of BWT:

- One important properties of T-Ranking is, in both Last and First columns, the sequence of T-Ranking comes in the same order.
- BWT takes very low amount of memory.

2.2.6 FM-index

FM-index[**?**] is the indexing implementation of Burrows-Wheeler Transform (BWT)[**?**]. It is an implementation with some auxiliary data structure. Binary search is used in suffix array which would no more effective as all the columns are removed except first and last one. While

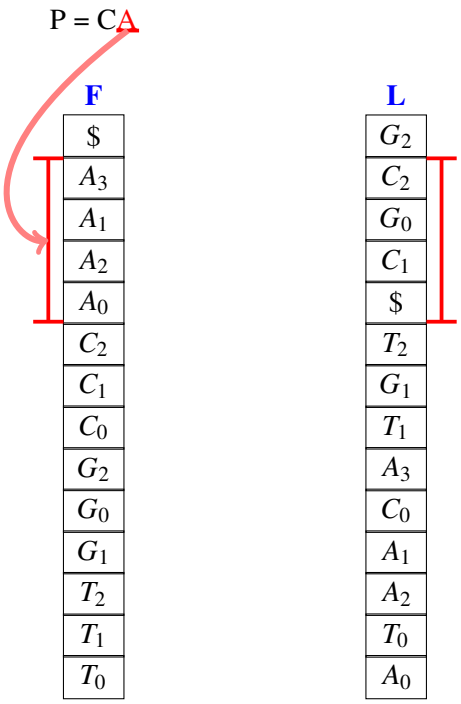

Figure 2.12: Finding Suffix A of Query String P in F of LF Mapping.

compressing, it is easy to store the first column as it would be sorted. The ranks of the first columns would still be preserved as they are same as the last column.

To match a string, suffix matching would be done here. Lets have an example,

$$P = CA$$

. So, the last character A should be searched in F of LF-Mapping. The next character should be occur in L and the range of that character would be found from there. Figure **??** shows that for the next character C, only rank 2 and 1 is eligible to be searched. So, C should be searched in the range of of C characters in F with starting rank 2 and ending rank 1. Figure **??** shows a better illustration. As we have arrived at the begin of the string, the string exists.

The exact positions could be found using a suffix array which is a long story. Further efficiency could be gained using some auxiliary data structures. It could be shown that if a range could be found in LF-mapping, then it would be in suffix array with same length.

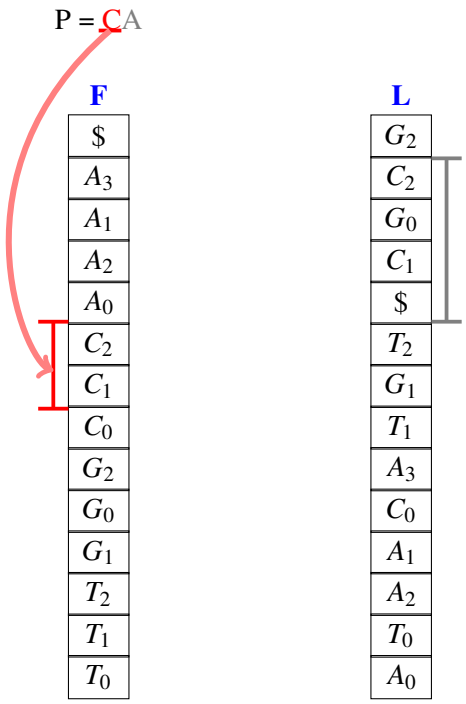

Figure 2.13: Finding Suffix C of Query String P in F of LF Mapping.

2.2.7 Minimizer

Minimizer is a special concept that indicates the lexicographically smallest M-mer among K-mers and their reverse complement in a window of X[**?**,**?**,**?**,**?**,**?**]. Figure **??** shows how to calculate a minimizer from a window. Here, the first window is `ATTCGAGCAT`, second one is `TTCGAGCATC`, third one is `TCGAGCATCA`, fourth and fifth windows are `CGAGCATCAG` and `GAGCATCAGT` respectively. The demonstration shows the first window only. For other windows, same calculation would be happaned. It is shown that the minimizer from first window is `ATGCTC`. In the second window, the minimizer would be switched to `AGCATC`.

The definition of minimizer could be changed for the necessity. In the last version of our tool, we have considered only the M-mers excluding their reverse complement as we are not working both of the directions at a time. We process with either forward direction or backward direction, but not both simultaneously. That does not mean that we are not considering the reverse complement totally. If it is stated that the read is from forward direction of reference, then there is no need to find it in reverse direction. On the other hand, if it is cut from reverse direction, then there is no need to seek in forward direction. The data set would tell us about the direction, that's why we are

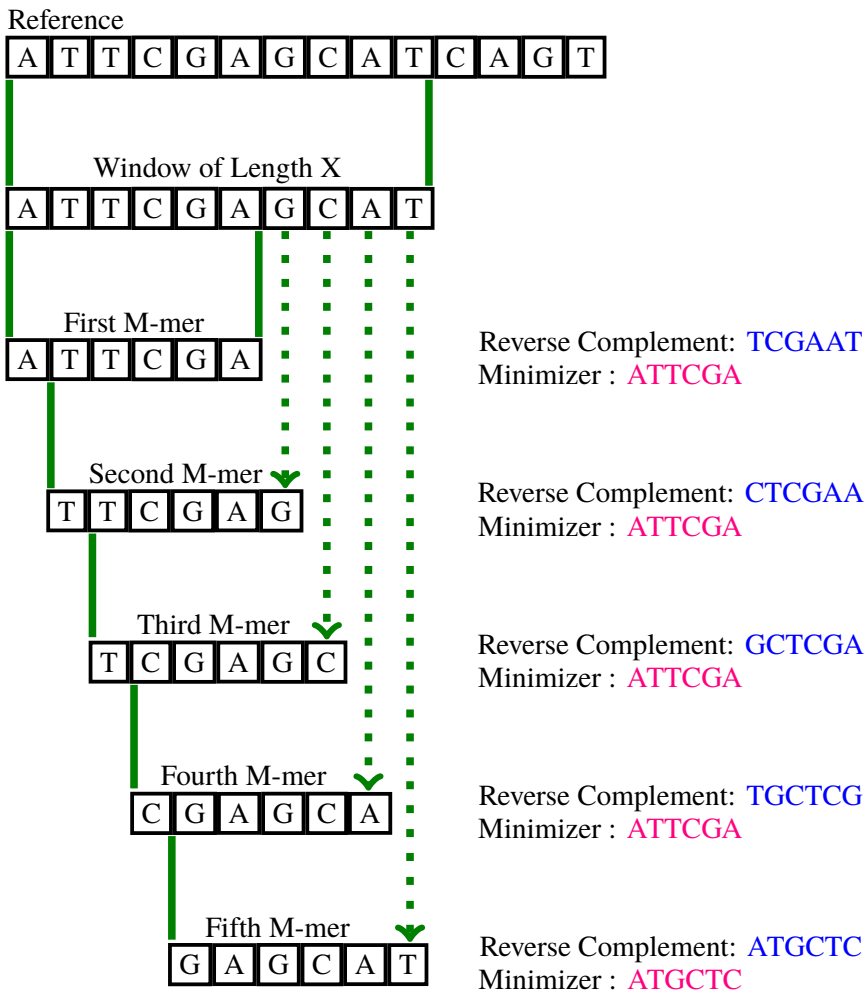

Figure 2.14: Step by Step Computation of a Minimizer in a Window of X = 10 having M-mer of Length 6. The Minimizer of the Window is ATGCTC.

omitting reverse complements.

2.2.8 Gapped K-mer

The data is noisy. The more the error, the more the mismatch. For the sake of allowing mismatch, we have to treat them as match while running matching algorithms. So, every mismatch which was actually match, but gone mismatch due to error, should be converted to same character. We introduce '_' as the common character. We would convert every third base of a K-mer to a '_'. So, we are considering 1/3 mismatch as match which is around 33%. But our data may have around 20% error. We took some more in percentage for reducing the biasness of the assumption to occur the mismatch at every third position.

Lets see an example. We have a K-mer of length 14:

```
ACTTGATCTAGTAC
```

If we assume 25% error, we would replace every fourth base as '_' like below.

```
ACT_GAT_TAG_AC
```

But in our case, we extended the error rate to 33% and to take the effect, we would replace every third base instead of fourth.

```
AC_TG_TC_AG_AC
```

As in both read and reference the character became same, it would match in both.

Chapter 3

Methodology

In this chapter we would discuss our approaches conceptually to reach a solution. Different approaches works better in different conditions. Some could map very efficiently but they takes time. Some may map rough but give result fast. Some work better when data is error free. Our target is ONT reads which has errors. So, at last we merged all the techniques that necessary to conclude a tool which could survive against so many errors in reads.

3.1 Naive Minimizer Approach

Minimizer is described enough above in the Basics section. We ran a W length window over the whole reference genome and stored the K length minimizers which could cover consecutive windows. By the term Consecutive Windows, we meant a set of windows where each window's starting position has starting position difference one with at least one other window in the set. We would only store the position of the first occurrence of the minimizer in the consecutive windows along with the lowest starting position and highest ending position among the windows. We would map this kind of triplets to the corresponding minimizer with the help of unordered map for better efficiency. For making the program more efficient in perspective of time and memory, we have stored minimizer converting it to an integer.

Then we took windows from reads and find it in the unordered map converting to integer for being compatible. On the other experiment, we have done only taking K-mers instead of window or minimizer. In the last version, this idea survived. In Figure **??**, there are four minimizers found

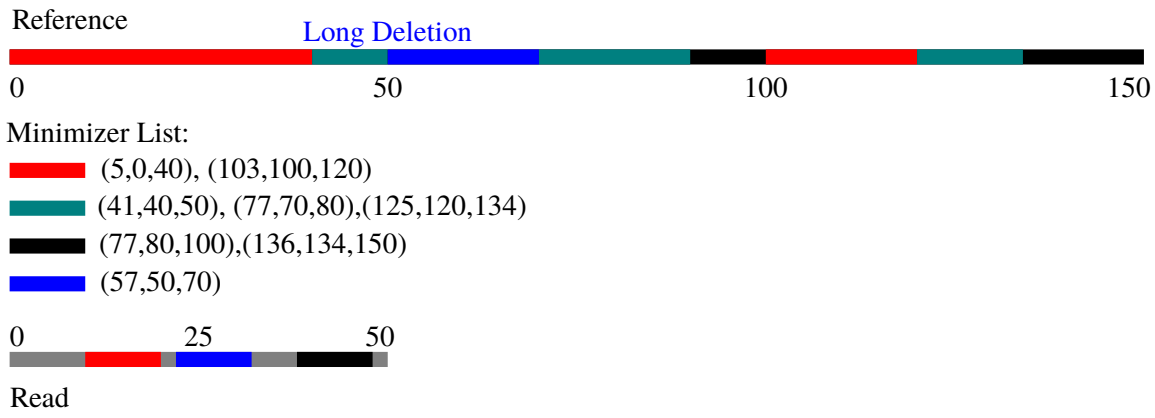

Figure 3.1: There are four distinct minimizer found in the reference which are showed with four different color. Their positions are listed with triplets (position of minimizer, starting position of the consecutive covered window, ending position of the consecutive covered window). In the read, we have found three minimizer among four. Grayed portion indicates that these minimizer are not found in the reference.In Naive Minimizer approach, mapping is done based on these information.

in different positions of the reference. In the read which positions could not be matched are shown in gray color. Minimap[**?**] also used this technique with some enhancement in their tool.

3.2 Efficient Window Traversing Technique

In general finding minimizer of length $\boldsymbol{K}$ of every $\boldsymbol{W}$ window a genome/read sequence would take a lot of time if it is done with polynomial complexity as well as if the sequence is very long. But the brighter side is we have done the whole process of finding minimizer of $\boldsymbol{K}$ of every $\boldsymbol{W}$ window of the reference sequence with a overall linear complexity. That helped our over all complexity of this tool boost up a lot.

Algorithm 1 Indexing All Minimizers in Reference Running a Window of Length W in $O(n)$

Input: S is the reference sequence, W is window size and K is the length of K-mer to work with the window.

Output: An Unordered Map of `int` to `boolean` where every integer represents a K-mer and the `boolean` value expresses it is in the reference as minimizer. All the entries should have this value equal to `true`.

1: **function** MINIMIZERINDEXING(S, W, K)

2: **Local Variables:** $n, i, temps, minimizer, prevmin, sliding\_window$ ▷

sliding_window is a deque data structure of type *my_data* which has two integer members *index* and *minimizer*

3: $n \leftarrow |S|$ ▷ |s| means length of the string s

4: $temps, minimizer, prevmin \leftarrow$-1

5: $temps \leftarrow$Pat2Num(S[0:K]) ▷ *Pat2Num()* function takes a K-mer string and converts it to the corresponding integer and returns.

6: **for** $i \leftarrow K$ **to** $i < n$ and $i < W$ **do**

7: **if** S[i] not equal to $'\_'$ **then**

8: $temps \leftarrow S[i : i + k]$

9: **end if**

10: $minimizer \leftarrow temps$

11: **while** *sliding_window* is **not empty** and $minimizer <$ *sliding_window.back().minimizer* **do**

12: sliding_window.pop_back()

13: **end while**

14: sliding_window.push_back({minimizer, i-K+1})

15: **end for**

16: $prevmin \leftarrow sliding\_window.front().minimizer$

17: **for** $i \leftarrow i$ **to** $i < n$ **do**

18: **if** S[i] not equal to $'\_'$ **then**

19: $temps \leftarrow S[i : i + k]$

20: **end if**

21: **if** S[i-K] not equal to $'\_'$ **then continue**

22: **end if**

23: $minimizer \leftarrow temps$

24: **while** *sliding_window* is **not empty** and $sliding\_window.front().index \leq (i - W)$ **do**

25: sliding_window.pop_front()

26: **end while**

27: **while** *sliding_window* is **not empty** and $minimizer <$ *sliding_window.back().minimizer*

```
    do
28:             sliding_window.pop_back()
29:         end while
30:         sliding_window.push_back( {minimizer, i-K+1} )
31:         if sliding_window.front().minimizer not equal to prevmin then
32:             if Map.count(sliding_window.front().minimizer) = 0 then
33:                 Map[prevmin] ← true
34:                 prevmin ← sliding_window.front().minimizer
35:             end if
36:         end if
37:     end for
38:     if Map.count(sliding_window.front().minimizer) = 0 then
39:         Map[sliding_window.front().minimizer] ← true
40:     end if
41:     return Map
42: end function
```

Using a simple data structure which contained a minimizer as an integer and its position and a C++ built in Standard Template Library(STL) container deque of that data structure enabled us to do the whole minimizer indexing process in linear time. We basically used a sliding window technique.

At first we fill up deque with the first K mer and its position as 0. Then for the first window of W length for each new base we update a temporary minimizer variable. Then we start erasing elements from the back of the deque until that minimizer is greater than our new temporary minimizer. After that we insert this temporary minimizer and it's position in the deque.

We set the global minimizer value as the minimizer of the first window which can be obtained from the front of the deque. After for each base of the sequence we take a new temporary minimizer variable containing the K mer considering the current base. Then we keep removing the elements from the deque which are out of the current window by popping from the front of the deque.Then again we start erasing elements from the back of the deque until that minimizer is greater than our new temporary minimizer. After that we insert this temporary minimizer and it's position in the

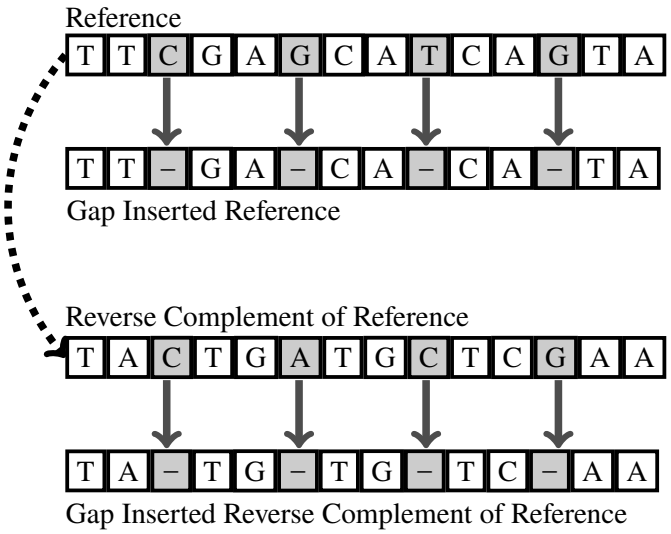

Figure 3.2: Reference Sequence and It's Reverse Complement with Gap Insert in Every Third Base.

deque.

From the front of the deque we get the minimizer of the current window. If its different from the previous global minimizer we update the map with the global minimizer and then update the global minimizer with minimizer of the current window.

Finally after iterating over the whole sequence, the minimizer of the last window will still be unchecked whether it is mapped or not. So we check it and map it if needed. In this whole process we just iterate over the given string once and for each base the insertion and deletion from the deque along with the insertion in the un-ordered map is ensured as $\boldsymbol{O}(1)$ in C++. So our overall complexity turns into as a linear one.

3.3 Gapped Minimizer

Why we use gapped minimizer is discussed in background study as gapped k-mer. Here we first take the reference genome and then insert gap or '_' in every third base of the reference sequence. Then we start building (W, K) - minimizer index from the reference sequence. Here (W,K)-minimizer notation means lexicographically minimum of all consecutive $\boldsymbol{K}$ mers from a $\boldsymbol{W}$ mer.

After the first W window's minimizer if found we map the minimizer with a vector. Here

the vector represents the position of that minimizer and the total length covered by the minimizer. Then ***W+1***th base is considered then for that window if a new minimizer is found we map this new minimizer in the same style. If no new minimizer is found that means the minimizer of the previous window is also a minimizer the current window then we just keep updating the end position that is covered by the minimizer. So ultimately we just iterating the sequence and for each right base we check if that window does not have a new minimizer then we update the covering end position of the last found minimizer and go on. When a new minimizer is found we just update the map with the previous minimizer with it's starting position, covering start position, covering end position triplet.

Finding the minimizer in every ***W*** window for the whole sequence takes almost linear time as shown in **??** and building the unordered map of minimizer with it's starting position, covering start position, covering end position triplet vector takes also constant time for every update and query in the map. So our method of building the whole index for gapped-minimizer for a reference sequence is almost linear in time.

For simulated data set we know which read is in forward direction and which read is in reverse direction. So for reads that are extracted from the reverse direction with compliment we need another such index of the reverse compliment of the reference sequence. For this we just take the reference first insert gap or '_' in it like afore mentioned method and then reverse complement the reference. Then again build another separate unordered map of minimizer with it's starting position, covering start position, covering end position triplet vector.

From some previous study [**?**] we find that a ***(W, K)***-minimizer occurs on average every ***(W+1)/2*** bases through a sequence. If minimizers occurred exactly every ***(W+1)/2*** bases, then every substring of length ***K +(W -1)/2*** would have a minimizer. As a result, if two strings have a ***K +W/2***-mer in common, then they are most likely to have a ***(W,K)***-minimizer in common. So for the reference we expect to have at least one new minimizer in every ***2*W*** window of the sequence.

Our another assumption is, for any read (from simulated data) number of total ***K***-mer of the read found in the reference as a minimizer will be at greater or equal than the total number of minimizer found in the location of reference from where the read was extracted from.

For this purpose we process some of the aligned read in such manner. We searched for every ***K***-mer of the read in the reference whether it occurs as a minimizer or not. If the read is in forward

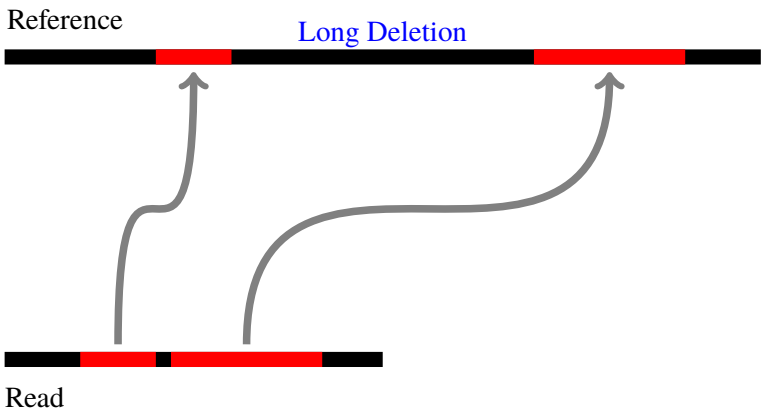

Figure 3.3: Mapping Two Long K-mers with Long Deletion in Read.

direction we searched it in the map generated for forward reference and if the read is in reverse direction we searched it in the map generated for reverse compliment of the reference. Then we counted how many ***K***-mer hits we get in the reference as minimizers. We also extracted the information of how many minimizers occurred in the section from the minimizer was taken from.

In the next chapter we will see that these two assumptions are met accordingly.

3.4 Mapping Using Naive BWT FM-index

BWT FM-index [**?**] is implemented naively at first. The main idea is if a long chunk of a read could be mapped to a long chunk of the reference, then the probability of this mapping to be accurate is quite high. One of the main drawback of NanoBLASTer [**?**] is it could not detect a long insertion or a long deletion. In this approach, the main goal is approaching this type of problem.

If two long chunks of a read is mapped to very close distance in reference but long distance in read, then it indicates that there is a long insertion in the read. Similarly, if these two mapped chunks maintain a long distance between them in reference but very short distance in read then there exists a long deletion in the read. Figure **??** and **??** make the idea more clear. The red portions indicate the long k-mers and the arrows indicates from read where they are mapped in reference.

So, from a certain value of K, we would check in the FM-index whether the K-mer exists in the reference or not. If we get any null set of location as result, we would consider the last not-null set of locations. That means, for an increasing K, if we get no location in the reference for that K-mer, then we would stop increasing the value of K and would work with the locations we get for $K-1$.

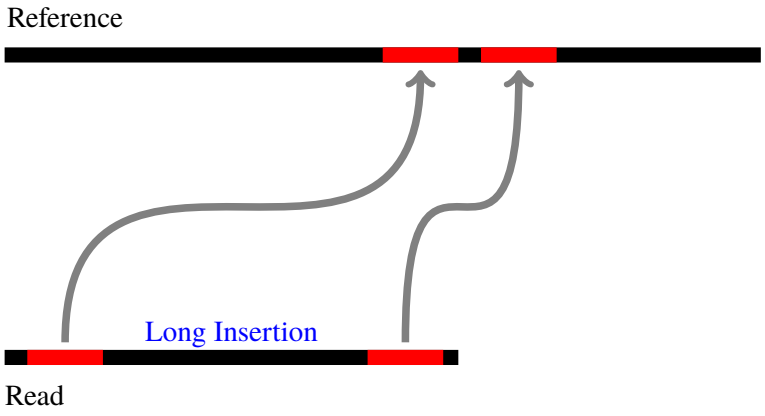

Figure 3.4: Mapping Two Long K-mer with Long Insertion in Read.

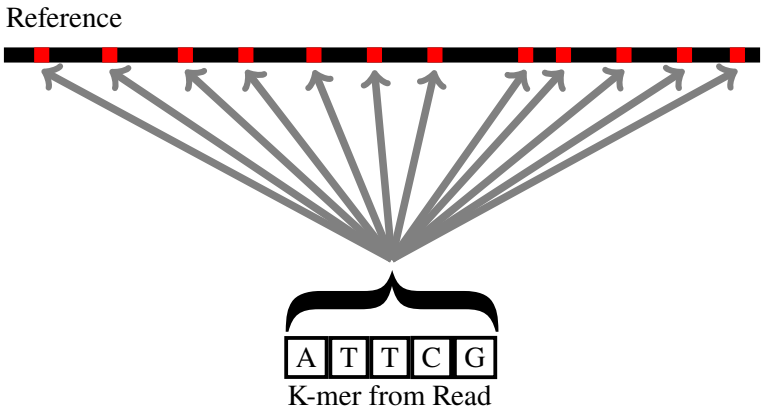

Figure 3.5: A K-mer with Value $K = K_{min}$ is Picked Up from Read and Indicated Where The K-mer is Found in the Reference.

Algorithm **??** may help to understand better. Figure **??** through **??** visualize the implementation.

3.5 Enhanced Search in BWT FM-index

For every K in the loop of BWT FM-index [**?**] in Algorithm **??** at line number **??** the *CountFM()* function is calling. And for every call, this function has to iterate over all previous K with the help of a function *BackwardSearch()*. It makes the complexity $O(K^2)$. Though it does not matter that more as the value of K would not be very large, but we are basically looking for larger K to map larger K-mers. So, it should be handled carefully. Therefore, if the read has less noise, it would increase the complexity which would be shown in the case of error free *Synthetic* Reads.

The *BackwardSearch()* and the *CountFM()* function would be modified in a way so that the

Algorithm 2 Mapping K-mers of Variable Lengths of a Read to Reference Using Naive FM-index

Input: K_{min} is the minimum length of K or the starting length of K, $fmIndex$ is the indexed reference which is basically a data structure, $read$ is the particular Read Sequence.

Output: A list of K-mers with their position in read and reference.

1: **function** MapWithNaiveFM($fmIndex, read, minK$)
2: **Local Variables:** $i, j, readLength, kmer, countOcc, kmer_{prev}, locations, retList$
3: readLength ← $|read|$ ▷ |s| means length of the string s
4: $retList$ ←null
5: **for** $i \leftarrow 0$ **to** $readLength - K_{min}$ **do**
6: $kmer_{prev}$ ←null
7: **for** $j \leftarrow K_{min}$ **to** $readLength - i + 1$ **do**
8: $kmer$ ←read[i : i + j] ▷ $s[a : b]$ means substring consisting of characters from index a(inclusive) to index b(exclusive) of string s
9: $countOcc$ ←CountFM(fmIndex, kmer) ▷ *CountFM* function takes two parameters, an *FM-index* data structure, a string and returns the count of occurrences of the string in the data structure
10: **if** $countOcc = 0$ **then break**
11: **end if**
12: $kmer_{prev}$ ←kmer
13: **end for**
14: **if** $kmer_{prev}$ **not *null* then**
15: $locations \leftarrow LocateFM(fmIndex, kmer_{prev})$ ▷ *LocateFM* function takes two parameters, an *FM-index* data structure, a string and returns the locations of occurrences of the string in the data structure
16: **Append** $(kmer_{prev}, i, locations)$ to $retList$
17: $i \leftarrow i + j - 1$
18: **end if**
19: **end for**
20: **return** retList
21: **end function**

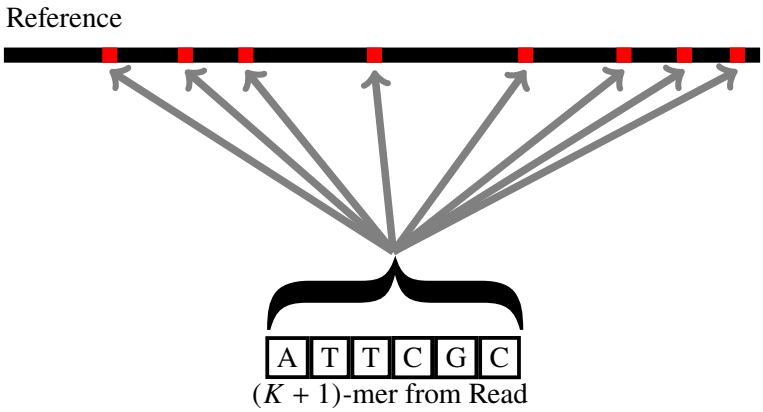

Figure 3.6: Extending One Base in K-mer From Read in Figure **??**, $(K + 1)$-mer is Found. The Locations of This $(K + 1)$-mer in the Reference is Reduced.

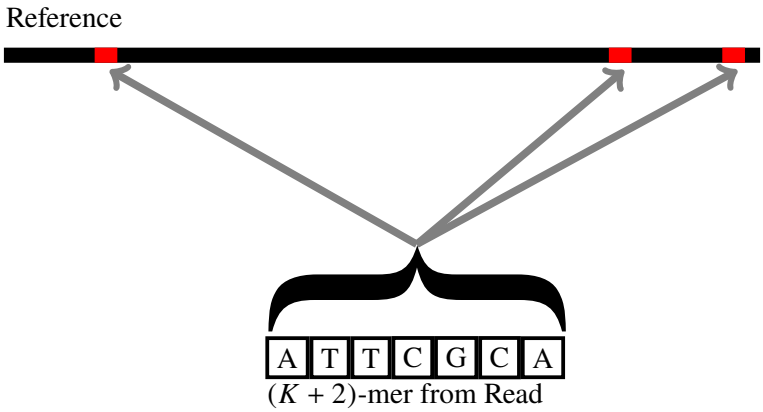

Figure 3.7: Extending One More Base From Figure **??**, $(K+2)$-mer is Constructed. The Locations of This $(K + 2)$-mer in the Reference is Reduced And Now It is Only 3.

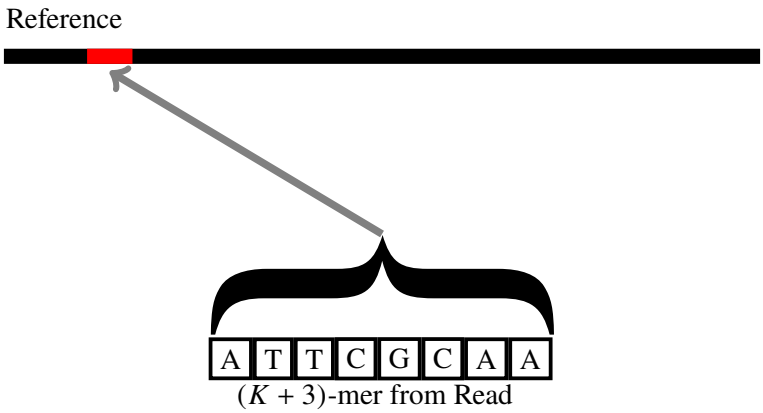

Figure 3.8: Continuing the Extension, From Figure **??** $(K + 3)$-mer is Created. The Count in Reference is Only One.

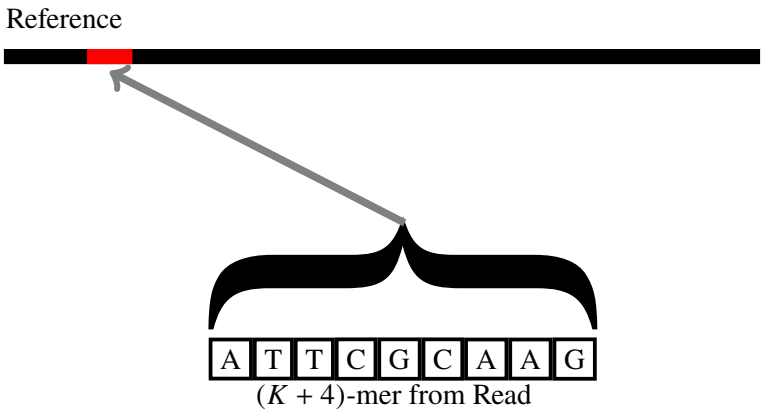

Figure 3.9: From Figure **??**, $(K + 4)$-mer is Made By Appending One Base From Read. It has No Consequence in The Count in Reference.

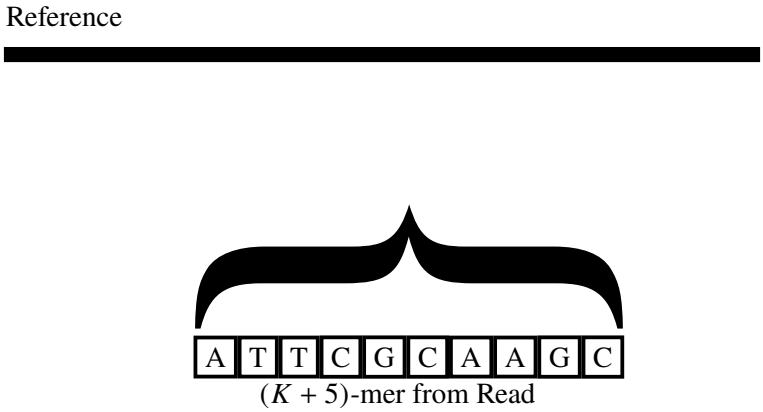

Figure 3.10: One Base Extension in $(K + 4)$-mer of Figure **??** Generates $(K + 5)$-mer. It Shows There is No Existence of Such $(K + 5)$-mer in Reference. So, the Locations Got From Figure **??** Would be Considered as Final.

work of the discussed loop could be handle efficiently inside those functions and the loop would disappear from the function of mapping. Algorithm **??** shows it in a better way.

Algorithm 3 Mapping K-mers of Variable Lengths of a Read to Reference Using Enhanced FM-index

Input: K_{min} is the minimum length of K or the starting length of K, $fmIndex$ is the indexed reverse of reference which is basically a data structure, $read$ is the particular Read Sequence.
Output: A list of K-mers with their position in read and reference.

```
1: function MapWithEnhancedFM(fmIndex, read, minK)
2:     Local Variables: i, j, readLength, i_adjusted, locations, retList
3:     readLength ← |read|                          ▷ |s| means length of the string s
4:     retList ←null
5:     for i ← 0 to readLength − K_min do
6:         i_adjusted ← readLength − i − 1
7:         locations ← MyLocateFM(fmIndex, read, i_adjusted, &j)
               ▷ MyLocateFM function takes four parameters, an FM-index data structure, a
                 string, a starting position and an address to send back the K-mer length. It
                 returns the locations of occurrences of the longest non-zero occurrences K-mer
                 in the data structure from the position i
8:         if |locations| not equal to 0 then
9:             Append (read[i : i + j], i, locations) to retList   ▷ s[a : b] means sub-
                                                                     string consisting of char-
                                                                     acters from index a(inclusive)
                                                                     to index b(exclusive) of
                                                                     string s
10:            i ← i + j − 1
11:        end if
12:    end for
13:    return retList
14: end function
```

3.6 Gapped Minimizer and BWT FM-index Merged Approach

After introducing plenty of options by minimizer and reducing error by gap, there still remains a lot of options to decide the actual position of the reads. On the other hand, BWT FM-index returns more reliable answer. So, merging all of these techniques may give a good result. This approach includes noise handling, double checking of a K-mer and gives a long reliable K-mer to map the read successfully.

Figure **??** illustrates the whole procedure. At first the minimizers from the whole reference is indexed running a window using `unordered_map` in C++ which has $O(1)$complexity. Every

minimizer should have gap in it's third base. In fact, every third base of the the reference genome is also replaced by a gap. The ultimate benefit of adding gap is it reduces the number of mismatches due to noise. Say, two K-mers are `GATTCATGCTTGCA` and `GATACTTGATCGAA` for $K = 14$ where 5 mismatches exist. Inserting gap in every third base makes them `GA_TC_TG_TT_CA` and `GA_AC_TG_TC_AA` which has now 3 mismatches. So, inserting gap reduce mismatches.

In the next step, the gapped reference is indexed using BWT FM-index. Then, the processing of reads are started. A window of K runs over each read. At first, gaps are inserted in every third base of K-mer. Then it is searched in the index of minimizer first. If it occurs as minimizer in the reference, then one base is extended every time and is searched in the FM-index. The last non-zero occurrences are taken in to account. These locations of occurrences are added in the result set and the window of K jumps there after. In this way, all reads are processed.

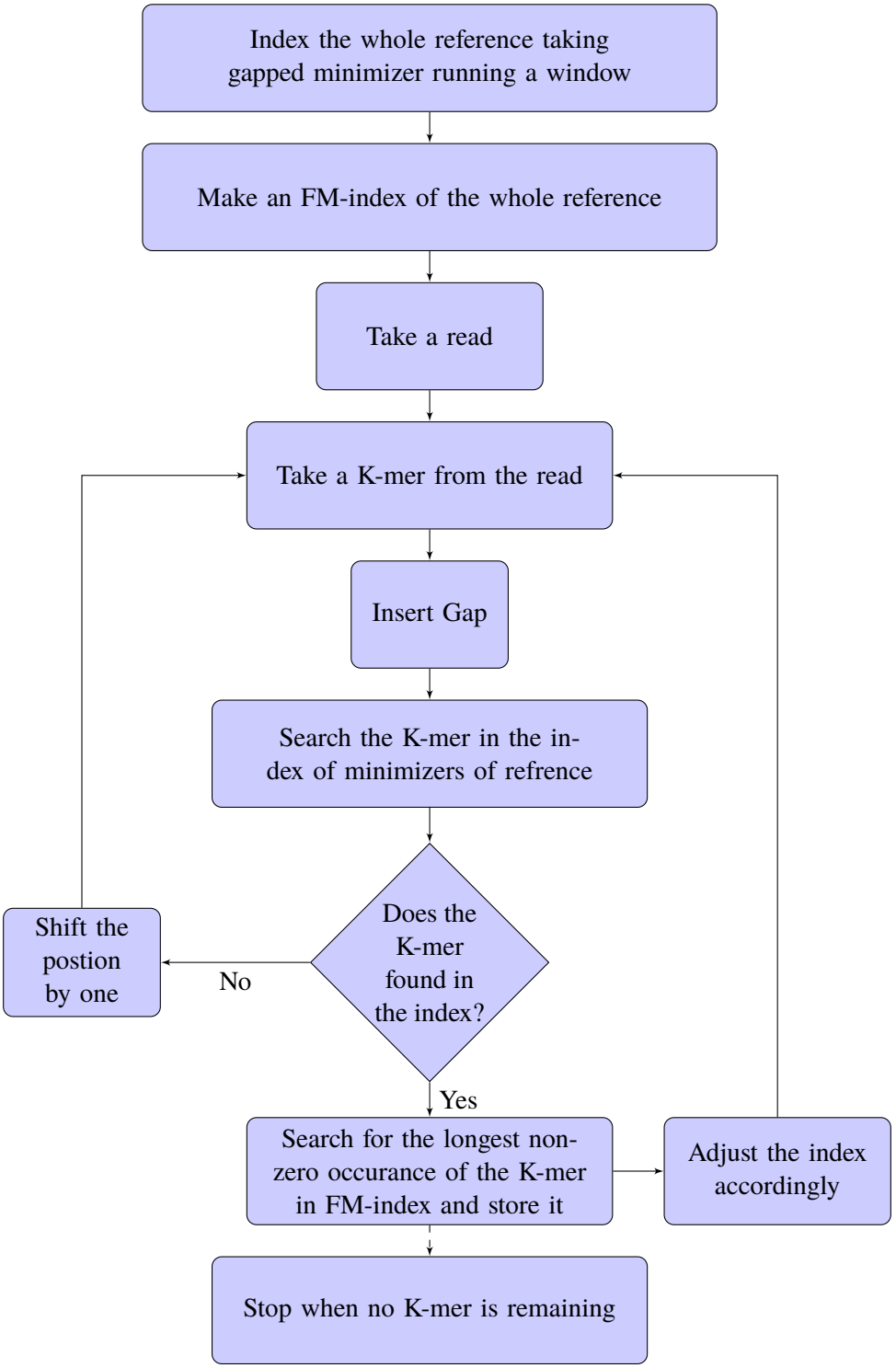

Figure 3.11: The whole process of gapped minimizer and BWT FM-index Merged Approach

Chapter 4

Data Sets

4.1 Data Sources

There we used mainly two data sets for reference and three data set for read. Among them, one synthetic reference data set and one read data set is generated by us randomly. Our generated data set for read contains fifty read sequences having length ten thousands each. Our reference is treated as *Synthetic Reference* and reads are treated as *Synthetic Reads* in the context.

As a real data, we have taken *GenBank: U00096.1* version of *Escherichia coli K-12 MG1655 complete genome*[**?**] which is freely available online. We have two read data sets for this reference genome. One is simulated by Ruhul Amin Shajib while developing NanoBLASTer [**?**]. We called it *20K Simulated Reads* or simply *20K Reads* during the whole work and writings. The other data is treated as 25K reads which is collected from a source of data used in Minimap[**?**]. The table **??** shows a general statistics about the data sets.

Table 4.1: Summary of Data Sets Used in The Experiments.

No	**Name of Data Set**	**# of Reads**	**# of Usable Reads**	**Total Length of Reads**	**Average Length of Reads**	**Reference Name**	**Length of Reference**
1	20K Simulated Reads	20000	17043	118335765	6943.36	*E. Coli*	4639211
2	25K Reads	25970	25970	216906558	8352.2	*E. Coli*	4639211
3	Synthetic Reads	50	50	500000	10000	Synthetic	493290

4.2 Error Rate in Data

20K Simulated Reads data has some *unaligned* reads which we totally ignored in experiment. This data set has 15% to 45% error in data. *20K Reads* data has up to 15% error. On the other hand, our *Synthetic Reads* data has no error as we randomly picked a chunk of 10K base at a time. However, we have added a limited number of random errors which have described in the corresponding context.

Chapter 5

Experiment and Result

Following the previous chapter the implementation and the findings would be shown in this chapter. For every section in the previous chapter, there would be a separate section in this chapter. The state and result of the tool after working to a specific approach is shown in every sections below.

5.1 Implementation and Result of Naive Minimizer Approach

KMC2[**?**], is a K-mer counting tool which inspired us first to think about minimizer. Later, we have found that Minimap[**?**] also used this technique with some enhancement in their tool. However, we could achieve almost same result in perspective of time and memory. Rather ours giving better mapping result for error free data. Here is the snippet from the result summary:

Error Free Data

```
Reference length : 493290
Read length : 10000
Index Building time :2.39232 seconds
Total : 2285 and 1 minimizers not found in Reference.
(Max) 16830 to 26830 : 2298
(Min) 204709 to 214709 : 4
Total : 1581 minimizers only present inside 16807 to 26806
```

```
Total : 701 minimizers present both in and out of 16807 to 26806
Total : 2 minimizers present out of 16807 to 26806
Total Processing Time with Index Building: 2.43698 seconds
```

There was some bug in our code that produced some extra minimizers than expected. However, we found the bug when we shifted to the efficient one. So, we did not wasted time to see find the exact time and minimizer. Here, the summary showed for only one read and the whole synthetic reference genome of length around 0.5 million.

Data Interpretation

- The length of the reference is 493290.
- The length of the only read is 10K.
- It took 2.4 seconds to build the index called map in C/C++.
- There are total 2285 minimizers in the read. Among them only one is not found in the reference.
- 1581 minimizers (69.2%) are found only in the range from where the read is cut.
- 701 minimizers (30.7%) are found both inside and outside of the corresponding range in reference.
- 2 minimizers (<0.001%) are found only outside but not inside of the above range.
- 2298 and 4 are maximum and minimum count of minimizers in a 10K length window in reference.

Discussion From the above statistics, it is clear that the read could be mapped to the actual position in the reference as the majority of the minimizers found in the right range. Note that, here no error is inserted. Read data is same as in the reference. 5

With 5% Error

```
-----------------------------------------------------
Reference length : 493290
Read length : 10000
3.12671
Total : 2264 and 638 minimizers not found in Reference.
Total : 1015 minimizers only present inside 16807 to 26806
Total : 443 minimizers present both in and out of 16807 to 26806
Total : 168 minimizers present out of 16807 to 26806
3.14251
-----------------------------------------------------
```

Data Interpretation

- The length of the reference is around 0.5M.
- The length of the only read is 10K.
- It took 3.13 seconds to build the index called map in C/C++.
- There are total 2264 minimizers in the read. Among them 638 (28.2%) is not found in the reference.
- 1015 minimizers (62.4%) are found only in the range from where the read is cut.
- 443 minimizers (27.2%) are found both inside and outside of the corresponding range in reference.
- 168 minimizers (10.33%) are found only outside but not inside of the above range.

Discussion It showed that about one third minimizer gone from the list of minimizer in the reference. Those which are found still have the majority inside the range. So, the read is still could be mapped in the reference in an approximate position with a limited sensitivity.

With 10% Error

```
-----------------------------------------------------
Reference length : 493290
Read length : 10000
3.07263
(In the range ) Min Diff is : 1 and Max Diff is : 115
(Out of the range ) Min Diff is : 1 and Max Diff is : 3983
Total : 2252 and 1040 minimizers not found in Reference.
Total : 640 minimizers only present inside 16807 to 26806
Total : 309 minimizers present both in and out of 16807 to 26806
Total : 263 minimizers present out of 16807 to 26806
3.07353
-----------------------------------------------------
```

Data Interpretation

- The length of the reference is around 0.5M.
- The length of the only read is 10K.
- It took 3.07 seconds to build the index called map in C/C++.
- There are total 2252 minimizers in the read. Among them 1040 (46.2%) is not found in the reference.
- 640 minimizers (52.8%) are found only in the range from where the read is cut.
- 309 minimizers (25.5%) are found both inside and outside of the corresponding range in reference.
- 263 minimizers (21.7%) are found only outside but not inside of the above range.
- 2298 and 4 are maximum and minimum count of minimizers in a 10K length window in reference.

- Minimum Difference among the corresponding windows of founded minimizers is 1 in both inside and outside the cutting region of read in the reference.
- Minimum Distance among the corresponding windows of founded minimizers is 115 in the range and 3983 outside the range.

Discussion Almost half of the total minimizers in the read disappeared from the reference. Those are found in the range is dominant till now. Approximation may vary in certain range. But it might turn to worse result in case of the long genome like human genome.

With 15% Error

```
-----------------------------------------------------
Reference length : 493290
Read length : 10000
3.11769
Total : 2267 and 1281 minimizers not found in Reference.
Total : 467 minimizers only present inside 16807 to 26806
Total : 214 minimizers present both in and out of 16807 to 26806
Total : 305 minimizers present out of 16807 to 26806
3.13327
-----------------------------------------------------
```

Data Interpretation

- The length of the reference is around 0.5M.
- The length of the only read is 10K.
- It took 3.11 seconds to build the index called map in C/C++.
- There are total 2267 minimizers in the read. Among them 1281 (56.5%) is not found in the reference.
- 467 minimizers (47.4%) are found only in the range from where the read is cut.

- 214 minimizers (21.7%) are found both inside and outside of the corresponding range in reference.
- 305 minimizers (30.9%) are found only outside but not inside of the above range.

Discussion As one third of the founded minimizers are outside the range, the approximation would be quite tough and rough. Here, no minimizer has extra priority. So, things are becoming tougher.

With 20% Error

```
-----------------------------------------------------
Reference length : 493290
Read length : 10000
3.13197
Total : 2249 and 1467 minimizers not found in Reference.
Total : 311 minimizers only present inside 16807 to 26806
Total : 140 minimizers present both in and out of 16807 to 26806
Total : 331 minimizers present out of 16807 to 26806
3.14647
-----------------------------------------------------
```

Data Interpretation

- The length of the reference is around 0.5M.
- The length of the only read is 10K.
- It took 3.13 seconds to build the index called map in C/C++.
- There are total 2249 minimizers in the read. Among them 1467 (65.2%) is not found in the reference.
- 311 minimizers (39.8%) are found only in the range from where the read is cut.

- 140 minimizers (17.9%) are found both inside and outside of the corresponding range in reference.
- 331 minimizers (42.3%) are found only outside but not inside of the above range.

Discussion All domination are almost gone. The approximate mapping might lead in wrong direction.

With 45% Error

```
-----------------------------------------------------
Reference length : 493290
Read length : 10000
3.12898
Total : 2146 and 2141 minimizers not found in Reference.
Total : 2 minimizers only present inside 16807 to 26806
Total : 2 minimizers present both in and out of 16807 to 26806
Total : 1 minimizers present out of 16807 to 26806
3.20383
-----------------------------------------------------
```

Data Interpretation

- The length of the reference is around 0.5M.
- The length of the only read is 10K.
- It took 3.13 seconds to build the index called map in C/C++.
- There are total 2146 minimizers in the read. Among them 2141 (99.8%) is not found in the reference.
- 2 minimizers (40%) found only in the range from where the read is cut.
- 2 minimizers (40%) found both inside and outside of the corresponding range in reference.
- 1 minimizers (20%) found only outside but not inside of the above range.

Discussion Approximation is almost impossible. If minimizers remain in the reference but in bad with a bad placement, then some biasness might work. But almost no data is found. So, the condition reached it's worst.

5.2 Experiment with gapped Minimizer and result

For this experiment we used Escherichia coli K-12 MG1655 complete genome (version U00096.1) as the reference sequence. For read sequence we used three dataset.

1. 20K simulated reads having average read length 7K bases
2. Above 25K PacBio reads having average read length 8K bases
3. Our generated 50 synthetic reads cut from synthetic reference having read length exactly 10K bases

From the 20K simulated reads have a name format like below

```
>Ecoli_1372212_aligned_3035_F_1_13638_30
```

Here the number 1372212 means this read was cut from the reference from position 1372212 having a length of 13638.

As we have ran our methods as mentioned in the methodology section in the 20K simulate read and with afore mentioned reference genome sequence we got results like blow

```
Ecoli_2753889_aligned_2957_R_26_6442_7
2753915 2760130 843
2136
Ecoli_3378975_aligned_2958_R_77_3044_22
3379052 3382182 409
1083
Ecoli_2567494_aligned_2959_F_0_3196_11
2567494 2570603 422
1143
```

Here for each read we see two lines are printed. The first line contains three space separated integers. The three numbers significance are

1. First number means start index of the read in the reference
2. Second number means end index of the read in the reference
3. Third number means count of minimizers found between start and end in the reference.

In the second line the integers means the total number of ***K***-mer of the read found as minimizer in the whole reference. For our experiment purpose we only considered the read that were aligned as described in their read name. After generating results from all aligned reads we found the ration of the total number of ***K***-mer of the read found as minimizer in the whole reference to the count of minimizers found between start and end in the reference is 2 to 3. Which meets our second assumption described in the methodology section. Before doing all these experiment we had to generate the minimizer index both for forward reference and the reverse compliment of the reference. The result of minimizer count is shown below

```
Reference Length: 4639211
Time needed to compute all minimizers of Reference : 0.451064
Number of Unique Minimizers:
Forward: 341184
Reverse: 341062
```

From this result we can figure out that on average after every 13 consecutive ***W***=24 mers we get a new minimizer. Which also meet our first assumption.

5.3 Implementation of Mapping Using Naive BWT FM-index and It's Result

Naive BWT FM-index is implemented on all three data sets. The best performance achieved from the *Synthetic Data*. It could map with 100% Accuracy as the data has no error. It took 5628.92 seconds (Around an hour and 34 minutes) to map all 50 reads each having length 10 thousands. *Synthetic* Reference of length 493290 took 0.18 seconds to index the whole genome

and it's reverse complement consuming 0.26 MB of memory. There is generated a summary of the result and below is the result snippet for *Synthetic Data*:

```
>>> INDEX CREATING <<<
Indexing of Reverse Reference:
Length of Reference: 493290
Reference Indexing Time: 0.177091
Memory Consumption in MB:
Forward = 0.124944
Reverse = 0.124959
>>> END - INDEX CREATING <<<

Total Reads = 50
LOCATIONS querying Time in normal procedure: 5628.92
```

20K Simulated Reads took 5659.44 seconds (Around an hour and 34 minutes) to map all the 17043 reads having an average length 6943 BP each. The result of *20K Simulated Reads* goes below.

```
>>> INDEX CREATING <<<
Indexing of Reverse Reference:
Length of Reference: 4639211
Reference Indexing Time: 1.96468
Memory Consumption in MB:
Forward = 1.15484
Reverse = 1.15482
>>> END - INDEX CREATING <<<

Total Reads = 20000
LOCATIONS querying Time in normal procedure: 5659.44
```

The output comes in the following manner:

```
> Ecoli_2753889_aligned_2957_R_26_6442_7
70 - 83 : ACGAAAAATGGAGC (1)
225322
97 - 111 : ATATGATGCAGTTAG (1)
4293648
124 - 160 : GCAGTAATCTTTTATCCAACAATAAAGCTCATCTGCT (1)
1880973
...
```

Here, the line containing an '>' sign indicates the name of the read with some of it's property. Then every two lines means the same thing until the result of next read. The first line shows the range of the K-mer in read, the K-mer itself and then it's count in the reference. The second line shows the positions or locations of the K-mer in the reference.

25K Reads took 11270.9 seconds (Around 3 hours 8 minutes) to map all 25970 reads having an average length 8352 BP each. Here is the outcome snippet:

```
>>> INDEX CREATING <<<
Indexing of Reverse Reference:
Length of Reference: 4639211
Reference Indexing Time: 2.01149
Memory Consumption in MB:
Forward = 1.15484
Reverse = 1.15482
>>> END - INDEX CREATING <<<

Total Reads = 25970
LOCATIONS querying Time in normal procedure: 11270.9
```

For both of the above data *E.Coli* reference genome of length 4639211 is used which took

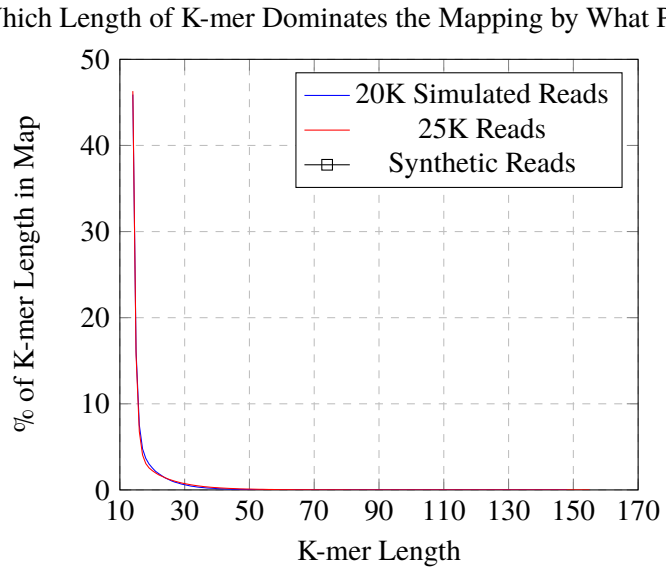

Figure 5.1: For Gradually Increasing the Length of K-mer in Naive FM-index Approach, Domination of Small K-mers is Clear In the Graph. Synthetic Reads has No Room in the Plot, Because It is Considered as Outliers. It Would be A Point at (10000, 100.0) as It has No Error.

around 2 seconds to index the whole genome and it's reverse complement. Only 2.31 MB of memory is needed to build this index.

One noticeable thing here is, the *Synthetic* reads of total length 0.5 million took almost same time to map compared to the *20K Simulated Reads* which is around 237 times larger in length. The fact is, for each of the 50 reads in *Synthetic* data, the program has to loop through K equal to 14 to 10,000. Because all of them are 100% correct and there existence are pretty sure. Figure **??** shows that, for the other data, loop goes to at most 155.

The count of locations of a K-mer for an increasing K would be zero at last. The count of the K-mer just before becoming zero would be treated as final count and locations would be called as final locations. For example, if the count of a (K+1)-mer is zero, then the count of K-mers would be treated as final count. If this final count is one, then the read or K-mer could be mapped at one place. The more the count, the more the options. The more the options, the more the difficulty to align. Figure **??** shows that, the number of K-mers which are mapped to a single position clearly dominating and It is above 96%. So, this method would give a good result, because the read could be mapped to only place.

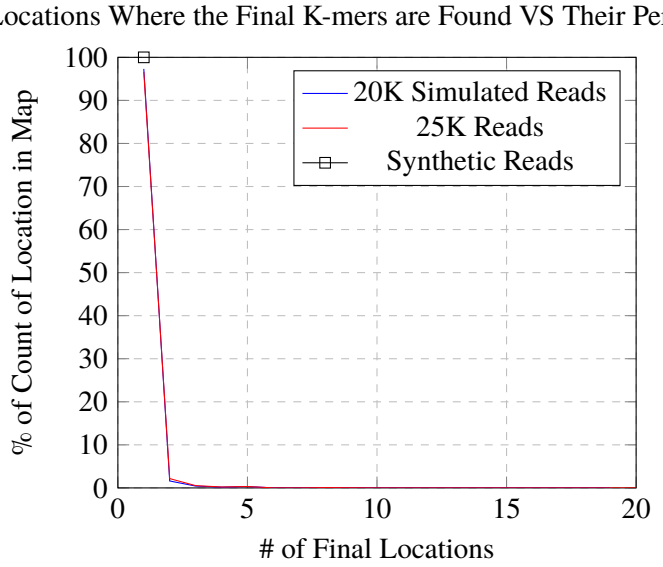

Figure 5.2: Count of Final Locations and Their Percentage among the Whole Final Locations of K-mers.

5.4 Performance of Enhanced BWT FM-index

Enhanced FM-index would work very fast if a long K-mer is found. In fact, this could make more efficient excluding the rest of the computation if a very long or reliable K-mer is found. Figure **??** shows that comparatively data with less error works more efficiently in respect to time. Below is one of the result summary snippet generated by the program.

Result Summary

```
>>> INDEX CREATING <<<
Indexing of Reverse Reference:
Length of Reference: 493290
Reference Indexing Time: 0.177091
Memory Consumption in MB:
Forward = 0.124944
Reverse = 0.124959

Indexing of Reference:
```

Execution Time Comparison Between Naive FM-index Approach and Enhanced FM-index Approach

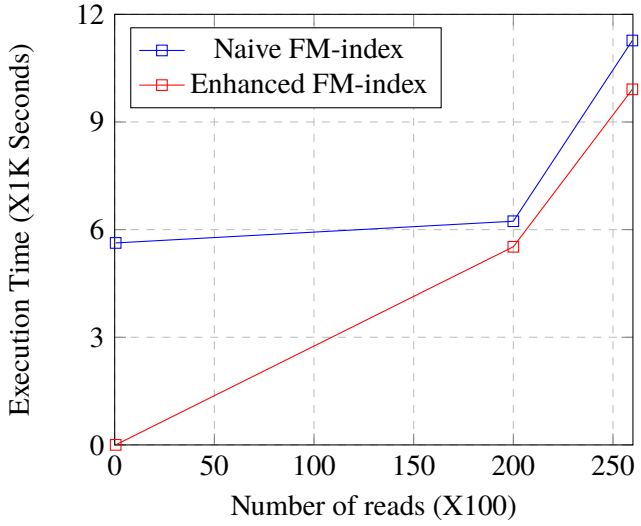

Figure 5.3: For error free data, enhanced FM-index working very fast than naive FM-index. For noisy data, the time gap is reducing.

```
Length of Reference: 493290
Reference Indexing Time: 0.173042
Memory Consumption in MB:
Forward = 0.124951
Reverse = 0.124936
>>> END - INDEX CREATING <<<
Total Reads = 50
LOCATIONS querying Time in normal procedure: 5628.92
LOCATIONS querying Time in modified procedure: 1.1666
```

Data Interpretation

- The length of the reference is 0.5M.
- Indexing time of the reference for naive FM-index is 0.17 seconds.
- Indexing time of the reverse reference for the enhanced FM-index is 0.18 seconds.
- For every type of indexing there needs to create two index, one is the main genome and other is the reverse complement of it. Each type of indexing taking 0.125 MB memory.

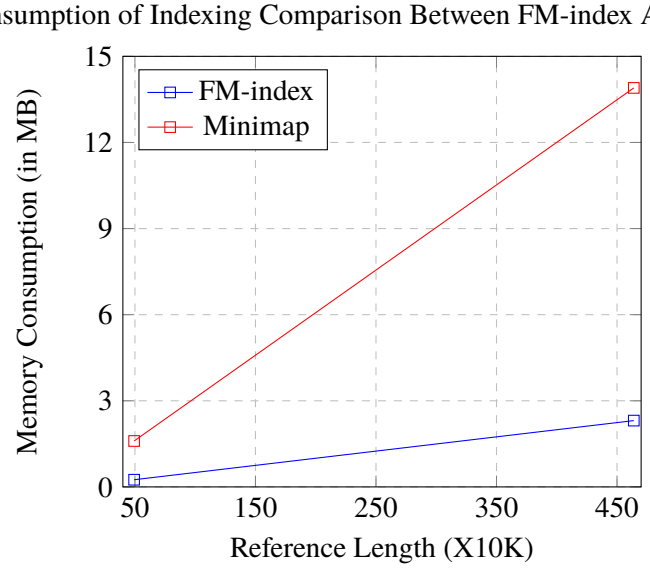

Figure 5.4: Reference Indexing Memory Consumption Comparison between FM-index Approach and Minimap.

- Total Read Count is 50.
- Naive FM-index took 5628.92 seconds to process all of the reads.
- Enhanced FM-index took only 1.17 seconds to process all of the reads.
- For enhanced FM-index, we need to index the reference reversely. From the above report, it is clear that this does not makes any difference with respect to memory and time.

Indexing Reference

Both of the indexes of the reference take same resources with respect to time and memory. However, The memory consumption by the index is lower in these approach comparing with the most recent tool minimap[**?**]. Figure **??** shows that minimiap takes at least six times memory comparing to the both type of indexing.

Table 5.1: Summary of Processing Reference genome

No	Reference Name	Reference Length	# of Minimizer	Indexing Time(Mini.)	Indexing Time(FM)	Memory Usage(MB)
1	E.Coli	4639211	682246	0.46	*1.76*	1.67
2	Synthetic	493290	12225	0.04	*0.16*	0.18

Table 5.2: Summary of Processing Read Sequences

No	Name of Data Set	Total Length of Reads	Time to Map (Naive)	Time to Map (Enhanced)
1	20K Simulated Reads	118335765	*19538.9 (5h 26m)*	*17051 (4h 44m)*
2	25K Reads	216906558	*9408.94 (2h 37m)*	*9869.78 (2h 45m)*
3	Synthetic Reads	500000	*3867.97 (1h 5m)*	*0.90*

5.5 Experiments and Outcomes from Gapped Minimizer and BWT FM-index Merged Approach

The summary of the processing is represented in figure **??** and **??**. From figure **??**, it is clear that memory consumption of the indexing is very low. Further comparison would be shown on the basis of this state of the tool which would be described in the next section. Below is the snippet from output file:

```
> Ecoli_2753889_aligned_2957_R_26_6442_7
1 - 15 : AC_AG_TT_AG_GC_ (3)
3139245 2334798 378942
16 - 30 : CT_TG_GA_TG_TT_ (2)
1812027 2551200
...
```

The fist line is the name of the read with necessary information. Every pair of lines till the next read name contains mapping information. Till now, the formatting of the information is like below:

```
<start_index_in_read> - <end_index_in_read> : <K-mer_with_gap> (<count_of_K-mer_in_refere
<space_separated_locations_of_K-mer_in_reference>
```

Table 5.3: Mapping Comparison While 5% Error Added in Read Data

Name of the Tool	Right Position (%)	Wrong Position (%)
BWA-MEM	88.4	11.6
NanoBLAST	86.55	13.45
NanoMapper	97.42	2.58

Every statement in '<>' briefly describes what would be there. The number of integers in location would be same as the count of K-mer in reference.

5.6 Comparison With Other Tools

With error free data, NanoMapper works best. It could map exact positions and map only one position where other tools give some worng positions along with the right position. However, when error chunk added in the data, NanoMapper performs well. As mapping to wrong position could be misleading, a penalty should be added for this. It is not possible to check every base is mapped to the right position or not for certain reasons. In fact, tools map overlapped ranges. Considering all of these restrictions, the following equations are used to evaluate the accuracy.

$$\textbf{Right \%} = \frac{\textbf{\# of Base Mapped to Right Range}}{\textbf{Total \# of Mapped Base}}$$

Similarly,

$$\textbf{Wrong \%} = \frac{\textbf{\# of Base Mapped to Out of Right Range}}{\textbf{Total \# of Mapped Base}}$$

With 5% error chunk added in the synthetic data, the result is like table **??**.

If we just eliminate mapping length less than or equal to 15, the accuracy reaches 99.17%. In fact, these should be eliminated as these are not our target K-mer. We are mapping based on long K-mers. However, from figure **??**, it is clear that NanoMapper always maintaining a consistant accuracy where other tools are showing curly curves. The consistent line drift upward when we eliminate K-mers less than or equal to length 15.

When 10% error added in the read, the performance of NanoBLAST became a little bit better than before. Table **??** shows that With removing K-mers $\leq$ 15, NanoMapper still outperforming any other existing tools. From the figure **??**, it could be noticed that for the read #15, NanoMapper's

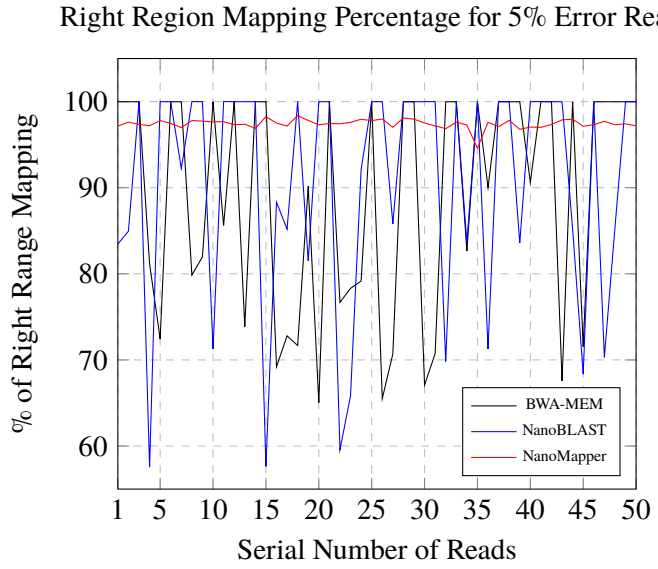

Figure 5.5: Percentage of Right Range Mapping Comparison Among Tools.

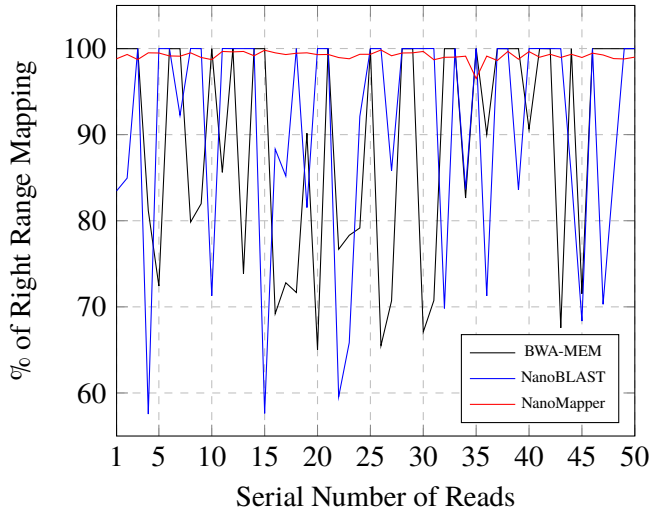

Figure 5.6: Percentage of Right Range Mapping Comparison Among Tools After Removing K-mer Having Length Less Than or Equal to 15.

Table 5.4: Mapping Comparison While 10% Error Added in Read Data Eliminating K-mer Having Length ≤ 15.

Name of the Tool	Right Position (%)	Wrong Position (%)
BWA-MEM	88.82	11.18
NanoBLAST	88.72	11.28
NanoMapper	99.02	0.98

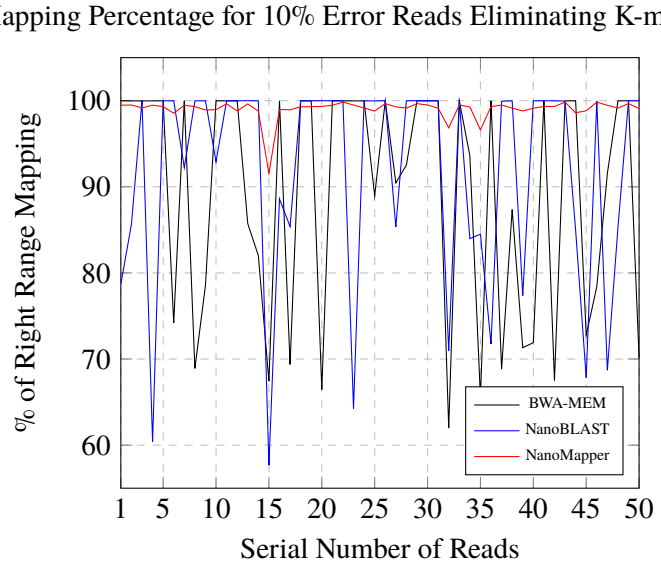

Figure 5.7: Percentage of Right Range Mapping Comparison Among Tools After Removing K-mer Having Length Less Than or Equal to 15 For 10% Noise In Data.

consistency degraded. But for that case, both the other tools give worse performance.

Chapter 6

Conclusion

The remarkable notes are briefed below. Another section describe how the tool could be made more useful for the world and could be served in greater extent.

6.1 Discussion

NanoBASTer[**?**] of Mohammad Ruhul Amin, BWA[**?**] and BWA-SW[**?**] of Heng Li, Bowtie[**?**] and Bowtie2[**?**] of Ben Langmead are good alignment tools. But none of them could be selected and announced as the best to use in general. Minimap[**?**] of Heng Li also a recent tool which is developed for mainly assembly tool like miniasm[**?**] of him. From their work, we learnt a lot of thing. Before this, the concept of Minimizer[**?**, **?**], Bloom filter[**?**, **?**], Hash Table[**?**, **?**], Count-min Sketch[**?**], Locality Sensitive Hashing[**?**] etc. helped us a lot to think from different perspective.

6.2 Future Work

6.2.1 API Enhancement

Our used API for FM-index, SDSL-lite[**?**], is not that fast comparing with other existing tools. This implementation is a general purpose API. So, tweaking for only four letter in the alphabet could be done.

6.2.2 Integration with NanoBLASTer

One of the drawbacks of NanoBLASTer[**?**] is it could not recognize long insertion or long deletion. So, it is the best implementation to enhance the performance and reduce the problem. As the input output format and parameters in functions disagree between NanoBLASTer and this tool, the integration has not been done. With some effort, it could be done easily to get output in greater extent.

6.2.3 Testing

As our datasets and resources were limited we could not test the tool vastly. Massive testing would build a set of statistics based on which some heuristics might be added to get better output.

6.2.4 Developing New Aligner

Based on this mapping tool, a new aligner tool could be made for some specific project. This may take lot of time to build such one, but it would be effective if built once. Enhanced alignment like Gotoh[**?**,**?**] and other could be developed and run parallel to this which may give a good output.

Currently this tool generates some

```
<variable_length_kmer>  <pos_in_read>  <pos_in_reference>
```

triplets for each read. Then to map the whole read to the desired segment of the reference we would need to cluster the triplets such that the maximum cluster will be formed with the maximum count of minimizers that maintains order both in read and reference.

Many type of clustering technique has been used in different tools. For example Miniasm-Minimap[**?**] uses a clustering technique that is inspired from Hough Transformation[**?**,**?**] a feature extraction technique used in Digital image processing, MUMmer uses their own clustering technique.

Appendices

Appendix A

Setting Up NanoMapper

A unix based operating system is needed to run this program. Please ensure that the system fulfilled the following requirements:

- The latest version of of C++ compiler. C++11 standard or later. clang version must be 3.2 or higher.
- A cmake build system.
- The operating system must be a 64-bit machine.
- zlib library installed.

A.1 API

The SDSL-lite[**?**] API is used to implement FM-index. It could be installed by the following command lines in the terminal:

```
$ git clone https://github.com/simongog/sdsl-lite.git
$ cd sdsl-lite
$ ./install.sh
```

The API could be un-installed after working by the following command having same directory in the terminal:

```
$ ./uninstall.sh
```

A.2 Installing NanoMapper

The following commands in terminal would let you install NanoMapper in your PC:

```
$ git clone https://github.com/enamcse/NanoMapper.git && cd NanoMapper
$ g++ -o NanoMapper -std=c++11 -O3 -DNDEBUG -I ~/include -L ~/lib
  Main_minimizer.cpp -lz -lsdsl -ldivsufsort -ldivsufsort64
```

A.3 Running NanoMapper

The NanoMapper could be run on your PC giving the parameters like below:

```
$ ./NanoMapper <path_of_reference_file> <path_of_reads_file>
  <K-mer_length> <window_length>
  <output_file_path_for_naive_fm_index_approach>
  <output_file_path_for_enhanced_fm_index_approach>
```

Example:

```
$ ./NanoMapper synthetic_reference_seq.fasta gen_seq.fasta 14 24
  loc_syn_out.txt loc_syn_out_1.txt
```

Note that, each command should be given in one line. Here, it is shown in multi-line for simplicity.