OPTIMUM SEARCH SCHEMES FOR APPROXIMATE STRING MATCHING USING BIDIRECTIONAL FM-INDEX

A Dissertation

by

BAHMAN TORKAMANDI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Kiavash Kianfar
Committee Members,	Sergiy Butenko
	Alan Dabney
	Alfredo Garcia
Head of Department,	Lewis Ntaimo

May 2020

Major Subject: Industrial Engineering

ABSTRACT

The objective of the research in this dissertation is to derive optimal search schemes for approximate string matching using bidirectional FM-index, and utilize them in increasing the speed of such searches. Such a problem arises in computer science with many applications. Approximate string matching problem is also central in bioinformatics where biologists are interested in aligning pieces of DNA back to genome. Given a text, the search for a given pattern can be accelerated by preprocessing the text through constructing a hash table or indexing the text. Bidirectional indices have opened new possibilities by allowing a search to start from anywhere within the pattern and extend in both directions. In particular, use of search schemes (partitioning the pattern and searching the pieces in certain orders with given bounds on errors) can yield significant speed-ups. However, finding optimal search schemes is a difficult combinatorial optimization problem. Prior work tends to use search heuristics but lacks the ability to find the best strategies for using an index to search for a pattern. In this dissertation, we will find the optimal search scheme for approximate string matching problem for a bidirectional index with the assumption of having the number of partitions. Moreover, we will investigate the computational gain from applying these optimal search schemes to search in a bidirectional FM-index.

Intellectual Merit. First, we propose an MIP formulation to find the optimal search scheme for approximate string matching problem using a bidirectional index under Hamming distance error. Second, we demonstrate that our MIP can solve the optimum search scheme problem to optimality in a reasonable amount of time for input parameters of considerable size, and enjoys very quick convergence to optimal or near-optimal solutions for input parameters of larger size. Third, we show that approximate search in a bidirectional FM-index can be performed significantly faster if the optimal schemes obtained from our MIP are used. This is demonstrated based on number of edges in the search tries as well as actual running time of in-index search for Illumina DNA Sequencing reads (up to 35 times faster than standard backtracking for 3 errors). Although our MIP solutions are for Hamming distance, they perform equally well for edit distance. Fourth, we demonstrate

that our optimal search schemes is superior to the best of in-index aligners for 2 and 3 errors. In an attempt to acquire a glimpse of the potential of combining our optimal search schemes with in-text verification, we combine optimal search scheme and in-text verification for Hamming distance. This experiment halved the running time for reads of size 101 and 125. Furthermore, we showcase the power of our optimal search schemes by demonstrating that for 1 to 3 errors, approximate string matching of reads of size $40,\ 101$, and 125 performed completely in index compete in running time with the best full-fledged aligners, which benefit from combining search in index with in-text verification for edit distance. Moreover, we will relax the assumption of having equal size partitions in our MIP and address the more general form of approximate string matching problem where the only assumption is the prespecified number of partitions. We will present an MIP formulation for edit distance and provide an alternative formulation for Hamming distance.

Broader Impacts. The results of this research promise a significant increase in speed of finding approximate occurrences of a pattern in a text. This is an important problem with many applications in bioinformatics and computer science such as recovering text in signal processing and information retrieval [23]. Approximate string matching plays an indisputable role in the realm of bioinformatics, where any downstream analysis on the genomic data starts with aligning sequenced DNA or RNA reads back to a reference genome. Technologies such as next generation sequencing has produced considerable amount of data leading to increasing demand for fast read aligners to map DNA pieces to genome. In order to solve this central problem, one could consider the genome of any species of interest as the "text" and the sequenced pieces of DNA as the "patterns" and therefore search for approximate occurrences of a pattern in a text using a full-text index. Some tolerance for errors is required due to mutations in genome of each individual organism such as single nucleotide variants (SNVs) as well as errors in sequencing technologies. This broad spectrum of applications indicates the significant impact of this research on many areas of health and life sciences and practice, where discovery, diagnosis, and treatment all depend on genome sequencing.

DEDICATION

To my dear family

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Kiavash Kianfar. This dissertation would have been impossible without his guidance. His critique of my work has developed me as an individual and as a professional. Research discussions with him have been very intriguing and has molded me into a better researcher, a critical thinker and helped me enjoy research. I would also like to thank my committee members Dr. Butenko, Dr. Dabney Yu and Dr. Garcia for their invaluable comments and suggestions. My appreciation goes to the Department of Industrial and Systems Engineering for pro- viding me with funding throughout this Ph.D. study. This dissertation is dedicated to my family. Finally, I would like to acknowledge Texas A&M University High Performance Research Computing (HPRC) for providing resources to perform parts of computational experiments.

CONTRIBUTORS AND FUNDING SOURCES

Funding Sources

This work has been partially supported by Texas Engineering Experiment Station (TEES) and department of Industrial and Systems Engineering, which are gratefully acknowledged.

NOMENCLATURE

NGS	Next Generation Sequencing
DNA	Deoxyribonucleic Acid
RNA	Ribonucleic Acid
mRNA	messenger Ribonucleic Acid
A	Adenine
C	Cytosine
G	Guanine
T	Thymine
A-T	Adenine-Thymine
G-C	Guanine-Cytosine
TF	Transcription Factor
SNP	Single Nucleotide Polymorphism
InDels	Insertion and Deletions
HTS	High Throughput Sequencing
HGP	Human Genome Project
ENCODE	Encyclopedia of DNA Elements
SE	Single End
PE	Paired End
SAM	Sequence Alignment Map
FM	Ferragina-Manzini
bp	base pair
nt	Nucleotide

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Approximate String Matching (ASM) problem, i.e., finding occurrences of a string called pattern or read in a text allowing for some error tolerance, is a central problem in bioinformatics where biologists are interested in aligning pieces of DNA back to genome.

The ASM problem for Hamming/edit distance is defined as follows: Given a number of mismatches K, a read of length R, and a text of length T, composed of characters from an alphabet of size σ, find a sub-string of the text whose Hamming/edit distance to the read is at most K.

ASM problem has become especially important in bioinformatics due to the advances in sequencing technology during the last years. Bidirectional indices have opened new possibilities by allowing a search to start from anywhere within the pattern and extend in both directions. The objective of the research in this dissertation is to derive optimal search schemes for approximate string matching using bidirectional FM-index.

Optimal Search Scheme (OSS) problem for Hamming distance (mismatch), considered in this dissertation, is defined as follows: What is the search scheme that minimizes the number of sub-strings searched in ASM with Bidirectional indexes (ASM-B) while ensuring all possible mismatch patterns are covered?

While ASM-B for a single read is an easy problem, the OSS problem, which is the focus of this reseach, is a difficult combinatorial optimization problem. There are a large number of attributes that define a solution and a large number of possibilities for each attribute; the solution must satisfy complex combinatorial constraints; and, calculating the objective function, i.e., number of sub-strings in the ASM-B algorithm, for a given solution is complicated.

In order to understand the role of ASM in bioinformatics and the importance of optimal search schemes, we provide a brief overview of DNA, sequencing technologies, and how biological processes can be interpreted as an approximate string matching problem.

1.1 Cellular Processes

DNA (Deoxyribonucleic acid) is a molecule that holds the information required for cellular processes and functions. The nucleus of a cell contains chromosomes, molecules that accounts for the genetic material of a cell. Chromosome is a long thread of DNA. Most of the processes in a cell are carried out by RNA (Ribonucleic acid) and proteins. DNA is a molecule that holds the set of instructions for synthesis of proteins and RNA molecules.

DNA is a double helical structure containing two long polymer chains called DNA strands [2], each of which is composed of four different molecules (bases) called Adenine (A), Thymine (T), Cytosine (C) and Guanine (G). In fact, the information needed for metabolism in a cell is coded into DNA by the arrangement of these four different bases. To understand the biology of a cell we need to determine the composition of genes, portions of DNA that encode for proteins.

1.2 DNA Sequencing

Next generation sequencing technologies such as Illumina sequencing are capable of sequencing fragments of DNA that play a role in protein synthesis. However, this technologies are not able to sequence the whole gene. They can reliably sequence shorter fragments of DNA called reads. In order to find the composition of genes, read mapping software packages align reads back to genome.

In 2003, the technology used to sequence the human genome was based on automated Sanger sequencing (first generation sequensing). The sequencing of human genome revealed the need for more advanced DNA sequencing technologies which led to the development of Next Generation Sequencing (NGS) [8].

NGS is a DNA sequencing technology which has made DNA sequencing much cheaper and faster. NGS is capable of sequencing millions of DNA fragments in a parallel fashion which in turn has generated huge amount of genomics data [3].

The mainstream second generation sequencing techniques like Illumina produce reads of length 150-250 with an error rate of about 1%, mostly substitutions caused by the sequencing technology. Other sequencing technologies, e.g., Pacific Bioscience or Oxford Nanopore, produce

much longer reads but with a higher error rate (in the range of 15%) containing both substitutions and insertions/deletions [9, 10, 11, 12].

1.3 Read Mapping

As mentioned earlier, read is a short sequence of A, C, G and T letters. A standard problem is to map the reads back to a reference genome while taking into account the errors introduced by the sequencing technology as well as those caused by biological variation, such as SNPs or small structural variations. Such a problem is almost always modeled as the ASM problem. Read mappers (aligners) align reads to a standard representative for the genome of a species, called reference genome, within a tolerance range for errors (Hamming or Edit distance errors). There are two main approaches adopted by various aligners:

1. *Filtering*: Filtering narrows down the search space by filtered out regions of reference genome. In lossless methods, it is guaranteed that the read will not align to those regions [13]. Consequently, aligners map reads to the remaining portion of the genome via Hashing and utilizing pigeonhole principle [14].

2. *Indexing*: Indexing is an approach in which the reference genome (a text) is transformed into a data structure such as a suffix tree or its variants. Since indices do not scan the entire DNA, the lookup process is considerably fast. Common indices used in read aligners include suffix array [15], enhanced suffix array [16] and FM-Index [17]. FM-index carries out the search in linear time with respect to the size of the read [13] with a low memory overhead.

Lam et al. [6] introduced *bidirectional* FM indices to speed up ASM for Hamming distance. For the cases $K = 1$ and 2, they partitioned the read into $K+1$ equal pieces, and argued that performing approximate matching on a certain combination of these pieces in a bidirectional index amounts to *faster* approximate matching of the whole read. This combination is such that all possible *mismatch patterns*, i.e., all possible distributions of K mismatches among the pieces, are covered. The main idea behind improved speed is that a bidirectional index not only can start the search from the beginning (or end) of the read, but also from the beginning (or end) of any of the pieces. Therefore, we can start the search from a middle piece and then expand it to the left or right into adjacent

pieces in any order we like. By choosing multiple appropriate orderings of pieces for this purpose, we can perform a much faster ASM compared to a unidirectional search because we can enforce exact or near-exact searches on the first pieces in the partition, significantly reducing the number of backtrackings, while using different orderings of pieces to ensure all possible mismatch patterns are still covered.

Kucherov et al. [7] formalized and generalized this idea by defining the concept of *search schemes*. Assume a read can be partitioned into a given number of pieces, denoted by P (not necessarily equal to $K + 1$). The pieces are indexed from left to right. A search scheme $\mathcal{S} = \{(\pi_s, L_s, U_s), s = 1, \ldots, S\}$ is a collection of S searches, where each search s is designated by a triplet (π_s, L_s, U_s). π_s is a permutation of $1, \ldots, P$ and denotes the order in which the pieces of the partition are searched in search s. If $\pi_{s,i} = j$, then piece j is searched at position i in the order (shortly referred to as *iteration* i in this paper). Due to the way a bidirectional index works, the permutation π_s must satisfy the so-called *connectivity* condition, i.e, a piece j can appear at iteration $i > 1$ in the permutation only if at least one of pieces $j - 1$ or $j + 1$ have appeared at an iteration before i. L_s and U_s each are strings of P numbers. $L_{s,i}$, is the lower bound on the cumulative number of mismatches allowed at iteration i of search s, and $U_{s,i}$ is the upper bound on this value.

1.4 Optimal Search Scheme Problem

Since the development of NGS, the cost to perform DNA test has significantly decreased, which translates into production of gigantic amount of raw data. These data needs to be processed by performing read alignment.

The input of an aligner is millions of reads that need to be aligned along the genome. Although the alignment for two strings is well studied, due to the shear number of reads, mismatch errors, and the extraordinary size of human genome, known alignment algorithms are computationally expensive. In order to address this issue, aligners use indexes and attempt to find the reads in those indexes. In order to find occurrences of a read, we need to traverse search tries that cover all the error patterns. An edge in a trie is correspondent to an step of ASM in FM index. Therefore,

minimizing the computational efforts in ASM-B is equivalent to finding the search scheme with the minimum number of edges while covering all the error patterns. The answer to the *Optimal Search Scheme* problem, can potentially have a great impact on improving the running time of ASM-B resulting in considerably superior read aligners.

The optimal search scheme problem defined in section 1 can now more formally be defined as follows:

Optimal Search Scheme Problem: *Given the number of errors K, the size of reads R, the number of partitions P, and the number of searches $\overline{S}$, what is the search scheme that minimizes the number of edges in search scheme tries while ensuring all possible mismatch patterns are covered?*

OSS outputs search schemes or more precisely, the order that we search π, the minimum errors for partitions L_s, and the maximum allowed number of errors U_s. It is important to note that in OSS, we are not solving the alignment problem, rather, we solve for optimal search schemes to use for the alignment of all the reads.

It turns out that this is a very difficult combinatorial optimization problem due to several reasons: There are a large number of attributes that define a solution (including S, P, size of each piece, and (π_s, L_s, U_s) for each search) with a large number of possibilities for each attribute; the solution must satisfy complex combinatorial constraints; and, calculating the objective function, i.e., number of steps in the ASM-B algorithm, for a given solution is complicated.

Kucherov et al. [7] presented some interesting results contributing initial insight into this key problem. More specifically, they assumed the number of steps in the ASM-B algorithm with a given search scheme, is a constant factor of the (weighted) total number of substrings enumerated by the algorithm in all searches. Assuming that a randomly generated read is to be matched to a randomly generated text, they presented a method to calculate this objective function for a given search scheme. They then showed that unequal pieces in the partition can potentially improve the objective function compared to equal pieces, and presented a dynamic programming (DP) algorithm that for a single prespecified search, with given P and (π, L, U), finds the optimal sizes of pieces assuming that we only calculate the objective function as the total number of substrings

up to a limited length (justified by total randomness of the read and the text); see [7] for more details. In fact, the superiority of this DP over explicit enumeration is only due to this assumption. Nevertheless, this DP is very inefficient, and most importantly, it only finds the optimal piece sizes for a *prespecified* search. In other words, it does not address the problem of finding an optimal search scheme which calls for determining S and all attributes of each search in the search scheme, and ensuring that they cover all mismatch patterns.

Kucherov et al. [7] also presented solutions for another limited problem, i.e., lexicographically minimizing the lexicographically maximal U string (critical U string) in a search scheme, only for $P = K + 1$ or $K + 2$ and assuming that the L strings for all searches contain only zeros. The usefulness of these solutions is justified by the high probability that the search with the critical U string has the largest share in the objective function; see [7] for details. Again from the perspective of finding an optimal search scheme, this result has similar limitations. Only one of the attributes (U) of one of the searches for two specific values of P are optimized by fixing all L strings, which is far from designing a globally optimal search scheme as defined above. Consequently, in their computational experiments, Kucherov et al. [7] use a greedy algorithm based on this limited result to construct search schemes with unknown quality and only optimize the piece sizes for these schemes using their DP.

1.5 Contributions

In this study, for the first time, we have proposed a method to solve the optimal search scheme problem for ASM-B with Hamming distance, for any given P and equal-size pieces. Our method is based on a novel and powerful mixed integer linear program (MIP) that gets K, R, P, and an upper bound on S, denoted by $\overline{S}$, as input, and provides, as its solution, all the attributes of the exact optimal search scheme (MIP methodology for optimization has been addressed in many references such as [18, 19]). To acquire insight on the properties of our MIP, we have presented the results of our computational study on the characteristics of the optimal solution of the MIP and its running time, for different values of its input parameters.

Furthermore, we have performed a search (for Hamming and edit distance) based on the optimal

search schemes obtained from our MIP. This bidirectional index search is implemented in SeqAn [20] and uses a recent fast implementation of bidirectional indices [5] based on EPR dictionaries. We have demonstrated that, for practical ranges of various input parameters, the number of substrings for the optimal search schemes found by our MIP can reduce to as small as half the number of substrings in the unidirectional complete backtracking. To further investigate the potential of our optimal search schemes, we have conducted a search for all occurrences of Illumina reads in the human genome using our optimal search schemes in bidirectional FM-index versus standard backtracking search for $K = 1$, 2, and 3.

In our MIP, we had assumed that all the partitions are of the same size. Kucherov et al. [7] showed that unequal pieces can potentially improve the objective function compared to equal pieces. We have relaxed the assumption of having equal size partitions in our MIP and addressed the more general form of approximate string matching problem where the only assumption is the prespecified number of partitions. We have presented an MIP formulation for edit distance and provided an alternative formulation for Hamming distance.

The drastic improvement over standard backtracking gained by using our optimal search schemes for bidirectional search in index suggests that the performance of read mappers that utilize an index can be significantly improved. To gauge this potential, we even challenged our optimal search schemes by performing a pure index-based search using them and comparing the performance with the full-fledged state-of-the-art aligners that benefit from using a combination of search in index and verification in text using dynamic programming.

Due to the exponential complexity of ASM with respect to K, the state-of-the-art aligners do not perform ASM completely in index but rather use a combination of search in the index and verification in text. Pure index-based search using standard backtracking is very slow for larger values of K. Locating the best point to stop verification in the index and start in-text verification is of high interest. In fact, this can be individually decided for each pattern. In an attempt to acquire a glimpse of the potential of combining our optimal search schemes with in-text verification for edit distance, we test our OSS against full-fledged in-index aligners BWA-aln [21] and Bowtie1 [4].

This comparison shows that optimal search scheme is faster than Bowtie1 for $K = 1, 2$, and 3. In addition, OSS outperforms BWA for all values of K but $K = 1$, for which BWA is slightly faster. This is an indication that combing our OSS and in-text verification will outperform state-of-the-art aligners by speeding up the in-index search. Next, we try to answer the following question: *Can in-text verification alone compensate for the speed up gained through optimal search schemes?* We observe that OSS, without in-text verification, is much faster than backtracking plus in-text verification for all values of $K = 1, 2$, and 3 which shows the performance gained through OSS will remain significant. We also combine optimal search scheme and in-text verification for Hamming distance. This experiment halved the running time for $K = 1,\ 2$ and 3 for reads of size $R = 101$ and 125. We decided to even challenge our optimal search schemes by using them in a pure index-based search and compare the results against the full-fledged state-of-the-art aligners for edit distance error. For the two data sets from human genome and $K = 1$ and 2, OSS outperformed other aligners with the exception of Bwolo for $K = 2$. For $K = 3$, the benefit of using in-text verification in full-fledged aligners catches up and thus outperform OSS which carries out the search entirely in index.

Specifically, Bwolo search tries produce long seeds which reduces the number of dynamic programming performed for in-text verification. Its search tries are basically $01 * 0$ seeds that can be efficiently searched in FM-index. Bwolo search schemes are not optimal in terms of total number of edges but work better when combined with in-text verification. This implies that, in order to design search schemes to be utilized in conjunction with in-text verification, the objective function of an MIP should incorporate the in-index and in-text computational expenses.

This dissertation is organized as follows: after a brief review of the necessary background in section 2, in section 3 we present our MIP for equal size partitions. In chapter 4, we present an alternative MIP for Hamming distance. Then, we will relax the assupmtion of having eqaul size partitions. Later in section 4, we present an MIP for edit distance. Finally in section 5, we present the computational performance and the potential of the optimal search scheme in full-fledged aligners.

2. NECESSARY BACKGROUND

To understand the scope of this research, a brief introduction on cell biology and its important components, the structure of DNA, the sources of errors in DNA, sequencing technologies, and the tools that process genomics data is necessary.

2.1 From DNA to Protein

In general, a human cell consists of a dense membrane-closed structure called the nucleus and other distinct subunits called organelles (Fig. 2.1). The nucleus contains chromosomes, molecules that accounts for a part or all of the genetic material of a cell. Chromosome is a long thread of DNA (Deoxyribonucleic acid) plus other molecules that package DNA and support its structure. Most of the processes in a cell are carried out by RNA (Ribonucleic acid) and proteins. DNA is a molecule that holds the set of instructions for synthesis of proteins and RNA molecules that control functioning and reproduction of cells.

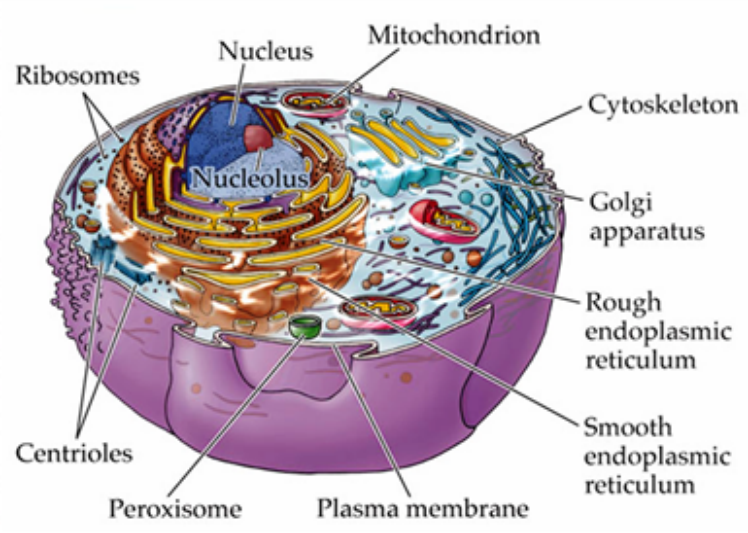


Figure 2.1: Eukaryote cell [22]

The instructions for producing protein molecules are stored in DNA and carried out via tran-

scription and translation processes inside the nucleus and cytoplasm, respectively (Fig. 2.2). In transcription, polymerase molecules make a copy of a piece of DNA (gene) which encodes for a particular protein. The copy molecule is called mRNA (messenger RNA) which moves into cytoplasm where ribosomes synthesize proteins from amino acids using the instructions written in mRNA molecules in a process called translation.

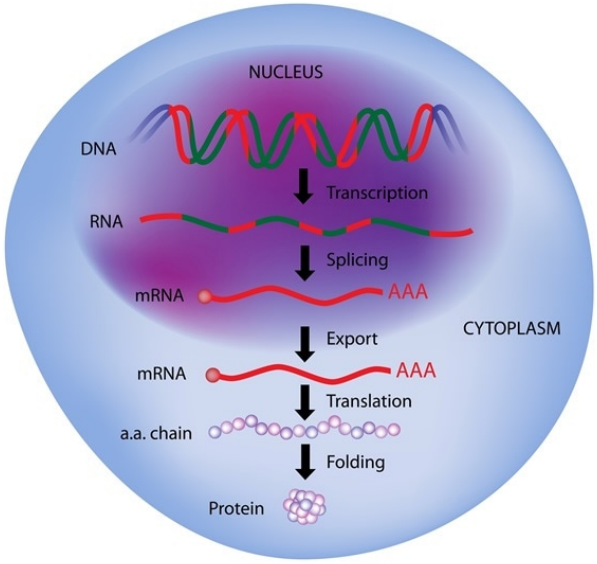


Figure 2.2: Flow of information from DNA to protein through transcription to translation [1]

DNA is a double helical structure (Fig. 2.3) containing two long polymer chains called DNA strands [2], each of which is made of four types of units called nucleotides which are composed of a sugar-phosphate group and a nitrogenous base. The sugar- phosphate group acts as a scaffold to hold the four different nitrogenous molecules called Adenine (A), Thymine (T), Cytosine (C) and Guanine (G). The two strands of DNA are aligned in such way that constructs adenine-thymine (A-T) and guanine-cytosine (G-C) complementary base pairs through hydrogen bonds. Adenine bonds with Thymine because they both need to make two hydrogen bonds to become stable. Similarly, Guanine bonds with Cytosine since they are able to make three stable hydrogen bonds (Fig. 2.3).

As a result one strand of DNA is exactly complementary to the other strand of DNA. In addition, the polarity (direction) of one strand is oriented in the opposite direction to the polarity of the other strand. The polarity (direction) in a DNA strand is shown by referring one end as 5' and the other as 3' end [2]. In fact, the information needed for metabolism, growth, and division of a cell is coded in each strand by the arrangement of the four different bases. This property lets us to consider the genome of a species as a long text composed of four letters. For instance, human reference genome contains 3,257,347,282 nitrogenous bases.

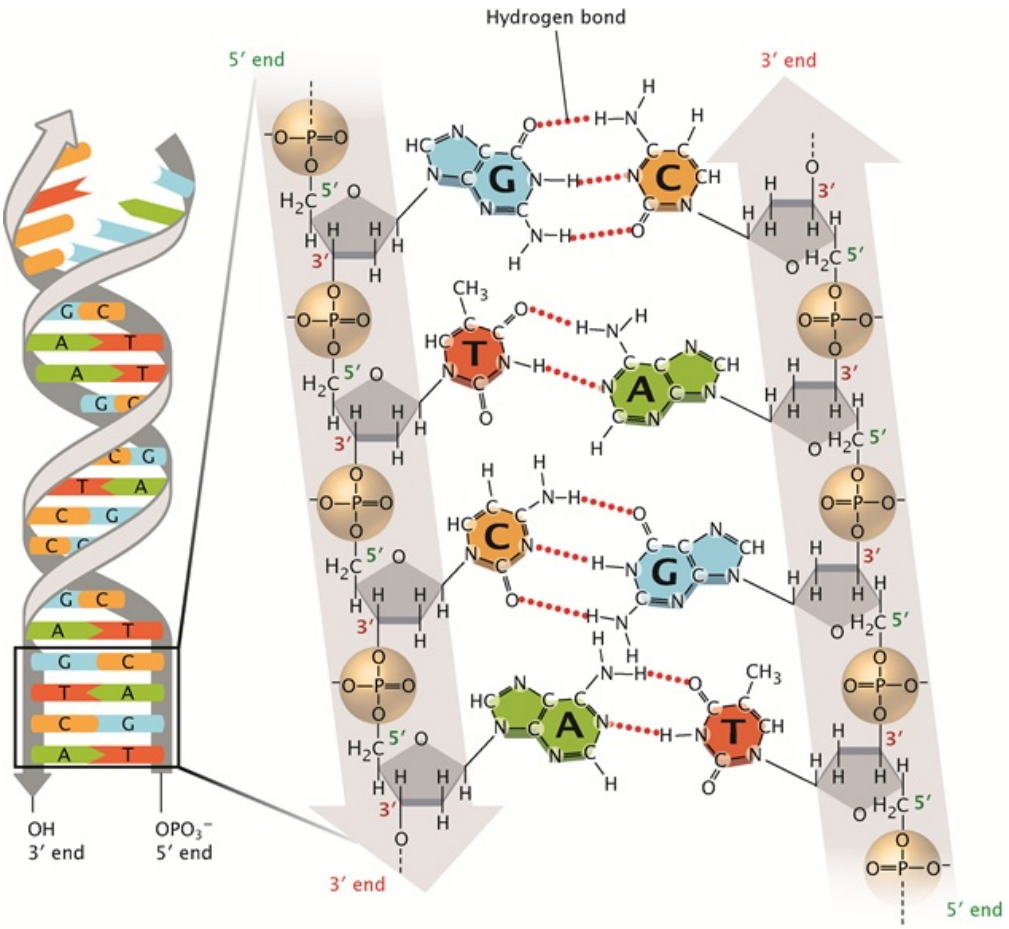


Figure 2.3: The structure of DNA [2]

To understand the biology of a cell and the origin of many diseases we need to determine

the composition of pieces of DNA that play a role in protein synthesis plus their locations in a genome. In addition to finding the composition of important parts of DNA (via next generation sequencing, NGS) we also need to find their locations on the genome. In fact, this resembles the approximate string matching problem. Errors occur in DNA due to biological, chemical and physical phenomena. Mutations occur due to imperfectness of biological complex nano machinery involved in DNA repair or replication. Additionally, active oxidative molecules may bond to DNA and alter its composition, and high energy particles and waves such as cosmic radiation and ultraviolet light of the Sun can damage the bases of DNA. Surprisingly, in absence of external factors, mutation happens spontaneously due to quantum tunneling of proteins engaged in hydrogen bonds of DNA. This alteration may become permanent if they occur during replication of DNA strands [23]. These mutations are the reasons that species and even individuals within species have different characteristics. Single nucleotide polymorphism (SNP) is an important difference between individuals' DNA which needs to be accounted for in aligning DNA fragments back to a reference genome.

2.2 Next Generation Sequencing

Next Generation Sequencing (NGS) is a DNA sequencing technology which has made DNA sequencing much cheaper and faster. NGS technology is capable of sequencing millions of DNA fragments in a parallel fashion (Fig. 2.4) which in turn has generated huge amount of genomics data [24]. As a result, several commercial NGS platforms have been introduced such as Illumina sequencing, Roche 454 sequencing, ION torrent sequencing, and SOLiD sequencing. NGS technology has been used to determine the composition of pieces of DNA in a variety of biological applications including gene expression analysis, structural variation detection (insertion, deletion or replacement in DNA molecule), protein-DNA interactions, de novo DNA sequence assembly and so forth.

As one of the most common technologies, Illumina NGS uses a library of DNA fragments as input. These libraries are DNA fragments produced in experiments that try to locate and determine the composition of protein-DNA binding sites or the structure of the coding part of genes that

eventually will be translated into proteins. At first, libraries of DNA fragments are prepared by fragmenting and denaturing a DNA sample and ligation of synthetic adapters onto both ends of DNA fragments (Fig. 2.4). These fragments will anchor in a solid surface, called a flow cell, which contains oligonucleotides (short DNA pieces) complementary to the adapter sequences attached to the fragment ends. Then, fragments hybridized to the flow cell are amplified into clusters (through bridge cluster amplification) in order to emit an adequately intense light signal for each DNA sequencing reaction cycle. The color of emitted light is used to determine types of nucleotides of fragments, this step is called base calling [10]. Depending on the NGS technology, there is a limitation of 35-700 bases on how much a DNA fragment can be sequenced [9]. To put this in perspective, recall that human DNA approximately consists of 3 billion bases. Hence, DNA is broken into smaller pieces called fragments which only the ends of them will be sequenced [25].

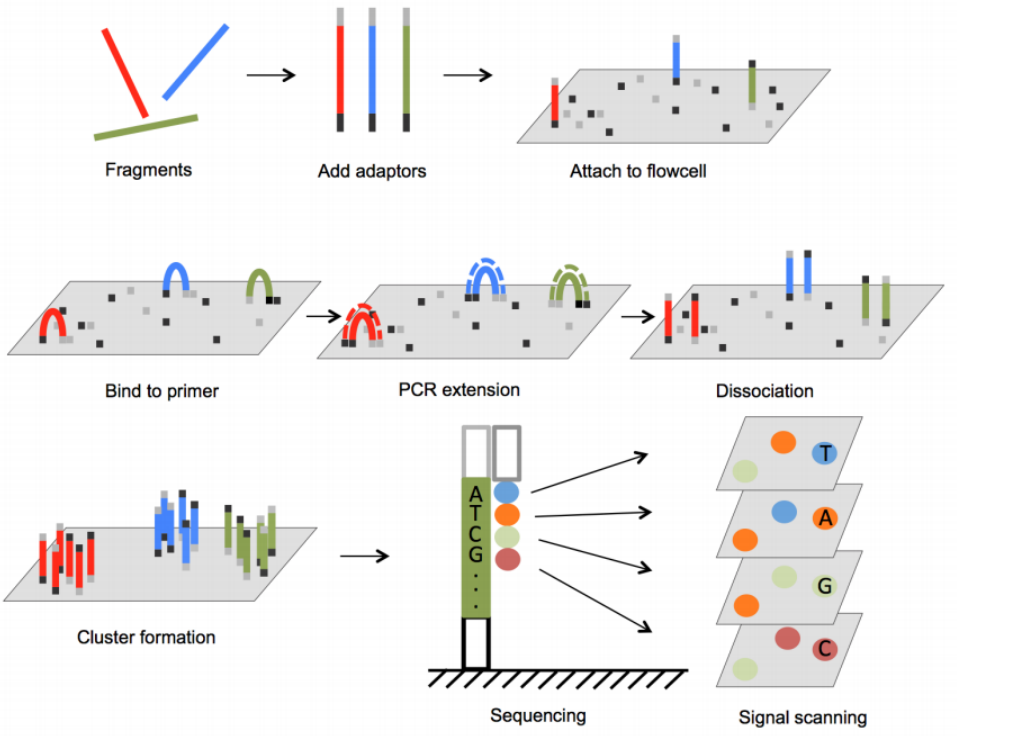


Figure 2.4: Illumina genome sequencing process [3]

DNA sequencing reaction/cycle consists of three steps; a. Nucleotide addition step; in each sequencing cycle four fluorescently marked deoxy-ribonucleotide triphosphates (dNTPs), representing four different nitrogenous bases, are added to the flow cell. dNTPs contain a nitrogenous base plus a terminator group which allows for addition of just one nucleotide at a time in a sequencing reaction. b. Signal detection step; following each addition step, an image of the flow cell is taken which will be scanned later to determine what nucleotide was attached to the template DNA fragments. In addition to the base call (detection of nucleotides) the quality of the base call is recorded. Quality , certainty of a base call, is calculated by Phred score which is $Q = -10\, log_{10}\, P$ where P represents the probability of a wrong base call [26]. Instead of sequencing a single DNA fragment, NGS massively extends this process across millions of fragments present on the flow cell in parallel. c. Wash step; after each signal detection step, the terminator group of dNTPs is removed leaving only the nitrogenous base on the fragment. Next, we perform another sequencing reaction followed by a detection step. The number of times this cycle can be repeated is restricted by the signal quality of each cycle which deteriorates and limits the length of the part of the fragment that can be sequenced [27]. A section of a fragment which has been sequenced is called a read [28].

2.3 How Reads Are Aligned to a Genome

Techniques like Illumina produce reads of length 150-250 bases with an error rate of about 1%, mostly substitutions caused by the sequencing technology. In order to find the location of genes or DNA-protein binding sites, one needs to map the reads back to the reference genome taking into account errors caused by sequencing and biological variations such as SNPs or small structural variations (deletion and insertion). This problem is modeled as approximate string matching problem for Hamming or edit distance.

There are two main algorithmic strategies to address the approximate string matching problem for large input sizes (in number of reads and size of the text): filtering and indexing. Filtering approaches quickly exclude large regions of the reference where no approximate match can be found. This can, for example, be done by identifying short regions in the reference (also known as k-mer) that share a short piece of the read without errors, often called a seed [13]. Regions that

do not share such a short region are filtered out. The simplest filtering algorithms are based on the pigeonhole lemma. The pattern (read) is divided into number of errors plus one parts. Each and every part is separately searched with zero error. Then, the locations on the genome that are found using those parts are verified by checking the vicinity of potential matches to investigate whether or not those locations can be extended to the full pattern within the range of tolerable error. The second main idea is to preprocess or index the reference sequence, the set of reads, or both, in a more intricate way. Such preprocessing into full-text string indices has the benefit that we usually do not have to scan the whole reference, but can conduct queries much faster at the expense of larger memory consumption. String indices that are currently used are suffix array [29], enhanced suffix array [30], and affix arrays [31, 32], as well as FM-index [33], a data structure based on the Burrows Wheeler Transform (BWT) [34] and some auxiliary tables. For an in-depth discussion see [35]. Such indices are usually used to pinpoint exact or approximate matches between a query and a text. For approximate string matching problem with Hamming or edit distance, the existing algorithms all have exponential complexity in the number of allowed errors K (e.g. [36, 37]), and therefore are only suited for small K.

2.4 FM Index

Burrows Wheeler Transform is a reversible cyclic permutation of the letters in a text. Indices based on Burrows Wheeler Transform, like FM index (full-text index in minute space), make it possible to inquiry substrings of a large text efficiently with a low memory usage. The Burrows Wheeler Transform of a text T, BWT(T), can be constructed as follows: The character $ is appended to end of the text T. $ does not exist in T and is lexicographically smaller than all characters in T. The Burrows Wheeler matrix of T is a matrix whose rows consists all cyclic rotations of T$ sorted in a lexicographic order[4]. BWT(T) is defined as the sequence of characters in the rightmost column of Burrows Wheeler matrix (Fig. 2.5a). BWT matrix possess an interesting property called 'last first (LF) mapping' whereby the i^{th} occurrence of character X in BWT(T) corresponds to the i^{th} occurrence of X in the first column. This property is the backbone of BWT-based indices that search for a substring in a text. BWT is reversible and Fig. 2.5b illustrates how repeatedly applying

the last first mapping recreates the original text T given its BWT(T). Similarly, LF mapping is capable of performing exact matching by applying a backward search. Since BWT matrix is sorted lexicographically, rows beginning with a given sequence appear consecutively. As pictured at Fig. 2.5c, at each step of the search the rightmost character of the pattern, which has not been searched, is selected. The search calculates the range of matrix rows successively starting with that character leading to longer suffixes of the pattern as the search continues. In other words at each step, the search picks another character and finds the ranges for that character. During the search, the size of the range shrinks. When the backward search terminates, rows beginning with the entire pattern correspond to exact occurrences of the pattern in the text and their locations can be calculated using a suffix array (SA) which is the starting positions of lexicographically sorted suffixes of the text. If the text does not contain the pattern, then the search returns a null range [4].

FM Index is an index consisted of BWT plus a few small auxiliary data structures. In fact, it is composed of first and last column of BWT matrix where the first column , F, can be represented in an extremely compact data structure and the last column , L, could also be compressed to create a very space efficient data structure. Carrying out LF mapping with scanning characters of L could take $O(|T|)$. By adding some auxiliary data structures one can much faster determine what character precedes the first character of the current range of BWT matrix. The solution is to tally the number of characters in L up to some rows, e.g. every 1024 rows such that at most by scanning 1024 characters of L we can return the number of occurrences of a character in L . More formally, Let $C[c]$ be a table containing the number of occurrences of characters lexicographically smaller than c in the text. $C[c] + 1$ is the first occurrence of c in F. Let $Occ(c, k)$ returns the number of occurrences of character c in the prefix $L[1..k]$. The LF mapping can be defined as $LF(i) = C[L[i]] + Occ(L[i], i)$ which maps element i of L into element $LF(i)$ of F. FM index is defined as the collection of C, Occ, and L [33, 38].

Figure 2.5: **(a)** Burrows Wheeler Transform, **(b)** Recovering T from BWT(T), **(c)** Backward search for pattern aac [4]

2.5 Bidirectional FM Index

Without loss of generality we will introduce bidirectional FM index only by focusing on BWT rather than its auxiliary data structures. Although BWT is very effective to search for an exact pattern, but backward search is not suitable to efficiently perform approximate string matching. Employing a combination of backward and forward search seems to be necessary and a naive solution would be to utilize two BWTs and perform the forward and backward search on T and its reverse T^{rev}. That is, one BWT is constructed for T, and another one is built for T^{rev}. Denote these BWTs as I and I^{rev} , respectively. Given any pattern P, to perform forward search, we could perform backward search on P^{rev} (the reversal of P) using I^{rev}. However, conversion of the range based on T^{rev} to the range with respect to T is not trivial [6].

To address this problem, bidirectional BWT stores I and I^{rev} while it is able to maintain the ranges with respect to T even when a forward search is performed. Consider performing forward search for one more character, i.e. finding the range for Pc_j given the range for P. We search for $c_j P^{rev}$ using I^{rev} to calculate the range $[a'_{rev}, b'_{rev}]$. The new range with respect to T is

$[a', b'] = [a + smaller, a + smaller + (b'_{rev} - a'_{rev})]$ where $[a, b]$ is the range for P and $smaller$ is the number of all suffixes Pc_i in T where c_i is lexicographically smaller than c_j. $smaller$ can be calculated by searching for $c_i P^{rev}$ and summing the ranges for all $i < j$ [5, 6]. Figure 2.6 illustrates this relationship. This novel method for updating the range allows us to start the search within a pattern and move to right or left as long as we maintain the connectivity of traversing the pattern.

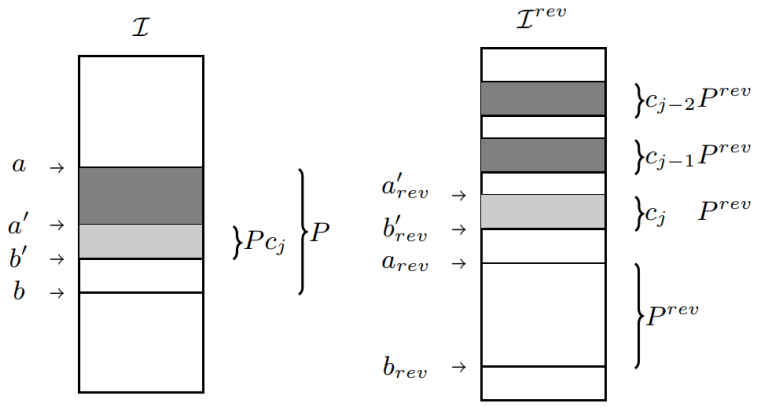


Figure 2.6: Updating the range with respect to text T for forward search [5]

As mentioned earlier FM-index is a combination of BWT and some auxiliary data structures such as an implementation of the occurrence table that holds the number of different alphabet letters in prefixes of BWT. Lam et al [6] used a bit vector to hold the occurrence table with the time and space complexity of $O(\sigma)$ and $O(\sigma \times |T|)$, respectively. In order to reduce the space consumption of occurrence table, Grossi et al. introduced the use of binary wavelet tree for string BWT [39, 40]. Schnattinger [41] utilized the binary wavelet tree to develop bidirectional wavelet index for string BWT with the motivation to search for microRNA. MicroRNA has a secondary structure that requires to search for regions of DNA that match the structure. This makes bidirectional search an obvious method of choice for microRNA (miRNA) analysis [41]. Schnattinger realized the $smaller$ value can be computed in $O(log\ \sigma)$ asymptotic time which is a straight forward task in bidirectional FM index [40]. Bidirectional wavelet index has a time complexity of $O(log\ \sigma)$ and a

space complexity of $O(|T| \times log\,\sigma)$. Furthermore, Ferragina et al. introduced $m - ary$ wavelet tree to speed up the search process. In $m - ary$ wavelet tree, nodes can have m children. Belazzougui et al. proposed a compact representation of bidirectional BWT of a string T that allowed to extend to right or left in a constant time with $O(|T| \times log\,\sigma)$ space consumption [42]. Pockrandt et al. introduced Enhanced Prefixsum Rank dictionary (EPR-dictionaries) , implemented in SeqAn C++ libraries, that performs the bidirectional search in FM-index with $O(1)$ time complexity and requires $O(|T| \times log\,\sigma)$ bits per character [40].

3. SOLVING OPTIMAL SEARCH SCHEME PROBLEM USING MIP

In this section, we provide our MIP-based methodology for finding optimal search schemes after presenting some preliminaries. As defined before, in *Optimal Search Scheme Problem* we seek the search scheme that minimizes the number of steps in ASM-B while ensuring all possible mismatch patterns are covered. We will then follow with a brief computational report on solving our MIP in order to find the optimal search schemes, including its optimal objective value as a function of its input parameters, its solution running time, and its convergence rate to optimal solution.

Our MIP is for Hamming distance, but as mentioned before, based on our computational experiments (Section 5), its optimal schemes for Hamming distance are very good (but not necessarily optimal) search schemes for the edit distance as well.

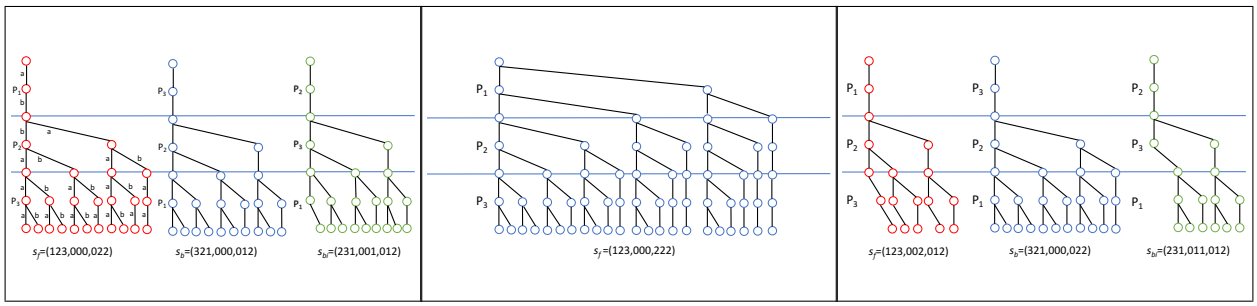
3.1 Preliminaries

Our MIP presented in Section 3.2 will solve the optimal search scheme problem assuming P is given as an input (is not a decision variable in optimization) and all P pieces of the partition are equal in length, i.e., $R = mP$, where m denotes the length of any piece. Note that these assumptions pose no practical restrictions. Given the upper bound on Hamming distance K (maximum number of mismatches) as an input, a *mismatch pattern* is a particular distribution of h mismatches among the P pieces, for any $h \leq K$. Specifically, the mismatch pattern q is a string of P integers $a_{q,1} \dots a_{q,P}$ such that $a_{q,j} \in \{0, \dots, \min\{m, K\}\}$ for $j = 1, \dots, P$, and $\sum_{j=1}^{P} a_{q,j} = h$. For given K and P, we denote the set of all possible mismatch patterns by $\mathcal{M}$. Note that if $K \leq m$ then $|\mathcal{M}| = \sum_{h=0}^{K} \binom{h+P-1}{h}$. Given a search $s = (\pi_s, L_s, U_s)$, a mismatch pattern q is said to be *covered* by s if at every iteration $i = 1, \dots, P$ of s, $L_{s,i} \leq \sum_{t=1}^{i} a_{q,\pi_{s,t}} \leq U_{s,i}$, i.e., the cumulative number of mismatches up to iteration i is between the allowed lower and upper bounds of search s. A search scheme $\mathcal{S}$ is feasible if and only if every mismatch pattern in $\mathcal{M}$ is covered by at least one search in $\mathcal{S}$.

A search scheme can be visualized by representing each of its searches as a trie that captures

all substrings enumerated by the search. Each edge at a level of the trie corresponds to a character of the alphabet at that level of search. A vertical edge represents a match, and a diagonal edge represents a mismatch. Fig. 3.1(a) shows the tries associated with the search scheme presented by [6] for $K = 2$ and $P = 3$, $\mathcal{S}_{Lam}$, applied on the six-character read "abbaaa" from alphabet $\{a, b\}$ (note that the tries are slightly different from the ones given in [7], which contained a small error). Fig. 3.1(b) shows a search scheme with a single unidirectional search (complete backtracking), $\mathcal{S}_{Uni}$, for the same problem, and Fig. 3.1(c) shows the optimal search scheme, $\mathcal{S}_{Opt}$, found by our MIP, for the same problem. Each one of the three schemes in Fig. 3.1 covers all 10 mismatch patterns, namely $\{000, 001, 010, 100, 011, 101, 110, 002, 020, 200\}$. Interestingly, the three searches s_f, s_b, s_{bi} in $\mathcal{S}_{Opt}$ cover the mismatch patterns $\{002, 011\}$, $\{000, 010, 100, 110, 020, 200\}$, and $\{001, 101\}$, respectively, which is indeed a partition of all mismatch patterns (see open problems in Section 6.1), whereas in $\mathcal{S}_{Lam}$, the searches s_f and s_b both cover 000 and 010 redundantly.

Following the method in [7], we define the performance of a search scheme as the number of forward and backward steps taken by the ASM-B algorithm, which is equal to the total number of substrings enumerated by all searches in the scheme. We assume a single step of forward or backward search in the bidirectional index takes the same amount of time. The tries of any search scheme in Fig. 3.1 contain all possible substrings of length R. The number of substrings in each trie is equal to the number of edges (or total number of non-root nodes). If the text contains all substrings of length R, the search enumerates all substrings in the tries; hence, the performance of the search scheme can be measured by the total number of edges in the search scheme. Otherwise, only a subset of the substrings in the tries will be enumerated depending on whether they occur in the text or not. To address the performance measure in this latter case, Kucherov et al. [7] assumed the read and the text are randomly and independently drawn from the alphabet according to a uniform distribution, and hence, calculated the expected number of substrings enumerated by the scheme as the sum, over all non-root nodes of the tries, of the probability that the corresponding substring appears in the text. As a result, they presented a *weighted* sum of number of edges as the measure of performance. Due to the assumption of complete randomness and independence of the



(a) (b) (c)

Figure 3.1: **(a)** The search of Lam et al. [6] as described by Kucherov et al. [7] for $K = 2$ and $P = 3$, i.e., $\mathcal{S}_{Lam} = \{s_f = (123, 000, 022), s_b = (321, 000, 012), s_{bi} = (231, 001, 012)\}$, shown for the read "abbaaa" from the alphabet {a,b}, i.e., $R = 6$ and $\sigma = 2$. The read is partitioned into P_1 =ab, P_2 =ba, and P_3 =aa. Partition borders are shown by horizontal lines. A vertical and a diagonal edge represent a match and a mismatch, respectively. Edge labels are only shown for s_f for a cleaner picture. The search corresponding to each trie is designated underneath it by its (π, L, U). The number of edges in $\mathcal{S}_{Lam}$ tries is 71. **(b)** The unidirectional search scheme $\mathcal{S}_{Uni} = \{s_f = (123, 000, 222)\}$ for the same problem. The number of edges in $\mathcal{S}_{Uni}$ is 62, i.e., for this particular problem, in which R is very small, $\mathcal{S}_{Lam}$ enumerates even more substrings than $\mathcal{S}_{Uni}$ (if all possible substrings are present in the text). Of course, if R gets larger, the situation is reversed, making $\mathcal{S}_{Lam}$ more efficient than $\mathcal{S}_{Uni}$ as reported in [6]. **(c)** The optimal search scheme $\mathcal{S}_{Opt} = \{s_f = (123, 002, 012), s_b = (321, 000, 022), s_{bi} = (231, 011, 012)\}$ for the same problem, found by our MIP. The total number of edges in $\mathcal{S}_{Opt}$ (optimal number of edges) is 59, which is less than that of $\mathcal{S}_{Uni}$, and significantly less than that of $\mathcal{S}_{Lam}$. As shown in Section 5, for bigger problems, the reduction in the total number of edges of the optimal search scheme found by our MIP compared to the unidirectional search is much more significant (up to 50%).

read and the text, they show that the weights of the edges at levels lower than $\lceil log_\sigma T \rceil + c_\sigma$ of the tries, where c_σ is that $((\sigma - 1)/\sigma)^{c_\sigma}$ is sufficiently small, are almost zero meaning that they can be dropped from the weighted summation.

For the main application of our interest, i.e. ASM of DNA sequence reads to reference genomes, the assumption of randomness and independence of the read and the text is far from reality. Calculating the expected number of substrings enumerated by a scheme calls for significant more study on determining probabilities that DNA sequence reads of particular length from a sample occur in the reference genomes. As currently there is no trivial answer to this problem, in this paper, we use the same performance measure of total number of edges in the tries of the search scheme even for the case where not all substrings occur in the text. Of course, our MIP can be easily

modified to incorporate any other weighting scenario which might be proposed in the future.

Adapting the method in [7], the total number of edges in the search scheme is calculated by

$$\sum_{s=1}^{S} \sum_{l=1}^{R} \sum_{d=0}^{K} n_{s,l,d}, \tag{3.1}$$

where $n_{s,l,d}$ is defined as the number of edges at level l of the trie of search s that end at nodes corresponding to substrings with d cumulative mismatches up to that level. The value of $n_{s,l,d}$ can be calculated using the following recursive equation, which is an adaptation of the formula in [7]:

$$n_{s,l,d} = n_{s,l-1,d} + (\sigma - 1) n_{s,l-1,d-1} \qquad \text{for } l \geq 1 \text{ and } \mathscr{L}_{s,l} \leq d \leq \mathscr{U}_{s,l}, \tag{3.2}$$

where, by definition, $n_{s,0,0} = 1$, $n_{s,0,-1} = 0$ and $n_{s,0,d} = 0$, for $d \geq 1$, $s = 1, \ldots, S$, and $\mathscr{L}_{s,l}$ and $\mathscr{U}_{s,l}$ denote the smallest and largest cumulative number of mismatches that can occur at level l of the trie of search s, respectively, calculated as $\mathscr{L}_{s,l} = \max\{L_{s,\lceil l/m \rceil - 1}, L_{s,\lceil l/m \rceil} - m\lceil l/m \rceil + l\}$ and $\mathscr{U}_{s,l} = \min\{U_{s,\lceil l/m \rceil}, \mathscr{U}_{s,l-1} + 1\}$. Here $\lceil l/m \rceil$, the smallest integer greater than or equal to l/m, would be the index of the iteration in which level l falls, and by definition, $\mathscr{L}_{s,0} = \mathscr{U}_{s,0} = 0$, for $s = 1, \ldots, S$. For example, for search s_{bi} of $\mathcal{S}_{Opt}$, we have $\mathscr{L}_{s_{bi}} = (0, 0, 0, 1, 1, 1)$ and $\mathscr{U}_{s_{bi}} = (0, 0, 1, 1, 2, 2)$.

3.2 MIP Formulation of Optimal Search Scheme Problem

Our MIP formulation, presented below, solves the optimal search scheme problem assuming P is given as an input and pieces are all equal in length. More specifically, for given K, R, P, and $\overline{S}$, this MIP finds the search scheme with minimum total number of edges among all feasible search schemes that have at most $\overline{S}$ searches. The optimal solution to the MIP provides the (π, L, U) of all searches in the optimal search scheme. The objective value of this optimal solution provides the minimum total number of edges (substrings) achievable among all feasible search schemes.

$$\min \quad \sum_{s=1}^{\overline{S}} \sum_{l=1}^{R} \sum_{d=0}^{K} n_{s,l,d} \tag{3.3}$$

subject to

$$\sum_{i=1}^{P} x_{s,i,j} = 1 \qquad \text{for all } s \text{ and } j \tag{3.4a}$$

$$\sum_{j=1}^{P} x_{s,i,j} = 1 \qquad \text{for all } s \text{ and } i \quad (3.4b)$$

$$\sum_{h=1}^{i} x_{s,h,j} - \sum_{h=1}^{i} x_{s,h,j-1} = t^{+}_{s,i,j} - t^{-}_{s,i,j} \qquad \text{for all } s,\ i = 2,\dots,P-1,\ j = 1,\dots,P+1 \quad (3.5a)$$

$$\sum_{j=1}^{P+1} (t^{+}_{s,i,j} + t^{-}_{s,i,j}) = 2 \qquad \text{for all } s,\ i = 2,\dots,P-1 \quad (3.5b)$$

$$d - (L_{s,\lceil l/m \rceil} - m\lceil l/m \rceil + l) + 1 \leq (K+m)\underline{z}_{s,l,d} \qquad \text{for all } s,\ l, \text{ and } d \quad (3.6a)$$

$$U_{s,\lceil l/m \rceil} + 1 - d \leq (K+1)\overline{z}_{s,l,d} \qquad \text{for all } s,\ l, \text{ and } d \quad (3.6b)$$

$$\tbinom{l}{d}(\sigma-1)^d(\overline{z}_{s,l,d} + \underline{z}_{s,l,d} - 2) \leq n_{s,l,d} - n_{s,l-1,d} - (\sigma-1)n_{s,l-1,d-1} \qquad \text{for all } s,\ l, \text{ and } d \quad (3.6c)$$

$$L_{s,i} \leq L_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1,\dots,P-1 \quad (3.7a)$$

$$U_{s,i} \leq U_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1,\dots,P-1 \quad (3.7b)$$

$$L_{s,i} + K(\lambda_{q,s} - 1) \leq \sum_{h=1}^{i} \sum_{j=1}^{P} a_{q,j} x_{s,h,j} \leq U_{s,i} + K(1 - \lambda_{q,s}) \qquad \text{for all } q,\ s, \text{ and } i \quad (3.8a)$$

$$\sum_{s=1}^{\overline{S}} \lambda_{q,s} \geq 1 \qquad \text{for all } q \quad (3.8b)$$

$$n_{s,l,d} \geq 0 \qquad \text{for all } s,\ l, \text{ and } d \quad (3.9a)$$

$$L_{s,i}, U_{s,i} \geq 0 \quad \text{Integer} \qquad \text{for all } s \text{ and } i \quad (3.9b)$$

$$x_{s,i,j}, \lambda_{q,s}, \overline{z}_{s,l,d}, \underline{z}_{s,l,d}, t^{+}_{s,i,j}, t^{-}_{s,i,j} \in \{0,1\} \qquad \text{for all } q,\ s,\ i,\ j,\ l, \text{ and } d \quad (3.9c)$$

The objective function (3.3) minimizes the total number of edges as calculated by (3.1) with $n_{s,l,d}$ as defined before. The binary variables $x_{s,i,j}$ capture the assignment of pieces to iterations, i.e., $x_{s,i,j} = 1$ if piece j is searched at iteration i of search s, and $x_{s,i,j} = 0$ otherwise. We define $x_{s,i,0} = x_{s,i,P+1} = 0$ to simplify presentation of constraints. At optimality, these variables determine the π_s values for the optimal search scheme. Constraints (3.4a) and (3.4b) make sure that for any search s, only one piece is assigned to an iteration and only one iteration is assigned to a piece.

Constraints (3.5a)-(3.5b) ensure the connectivity of the pieces and are in fact linearization of

the following constraint using auxiliary binary variables $t^+_{s,i,j}$ and $t^-_{s,i,j}$:

$$\sum_{j=1}^{P} \left| \sum_{h=1}^{i} x_{s,h,j} - \sum_{h=1}^{i} x_{s,h,j-1} \right| = 2 \qquad \text{for all } s \text{ and } i = 2, \ldots, P-1, \tag{3.10}$$

which is one way to enforce connectivity of pieces. The term $\sum_{h=1}^{i} x_{s,h,j}$ will have a binary value which denotes whether or not piece j has been searched at any of iterations 1 to i of search s. The term $\sum_{h=1}^{i} x_{s,h,j-1}$ captures the same notion for piece $j-1$. If at any iteration all searched pieces form a connected block on the read, the value of $\sum_{h=1}^{i} x_{s,h,j} - \sum_{h=1}^{i} x_{s,h,j-1}$ will be equal to 1 only for one j, -1 for another j, and 0 for all other j's, which is ensured by (3.10), and hence its linearization.

Constraints (3.6a)-(3.6c) enforce calculation of $n_{s,l,d}$ based on the recursive equation (3.2) with the help of binary variables $\underline{z}_{s,l,d}$ and $\overline{z}_{s,l,d}$. Due to (3.6a), if $d \geq L_{s,\lceil l/m \rceil} - m\lceil l/m \rceil + l$, then $\underline{z}_{s,l,d} = 1$, and due to (3.6b), if $d \leq U_{s,\lceil l/m \rceil}$, then $\overline{z}_{s,l,d} = 1$. Calculation of equation (3.2) is then enforced by (3.6c). When $\underline{z}_{s,l,d} = \overline{z}_{s,l,d} = 1$, (3.6c) reduces to $n_{s,l,d} - n_{s,l-1,d} - (\sigma-1)n_{s,l-1,d-1} \geq 0$, which implies $n_{s,l,d} - n_{s,l-1,d} - (\sigma-1)n_{s,l-1,d-1} = 0$ since the objective function is to be minimized. If any of $\underline{z}_{s,l,d}$ or $\overline{z}_{s,l,d}$ is equal to 0, (3.6c) does not enforce anything as $-\binom{l}{d}(\sigma-1)^d$ is a lower bound on the right-hand side of (3.6c). Constraints (3.7a)-(3.7b) ensure $L_{s,i}$ and $U_{s,i}$ are non-decreasing as they are cumulative values. Constraints (3.8a)-(3.8b) ensure feasibility of the search scheme. $\lambda_{q,s}$ is a binary variable designating whether or not mismatch pattern q is covered by search s. Constraint (3.8a) forces $\lambda_{q,s} = 0$ if search s does not cover mismatch pattern q and constraint (3.8b) ensures every mismatch pattern q is covered by at least one search, for $q = 1, \ldots, |\mathcal{M}|$.

Constraints (3.4a)-(3.9c) are enough to formulate the MIP; however, we have noticed that imposing the additional constraints

$$x_{1PP} = 1 \tag{3.11a}$$

$$\sum_{t=s}^{\overline{S}} \sum_{k=1}^{j-1} x_{t,1,k} \leq (\overline{S} - s + 1)(1 - x_{s,1,j}) \qquad \text{for all } s \text{ and } j = 2, \ldots, P \tag{3.11b}$$

$$\sum_{j=1}^{P-i+1} x_{sij} + \sum_{j=i}^{P} x_{sij} = 1 \qquad \text{for all } s \text{ and } i \geq \lceil P/2 \rceil + 1 \qquad (3.12)$$

strengthens the formulation while preserving at least one optimal solution, resulting in faster solution time for the MIP. Constraints (3.11a) and (3.11b) eliminate some symmetry in the solution space. For every search scheme, there is an equivalent search scheme obtained by reversing all π_s, $s = 1, \ldots, \overline{S}$. Constraint (3.11a) eliminates one of these two equivalent solutions in each pair by forcing piece P to be assigned to iteration P in the first search, eliminating the solutions in which piece 1 is assigned to iteration P. For any search scheme, another equivalent search scheme can be obtained by permuting the indices of searches within the scheme. Existence of only one of the search schemes obtained by this index permutation in the feasible solution set is enough. This can be achieved by sorting (in ascending order) the searches based on the piece assigned to their first iteration. This is done by constraint (3.11b), which does not allow pieces $1, \ldots, j-1$ to be assigned to the first iteration of searches $s, \ldots, \overline{S}$ if piece j is assigned to the first iteration of search s. In addition to symmetry elimination, notice that the connectivity condition of pieces implies that the piece assigned to iteration P is either piece 1 or piece P, and in general, the piece assigned to iteration $i \geq \lceil P/2 \rceil + 1$ is one of pieces $1, \ldots, P-i+1, i, \ldots, P$. Constraint (3.12) enforces this property, which strengthens the formulation, and according to our computational tests, reduces the running time of the MIP.

3.3 Solving MIP

We used CPLEX 12.7.1 solver[1] to solve our MIP by implementing the code[2] in C++ using CPLEX Callable Library. All instances were run over four 28-core nodes (2.4 GHz Intel Broadwell) with 64GB of memory per node. We ran our MIP solver for instances generated for a broad range of parameters K, R, $\overline{S}$, and P and gave each instance a 3-hour time limit. Fig.3.2 is a

[1] https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.7.1/ilog.odms.studio.help/Optimization_Studio/topics/COS_home.html

[2] Our MIP Code along with instructions to use it is available at https://github.com/kianfar77/OptimumSearchSchemes. The code accepts any arbitrary set of input parameters. The particular set of parameters used to prepare the data for Figure 3.2, 3.3, and Table 3.2 are provided at the aforementioned address as well.

small representative of our results. It shows the optimal objective value (total number of edges) for $R = 15, 25, 35, 50, 75, 100$, $K = 1, \ldots, 4$, $P = 5, 6$, and $\overline{S} = 1, \ldots, 5$. If the problem is not solved to optimality in 3 hours, the best solution found within this time limit is shown. The optimal objective value does not show a consistent change pattern in terms of change in P; however, as expected, it increases as K increases, as R increases (not shown), and as $\overline{S}$ decreases. In all instances, the optimal objective value shows a sharp drop from $\overline{S} = 1$ to $\overline{S} = 2$, then a modest drop to $\overline{S} = 3$, and negligible change beyond $\overline{S} = 3$, generating empty searches in many cases. Therefore, as long as $\overline{S} = 5$, it is advisable to use $\overline{S} = 3$ instead if we would like to reduce the MIP running time and still find an optimal or near-optimal solution for $\overline{S} = 5$. We also noticed that the optimal search scheme obtained by our MIP is not sensitive to the value of R (see open problems in Section 6.1). Therefore, when R is large, it is advisable to solve the MIP for a much smaller reasonable value of R, e.g., $R = KP$, in order to get a solution that is most probably optimal for the large R in a much shorter amount of time.

Using the MIP formulation, we were able to solve considerable size problems to optimality. For instance, we were able to solve a problem with $K = 4$, $R = 100$, $P = 3$, and $\overline{S} = 3$ to optimality in 5802 seconds. However, more complicated cases reached the time limit of 3 hours without proving solution optimality. Consequently, it is important to investigate the rate of convergence of the solutions found during execution of MIP to the optimal solution. Fig. 3.3 illustrates the ratio of the best solutions found by MIP during its execution to the final optimal objective value plotted against running time for some instances which reached optimality. We observe that in all cases, within 0.1% to 1% of the total running time, the MIP finds a solution which is finally proved to be optimal or very close to the optimal after MIP execution is complete. In other words, the MIP solver finds optimal or near optimal solutions very early on and spends the rest of its time ensuring that no better solution exists. This can be partly due to the remaining symmetry in the solution space. Nevertheless, from practical perspective, this is an attractive property because, when the input parameters are much larger, we can run the MIP for a short time and find solutions which are most probably optimal or near-optimal.

3.4 Sensitivity Analysis for Parameters of The MIP

Since ASM is exponential in terms of the number of errors, the behavior of the MIP for different values of parameters such as read length, number of partitions, number of errors, and number of searches is of high interest. In addition, the overall convergence behavior of the MIP plays an important role in answering how big a problem can be solved using the MIP. Therefore, we will be examining the effects of aforementioned parameters in this section.

We used CPLEX 12.7.1 solver [43] to solve our MIP by implementing the code in C++ using CPLEX Callable Library. All instances were run over four 28-core nodes (2.4 GHz Intel Broadwell) with 64GB of memory per node. We ran our MIP solver for instances generated for a broad range of parameters K, R, $\overline{S}$, and P and gave each instance a 3-hour time limit. Fig. 3.2 shows the optimal objective value (total number of edges) for $R = 15, 25, 35, 50, 75, 100$, $K = 1, \ldots, 4$, $P = 5, 6$, and $\overline{S} = 1, \ldots, 5$. If the problem is not solved to optimality in 3 hours, the best solution found within this time limit is shown. The optimal objective value does not show a consistent pattern in terms of change in P; however, as expected, it increases as K increases, as R increases, and as $\overline{S}$ decreases. In all instances, the optimal objective value shows a sharp drop from $\overline{S} = 1$ to $\overline{S} = 2$, then a modest drop to $\overline{S} = 3$, and negligible change beyond $\overline{S} = 3$, generating empty searches in many cases. Therefore, as long as $\overline{S} = 5$, it is advisable to use $\overline{S} = 3$ instead if we would like to reduce the MIP running time and still find an optimal or near-optimal solution for $\overline{S} = 5$. We also noticed that the optimal search scheme obtained by our MIP is not sensitive to the value of R. Therefore, when R is large, it is advisable to solve the MIP for a much smaller reasonable value of R, e.g., $R = KP$, in order to get a solution that is most probably optimal for the large R in a much shorter amount of time.

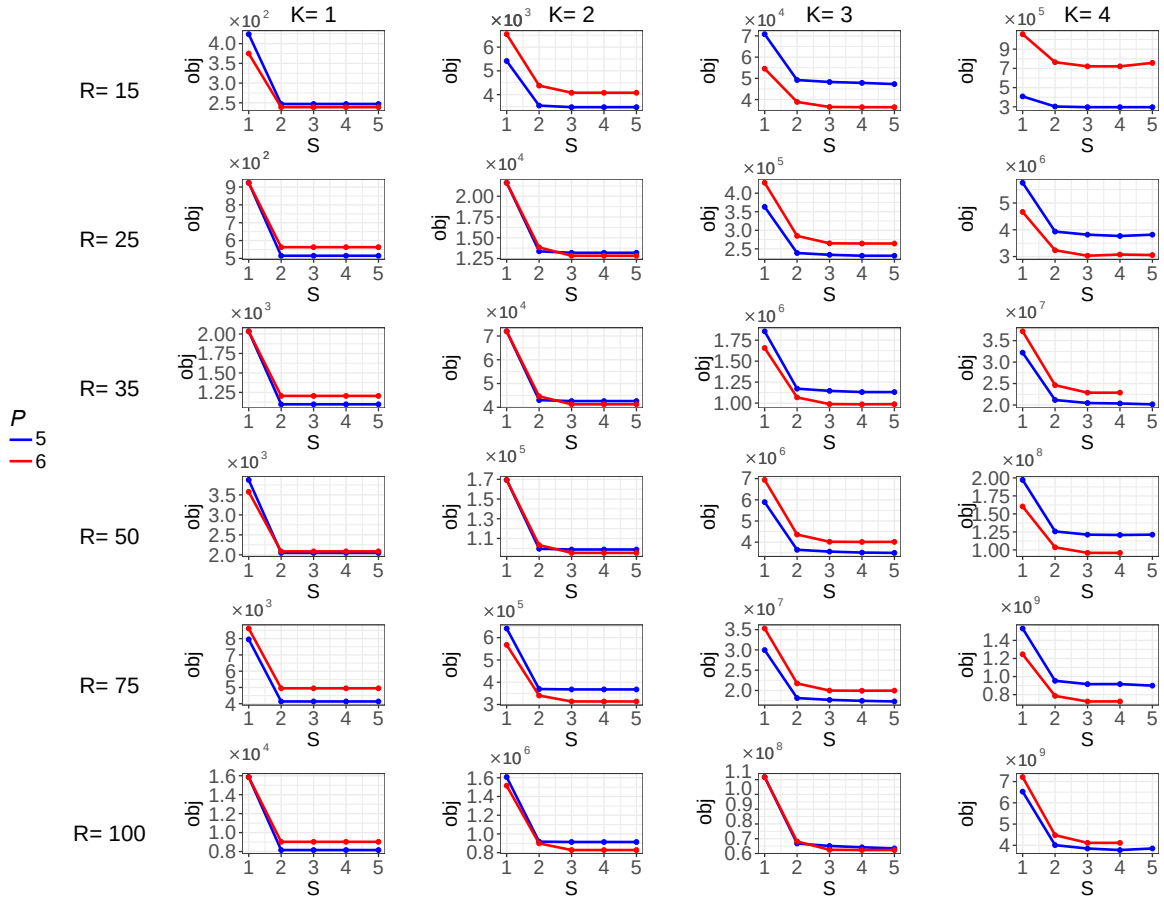


Figure 3.2: Sensitivity of optimal objective value to parameters R, K, $\overline{S}$, and P. For some cases due to memory overflow, there is no data point.

Using the MIP formulation, we were able to solve considerable size problems to optimality. For instance, we were able to solve a problem with $K = 4$, $R = 100$, $P = 3$, and $\overline{S} = 3$ to optimality in 5802 seconds. However, more complicated cases reached the time limit of 3 hours without proving solution optimality. Consequently, it is important to investigate the rate of convergence of the solutions found during execution of MIP to the optimal solution. Fig. 3.3 illustrates the ratio of the best solutions found by MIP during its execution to the final optimal objective value plotted against running time for some instances which reached optimality. We observe that in all cases, within 0.1% to 1% of the total running time, the MIP finds a solution which is finally proved to be optimal or very close to the optimal after MIP execution is complete. In other words, the MIP

solver finds optimal or near optimal solutions very early on and spends the rest of its time ensuring that no better solution exists. This can be partly due to the remaining symmetry in the solution space. Nevertheless, from practical perspective, this is an attractive property because, when the input parameters are much larger, we can run the MIP for a short time and find solutions which are most probably optimal or near-optimal.

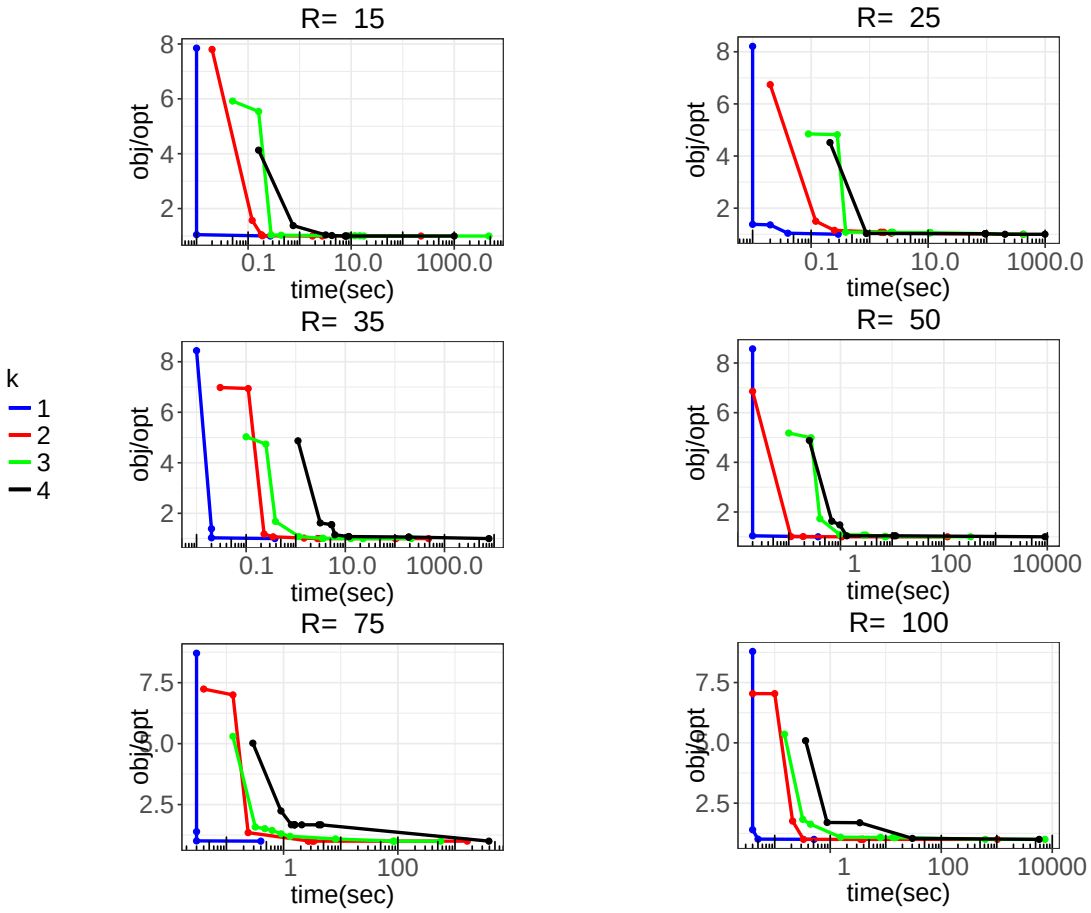


Figure 3.3: Rapid convergence of feasible solutions to the optimal solution.

3.5 Search-in-Index Computational Gains Achieved by Optimum Schemes

In this section, we present the computational advantages[3] our optimal search schemes achieved by using our optimal search schemes in ASM-B (ASM performed completely in bidirectional FM-index).

While our optimal search schemes can be found for any alphabet size and read length, we chose to concentrate on parameter values relevant to standard sequencing reads, e.g., Illumina reads. In Table 3.1, for a number of relevant parameter values, we have shown how the total number of edges using the optimal search schemes found by our MIP is reduced compared to the unidirectional backtracking scheme for reads of length $R = 101$. It can be seen that the reduction is between 41% and 49%. For $K = 1$, 2, 3, and 4, the optimal search scheme with $P = K + 3$ has the fewest number of edges. The corresponding optimal search schemes are presented in Table 3.2.

Table 3.1: Total number of edges in the optimal search schemes found by our MIP for $K = 1, 2, 3$ and $P = K + 1$, $P = K + 2$ and $P = K + 3$ compared to full backtracking. The factor column shows the ratio of total number of edges in each scheme to that in backtracking. The optimal search schemes are listed in Table 3.2.

Distance	Search Scheme	$K = 1$		$K = 2$		$K = 3$		$K = 4$	
		Edges	Factor	Edges	Factor	Edges	Factor	Edges	Factor
Hamming	Backtracking	15,554	1.00	1,560,854	1.00	116,299,379	1.00	6,862,924,649	1.00
	Optimal ($P = K + 1$)	8,004	0.51	892,769	0.57	67,888,328	0.58	4,064,852,156	0.59
	Optimal ($P = K + 2$)	8,922	0.57	854,303	0.55	65,116,676	0.56	3,916,700,994	0.57
	Optimal ($P = K + 3$)	8,004	0.51	835,213	0.54	64,060,718	0.55	3,887,857,820	0.57
Edit	Backtracking	41,208	1.00	11,154,036	1.00	2,264,515,748	1.00	367,846,294,116	1.00
	Optimal ($P = K + 1$)	20,908	0.51	6,315,779	0.57	1,299,709,022	0.57	213,296,122,595	0.58
	Optimal ($P = K + 2$)	23,356	0.57	6,025,907	0.54	1,246,126,103	0.55	205,509,484,572	0.56
	Optimal ($P = K + 3$)	20,908	0.51	5,892,667	0.53	1,226,903,544	0.54	203,270,363,390	0.55

[3]The code and input data files for the benchmarking experiments in this section along with instructions to use it is available at https://github.com/kianfar77/OptimumSearchSchemes. This code can be used to generate the results in Tables 3.1 to 5.4. It can also be used to do user-customized benchmarking.

Table 3.2: Search schemes found by our MIP for $K = 1, 2, 3$ and $P = K + 1$, $P = K + 2$ and $P = K + 3$ used for experiments in Tables 1 and 2. The schemes for $K = 1$ and 2 in all cases are optimal schemes with $\overline{S} = 5$, and the scheme for $K = 3$ and $P = K + 1$ is the optimal scheme with $\overline{S} = 3$. To control the running time of MIP, the schemes for $K = 3$ and 4 are best solutions found by running the MIP for 2 hours with $\overline{S} = 3$. These schemes are most probably optimal for $\overline{S} = 3$.

	$K = 1$	$K = 2$	$K = 3$	$K = 4$
Optimal ($P = K + 1$)	$(12, 00, 01)$ $(21, 01, 01)$	$(123, 002, 012)$ $(321, 000, 022)$ $(231, 011, 012)$	$(1234, 0003, 0233)$ $(2341, 0000, 1223)$ $(3421, 0022, 0033)$	$(12345, 00004, 03344)$ $(23451, 00000, 22334)$ $(54321, 00033, 00444)$
Optimal ($P = K + 2$)	$(123, 001, 001)$ $(321, 000, 011)$	$(2134, 0011, 0022)$ $(3214, 0000, 0112)$ $(4321, 0002, 0122)$	$(12345, 00022, 00333)$ $(43215, 00000, 11223)$ $(54321, 00003, 02233)$	$(123456, 000004, 033344)$ $(234561, 000000, 222334)$ $(654321, 000033, 004444)$
Optimal ($P = K + 3$)	$(1234, 0000, 0011)$ $(4321, 0001, 0011)$	$(21345, 00011, 00222)$ $(43215, 00000, 00112)$ $(54321, 00002, 01122)$	$(123456, 000003, 022233)$ $(234561, 000000, 111223)$ $(654321, 000022, 003333)$	$(1234567, 0111111, 3333334)$ $(1234567, 0000000, 0044444)$ $(7654321, 0000004, 0333344)$

Although the reduction factors in total number of edges obtained by our optimal search schemes in Table 3.1 are very significant in themselves, due to the stochastic nature of occurrence of errors in sequencing reads and occurrence of approximate matches in the reference genome, the real-case ASM speed-up factors achieved by these optimal search schemes compared to backtracking can be yet much more significant. To gain insight into this speed-up, we performed an experiment searching for *all* approximate matches (for $K = 1$, 2, and 3) of $100,000$ real Illumina reads of length $R = 101$ [4] in the human genome hg38 and compared the running time of ASM-B performed with optimal search schemes obtained by our MIP for $P = K + 1$ and $P = K + 2$ to that of unidirectional backtracking for Hamming and edit distance. Throughout this article, we use OSS to refer to an implementation of in-index bidirectional search using optimal search schemes found by our MIP. All tests were conducted on one Linux-based computing cluster node with an Intel 14-core 2.4GHz Broadwell processor and 112GB of memory. All data was stored on tmpfs, a virtual

[4] Taken from SRA accession number ERX1959065 and available at https://github.com/kianfar77/OptimumSearchSchemes

file system in main memory to prevent loading data just on demand during the search and thus affecting the speed of the search by I/O operations. All tools were run with a single thread to make the results comparable. The results are shown in Table 3.3. We can see that for both Hamming and edit distance, employing our optimal search schemes, is much faster than backtracking, verifying our expectation. The respective largest speed-up factors for $K = 1$, 2, and 3 are 4.53, 14.67, and 47.17 for Hamming distance, and 4.04, 10.26, and 19.38 for edit distance, respectively, much more significant than reduction in the total number of edges reported in Table 3.1.

Table 3.3: Running time comparison of searching all approximate matches of $100,000$ Illumina reads ($R = 101$) using optimal bidirectional scheme with $P = K + 1$ and $P = K + 2$ versus backtracking for Hamming and edit distance. The factor column is the speed-up ratio versus backtracking in each category.

Dist.	Search Tool	$K = 1$		$K = 2$		$K = 3$	
		Time	Factor	Time	Factor	Time	Factor
Hamm.	Backtracking	25.61s	1.00	215.55s	1.00	1931.16s	1.00
	Optimal-scheme bidirect. ($P = K + 1$)	10.89s	2.35	17.30s	12.46	61.78s	31.26
	Optimal-scheme bidirect. ($P = K + 2$)	5.65s	4.53	14.69s	14.67	40.94s	47.17
Edit	Backtracking	34.60s	1.00	975.71s	1.00	21321.08s	1.00
	Optimal-scheme bidirect. ($P = K + 1$)	8.62s	4.01	101.19s	9.64	1158.31s	18.41
	Optimal-scheme bidirect. ($P = K + 2$)	8.56s	4.04	95.13s	10.26	1100.37s	19.38

4. RELAXING ASSUMPTIONS OF EQUAL SIZE PARTITION MIP

The formulation in section 3.2 solves OSS problem with two restrictions: First, it assumes the partitions are of the same size. Second, it solves the problem for Hamming distance. In this section, we provide an alternative formulation to equal size partition MIP of section 3.2, called Fixed General Partitioning. We will relax the assumption of having equal size partitions but still assume that the partition sizes are given. In addition, this formulation has significantly less variables. In the next step we relax the assumption of having the sizes of partitions and introduce Variable Partition MIP. We let this MIP to produce the sizes of the partitions. Finally, we introduce Fixed General Edit Distance MIP which assumes that the sizes of partitions are arbitrary and given. This MIP solves the OSS problem for edit distance.

Solving Variable Partition MIP for real world read sizes needs huge amount of computational resources. Also, the optimal solutions from edit distance MIP is very close to those of the MIP from section 3.2. Therefore, we will use the results of equal size partition MIP for computational purposes throughout chapter 5.

4.1 MIP Formulation for a Fixed General Partitioning

The formulation for pre-determined general partitioning is different than the MIP presented in section 3.2 in two ways. First, it relaxes the assumption of having equal size partitions which could improve the runtime for mapping short reads [7]. Second, it uses a recursive approach for iterations rather than the levels of the trie which leads to a formulation with significantly fewer number of variables.

The MIP formulation for a fixed general partitioning, solves the optimal search scheme problem assuming P is given as an input and pieces vary in size. More specifically, for given $K, R, P, \overline{S}$, and sizes of all partitions Φ, this MIP finds the search scheme with minimum number of edges among all feasible search schemes that have at most $\overline{S}$ searches. The optimal solution to the MIP provides the (π, L, U) of all searches in the optimal search scheme.

The idea is to build a *reference trie* with a considerable number of mismatches and only one large partition. We utilize the reference trie to compile a table for counting the number of edges in search tries with the help of the recursive approch for iterations. The reference trie will be constructed by using Kucherov's recursive function assuming there is one partition. We count its number of vertices only once and compile a table containing the number of nodes with d mismatches at depth l of the reference trie, N_{ld}.

Although the MIP uses a recursive approach for iterations rather than the levels of search tries, we need to count all of the edges of a search trie and therefore we need to know the number of nodes not only at borderlines of iterations but also for all levels of a search trie. Therefore, we tabulate the cumulative number of vertices from the root of the reference trie to any level l with mismatches between a lower bound $\mathscr{L}$ and an upper bound $\mathscr{U}$ plus their ancestor nodes which are necessary to maintain a tree structure. We denote the cumulative number of nodes with $V_{l\mathscr{L}\mathscr{U}}$ and define it as

$$V_{l\mathscr{L}\mathscr{U}} = (\sum_{d=\mathscr{L}}^{\mathscr{U}} \sum_{h=0}^{l} N_{hd} + \sum_{d=0}^{\mathscr{L}-1} \sum_{h=0}^{l-(\mathscr{L}-d)} N_{hd}) - 1 \tag{4.1}$$

where the first term counts the number of nodes with an error between $\mathscr{L}$ and $\mathscr{U}$ mismatches, the second term counts their ancestor nodes, and subtracting 1 ensures that we do not count nodes at the borders twice when we calculate the total number of edges in a search trie. We conform to the convention whereby $V_{l\mathscr{L}\mathscr{U}} = 0$ for all $\mathscr{L} < 0$.

Figure 4.1 shows a reference trie allowing $K = 5$ mismatches with one partition of size 100 which would be large enough for most applications. As mentioned earlier, we need to calculate $V_{l\mathscr{L}\mathscr{U}}$ for different values of $l, \mathscr{L}$, and $\mathscr{U}$. For instance, we calculate $V_{5,2,3}$ by adding blue nodes (2 mismatches), red nodes (3 mismatches), and their ancestor nodes needed to preserve a tree structure (green nodes).

Having the table, we can introduce a new recursive function for iterations (borders of partitions). If we know the number of nodes with certain cumulative mismatches at a borderline and the length of the following iteration, we can use our table to generate the number of nodes with various number

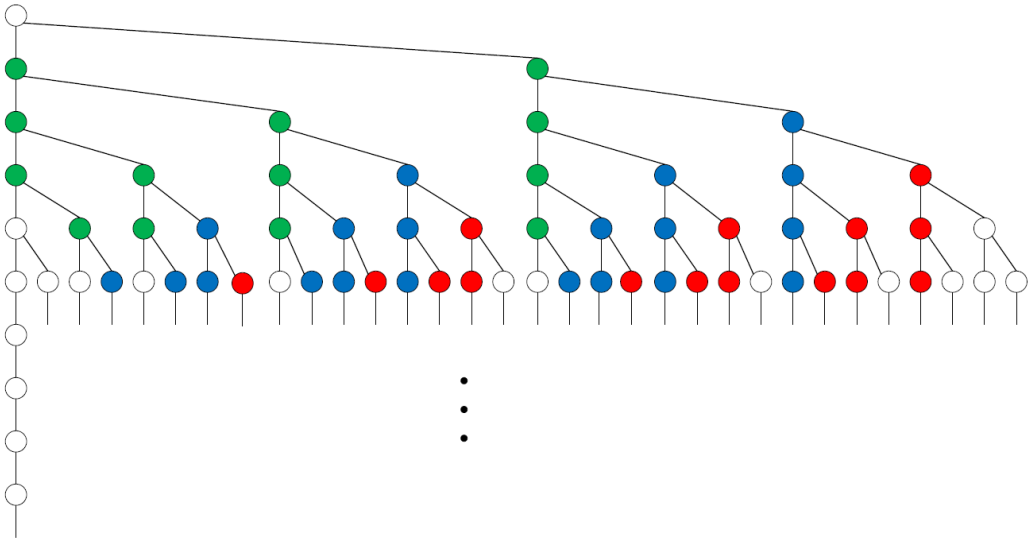


Figure 4.1: A reference trie for $K = 5$ with one partition of size 100. Nodes with 2 and 3 errors are presented in blue and red, respectively. Their ancestor nodes are depicted in green. These nodes represent $V_{5,2,3}$. $V_{l\mathscr{L}\mathscr{U}}$ will be used to count the number of edges in search tries. The root node is excluded to prevent counting the nodes at border of partitions twice. The rest of the trie has not been presented.

of mismatches at the next borderline. Let n_{sid} denote the number of nodes with d cumulative mismatches at iteration i, borderline of i and $i+1$ iterations, of search s. We establish the following recursive formula for the number of nodes at the end of iterations using N_{ld}, from the reference trie, with l_i denoting the length of partition i. We also define $n_{s,0,0} = 1$ and $n_{s,0,d} = 0$ for $d \neq 0$.

$$n_{sid} = \sum_{h=0}^{d} n_{s,i-1,h} \; N_{l_i,d-h} \tag{4.2}$$

Furthermore, we count the total number of edges by adding the number of nodes for all iterations.

$$Total\ edges \; = \sum_{i=1}^{P} \sum_{d=0}^{U_{s,i-1}} n_{s,i-1,d} \; V_{l_i,L_{si}-d,U_{si}-d} \tag{4.3}$$

where l_i represents the length of iteration i, L_{si} denotes minimum cumulative number of errors at iteration i, and U_{si} represents maximum cumulative number of errors at iteration i. We define U_{s0}

as 0.

Figure 4.2 demonstrates how to use $V_{5,2,3}$ to count the number of nodes in the second iteration of a search trie for a read of length 7 with 3 partitions of sizes $1, 5$, and 1, respectively. We have assumed the alphabet is consisted of only two characters. The lower bound and upper bound for errors are $L_{s2} = 3$, and $U_{s2} = 4$. Since the node at the borderline of iteration one and two has one cumulative error, $d = 1$, its decedents acquire another extra 2 or 3 errors. Therefore, the structure of nodes in the second iteration of the search would be identical to $V_{5,2,3}$ of the reference trie.

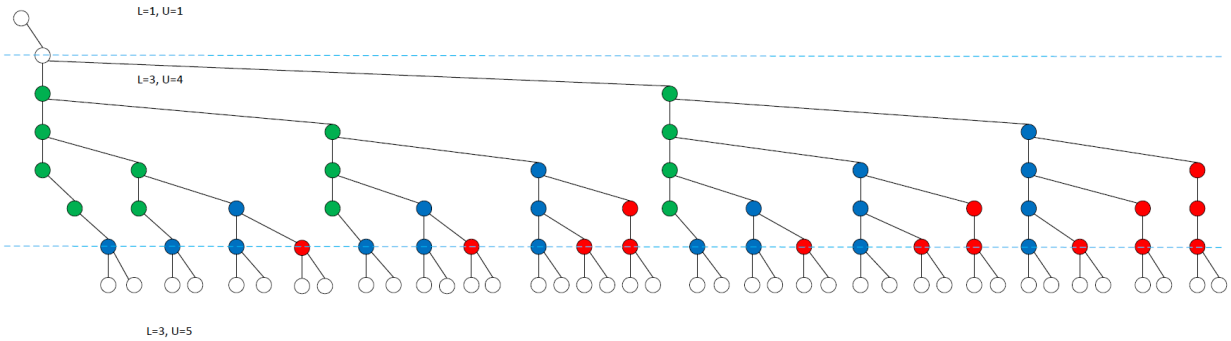


Figure 4.2: A search trie with three partitions of size $1, 5$, and 1, respectively. $L_{s1} = 1, U_{s1} = 1, L_{s2} = 3, U_{s2} = 4, L_{s3} = 3, U_{s3} = 5$. The structure of the trie at the second iteration is exactly $V_{5,2,3}$ for the decedents of a node with $d = 1$ located at the border of the first and second iterations.

Our MIP formulation for a fixed general partitioning, presented below, solves the optimal search scheme problem assuming P is given as an input and pieces vary in size. More specifically, for given $K, R, P, \overline{S}$, and sizes of all partitions Φ, this MIP finds the search scheme with minimum number of edges among all feasible search schemes that have at most $\overline{S}$ searches. The optimal solution to the MIP provides the (π, L, U) of all searches in the optimal search scheme. The objective value of this optimal solution provides the minimum number of edges achievable among all feasible search schemes.

$$\min \sum_{s=1}^{S} \sum_{i=1}^{P} \eta_{si} \tag{4.4}$$

subject to

$$\sum_{i=1}^{P} x_{sij} = 1 \qquad \text{for all } s \text{ and } j \tag{4.5a}$$

$$\sum_{j=1}^{P} x_{sij} = 1 \qquad \text{for all } s \text{ and } i \tag{4.5b}$$

$$\sum_{h=1}^{i} x_{shj} - \sum_{h=1}^{i} x_{sh,j-1} = t^{+}_{sij} - t^{-}_{sij} \qquad \text{for all } s,\ i = 2, \ldots, P-1,\ j = 1, \ldots, P+1 \tag{4.5c}$$

$$\sum_{j=1}^{P+1} (t^{+}_{sij} + t^{-}_{sij}) = 2 \qquad \text{for all } s,\ i = 2, \ldots, P-1 \tag{4.5d}$$

$$y_{sil}\ l \leq \sum_{j=1}^{P} m_j\ x_{sij} \qquad \text{for all } s, i \text{ and } l \in \Phi \tag{4.6a}$$

$$R\,(1-y_{sil})+l \geq \sum\nolimits_{j=1}^{P} m_j\; x_{sij} \qquad \text{for all } s, i \text{ and } l \in \Phi \tag{4.6b}$$

$$\sum\nolimits_{l\in\Phi} y_{sil} = 1 \qquad \text{for all } s \text{ and } i \tag{4.6c}$$

$$d - L_{si} + 1 \leq (K+1)\underline{z}_{sid} \qquad \text{for all } s,\ i, \text{ and } d \tag{4.7a}$$

$$-\,d + U_{si} + 1 \leq (K+1)\overline{z}_{sid} \qquad \text{for all } s,\ i, \text{ and } d \tag{4.7b}$$

$$\binom{R}{d}(\sigma-1)^d(3-y_{sil}-\overline{z}_{sid}-\underline{z}_{sid}) + n_{sid} \geq \sum_{h=0}^{d} n_{s,i-1,h}\; N_{l,d-h} \qquad \text{for all } s,\ i,\ d, \text{ and } l \in \Phi \tag{4.7c}$$

$$R\sum_{d=0}^{K}\binom{R}{d}(\sigma-1)^d(3-y_{sil}-\underline{z}_{si\mathscr{L}}-\overline{z}_{si\mathscr{U}}+\eta_{si} \geq \sum_{d=0}^{K} n_{s,i-1,d}\; V_{l,\mathscr{L}-d,\mathscr{U}-d} \qquad \text{for all } s, i,\ l \in \Phi\ \ \mathscr{L},\mathscr{U} \in \{0,...,K\} \tag{4.8a}$$

$$L_{si} \leq L_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1,\ldots,P-1 \tag{4.9a}$$

$$U_{si} \leq U_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1,\ldots,P-1 \tag{4.9b}$$

$$x_{1PP} = 1 \tag{4.10a}$$

$$(S-s+1)\,x_{s,1,j}+\sum_{t=s}^{S}\sum_{k=1}^{j-1}x_{t,1,k}\leq S-s+1 \qquad \text{for all } s \text{ and } j=2,\ldots,P \tag{4.10b}$$

$$A_{qsi}=\sum_{h=1}^{i}\sum_{j=1}^{P}a_{qj}x_{shj} \qquad \text{for all } q,\ s, \text{ and } i \tag{4.11a}$$

$$L_{si}+K(\lambda_{qs}-1)\leq A_{qsi}\leq U_{si}+K(1-\lambda_{qs}) \qquad \text{for all } q,\ s, \text{ and } i \tag{4.11b}$$

$$\sum_{s=1}^{S}\lambda_{qs}\geq 1 \qquad \text{for all } q \tag{4.11c}$$

$$\sum_{k=1}^{P-i+1}x_{sik}+\sum_{k=i}^{P}x_{sik}=1 \qquad \text{for all } s \text{ and } i\geq\lfloor P/2\rfloor+2 \tag{4.12a}$$

$$n_{sid},A_{qsi}\geq 0 \qquad \text{for all } q,\ s,\ i, \text{ and } d \tag{4.13a}$$

$$L_{si},U_{si}\geq 0 \text{ and integer} \qquad \text{for all } s \text{ and } i \tag{4.13b}$$

$$x_{sij},\lambda_{qs},y_{sil},\overline{z}_{sid},\underline{z}_{sid},t^{+}_{sij},t^{-}_{sij}\in\{0,1\} \qquad \text{for all } q,\ s,\ i,\ j, \text{ and } d \tag{4.13c}$$

The objective function (4.4) minimizes the total number of edges as calculated in (4.3) with the help of (4.7c) and (4.8a). η_{si} is defined as $\sum_{d=0}^{K} n_{s,i-1,d}\ V_{l,\mathscr{L}-d,\mathscr{U}-d}$ where $V_{l,\mathscr{L}-d,\mathscr{U}-d}$ is defined by (4.1) .

The binary variable $x_{s,i,j}$ captures the assignment of pieces to iterations. Constraints (4.5a) and (4.5b) make sure that for any search s, only one piece is assigned to an iteration and vice versa.

Constraints (4.5c) and (4.5d) ensure the connectivity of the pieces and are in fact linearization

of (3.10) which is one way to enforce connectivity of pieces.

Constraints (4.6a), (4.6b), and (4.6c) assign a proper size to iteration i. m_j is the size of partition j and y_{sil} is a binary variable. When $y_{sil} = 1$, constraints (4.6a) and (4.6b) reduce to $\sum_{j=1}^{P} m_j \; x_{sij} = l$ which is true only if l , the size of iteration i, equals the size of partition j for whom $x_{s,i,j} = 1$. Constraint (4.6c) makes sure that only one size is assigned to an iteration.

Constraints (4.7a) - (4.7c) enforce the recursive equation (4.2) to calculate the number of nodes at the iteration borderlines of a search trie. With the help of binary variables $\overline{z}_{sld}$ and $\underline{z}_{sld}$ in constraints (4.7a) and (4.7b), if $L_{si} \leq d \leq U_{si}$ and $y_{sil} = 1$ the recursive formula $n_{sid} = \sum_{h=0}^{d} n_{s,i-1,h} \; N_{l,d-h}$ is enforced via constraint (4.7c). When $y_{sil} = \underline{z}_{s,l,d} = \overline{z}_{s,l,d} = 1$, (4.7c) reduces to $n_{sid} \geq \sum_{h=0}^{d} n_{s,i-1,h} \; N_{l,d-h}$, which implies $n_{sid} = \sum_{h=0}^{d} n_{s,i-1,h} \; N_{l,d-h}$ because the objective function is to be minimized. Otherwise, (4.7c) becomes a trivial inequality.

Constraints (4.7a) - (4.7c) and (4.8a) count the number of nodes within iteration i of search s. With the help of constraints (4.7a) and (4.7b), $\underline{z}_{si\mathscr{L}}$ becomes one for $\mathscr{L} = L_{si}$. Similarly, $\overline{z}_{si\mathscr{U}}$ is one if $\mathscr{U}$ equals the upper bound U_{si}. These two force (4.7c) to generate the nodes. Although for $\mathscr{L} > L_{si}$ and $\mathscr{U} < U_{si}$, $\underline{z}_{si\mathscr{L}} = \overline{z}_{si\mathscr{U}} = 1$, but this only enforces η_{si} to be greater than some values smaller than $\sum_{d=0}^{K} n_{sid} \; V_{l,L_{si}-d,U_{si}-d}$ which will not impose any problem since $\eta_{si} \geq \sum_{d=0}^{K} n_{sid} \; V_{l,L_{si}-d,U_{si}-d}$.

When $y_{sil} = \underline{z}_{si\mathscr{L}} = \overline{z}_{si\mathscr{U}} = 1$, (4.8a) reduces to $\eta_{si} \geq \sum_{d=0}^{K} n_{sid} \; V_{l,\mathscr{L}-d,\mathscr{U}-d}$, which in turn reduces to $\eta_{si} = \sum_{d=0}^{K} n_{sid} \; V_{l,\mathscr{L}-d,\mathscr{U}-d}$ because the MIP is a minimization problem. Otherwise, (4.8a) becomes a trivial inequality.

As in prior MIP formulation, constraints (4.9a)-(4.9b) enforce the non-decreasing property of lower bounds and upper bounds L_{si} and U_{si}. Constraints (4.10a)-(4.10b) eliminate some symmetry in the feasible region of the problem.

Constraints (4.11a)-(4.11c) make sure that every mismatch pattern over P partitions of the read is covered by at least one search. Constraint (4.11a) calculates A_{qsi} which is the cumulative number of mismatches up to and including iteration i of search s. If $\lambda_{qs} = 1$ in constraint (4.11b), A_{qsi} is forced to be between lower bound L_{si} and upper bound U_{si} at every iteration i which means the

mismatch pattern q is covered by s. Constraint (4.11c) ensures each pattern is covered at least by one search. Constraints (4.12a) is not necessary however it speeds up the MIP.

4.2 Variable Partition

All the formulations described in prior sections answer the optimal search scheme problem assuming the size of the partitions are given. We will present a formulation based on the MIP discussed in section 3.2 which is capable of answering the optimal search scheme problem without knowing the size of partitions. In fact, by setting an upper bound on the number of partitions, The MIP is not only able to find the optimal size of partitions but also determines the optimal number of partitions. The number of partitions with a none zero size, equals the optimal number of partitions.

Our MIP formulation for variable partitioning, presented below, solves the optimal search scheme problem assuming $\overline{P}$ is an upper bound on the number of partitions. For given $K, R, \overline{P}$, and $\overline{S}$ this MIP finds the search scheme with minimum number of edges that have at most $\overline{S}$ searches. The optimal solution to the MIP provides the (π, L, U), the number of non empty searches, and the number, and sizes of the partitions in the optimal search scheme.

$$\min \sum_{s=1}^{S} \sum_{l=1}^{B} \sum_{d=0}^{K} n_{sld} \tag{4.14}$$

subject to

$$\sum_{i=1}^{\overline{P}} x_{sij} = 1 \qquad \text{for all } s \text{ and } j \tag{4.15a}$$

$$\sum_{j=1}^{\overline{P}} x_{sij} = 1 \qquad \text{for all } s \text{ and } i \tag{4.15b}$$

$$\sum_{h=1}^{i} x_{shj} - \sum_{h=1}^{i} x_{sh,j-1} = t^{+}_{sij} - t^{-}_{sij} \qquad \text{for all } s,\ i = 2, \ldots, \overline{P} - 1,\ j = 1, \ldots, \overline{P} + 1 \tag{4.15c}$$

$$\sum_{j=1}^{\overline{P}+1} (t^{+}_{sij} + t^{-}_{sij}) = 2 \qquad \text{for all } s,\ i = 2, \ldots, \overline{P} - 1 \tag{4.15d}$$

$$\sum_{h=1}^{i} \sum_{j=1}^{\overline{P}} m'_{shj} = \mu_{si} \qquad \text{for all } s \text{ and } i \tag{4.16a}$$

$$\sum_{j=1}^{\overline{P}} m_j = R \tag{4.16b}$$

$$m'_{sij} \leq R\, x_{sij} \qquad \text{for all } s,\ i, \text{ and } j \tag{4.16c}$$

$$\sum_{i=1}^{\overline{P}} m'_{sij} = m_j \qquad \text{for all } s \text{ and } j \tag{4.16d}$$

$$l - \mu_{s,i-1} \leq l y_{sil} \qquad \text{for all } s,\ i, \text{ and } l \tag{4.17a}$$

$$\sum_{l=1}^{R} y_{sil} = R - \mu_{s,i-1} \qquad \text{for all } s \text{ and } i$$

(4.17b)

$$\mathscr{L}_{sl} - L_{si} + (\mu_{si} - l) \leq (K+R)(1 - y_{sil} + y_{s,i-1,l}) \qquad \text{for all } s,\ i, \text{ and } l$$

(4.17c)

$$K(-1 + y_{sil} - y_{s,i-1,l}) \leq \mathscr{U}_{sl} - U_{si} \qquad \text{for all } s,\ i, \text{ and } l$$

(4.17d)

$$d - \mathscr{L}_{sl} + 1 \leq (R+1)\underline{z}_{sld} \qquad \text{for all } s,\ l, \text{ and } d$$

(4.18a)

$$\mathscr{U}_{sl} + 1 - d \leq (K+1)\overline{z}_{sld} \qquad \text{for all } s,\ l, \text{ and } d$$

(4.18b)

$$\tbinom{l}{d}(\sigma-1)^d(\overline{z}_{sld} + \underline{z}_{sld} - 2) \leq n_{sld} - n_{s,l-1,d} - (\sigma-1)n_{s,l-1,d-1} \qquad \text{for all } s,\ l, \text{ and } d$$

(4.18c)

$$L_{si} \leq L_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1, \ldots, \overline{P} - 1$$

(4.19a)

$$U_{si} \leq U_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1, \ldots, \overline{P} - 1$$

(4.19b)

$$x_{1\overline{P}P} = 1 \qquad (4.20a)$$

$$(S - s + 1)\, x_{s,1,j} + \sum\nolimits_{t=s}^{S} \sum\nolimits_{k=1}^{j-1} x_{t,1,k} \leq S - s + 1 \qquad \text{for all } s \text{ and } j = 2, \ldots, \overline{P}$$

(4.20b)

$$(R+1)\overline{w}_{qj} \geq a_{qj} - m_j - 0.1 \qquad \text{for all } q \text{ and } j \tag{4.21a}$$

$$(R+1)\underline{w}_{qj} \geq -(a_{qj} - m_j - 0.1) \qquad \text{for all } q \text{ and } j \tag{4.21b}$$

$$\overline{w}_{qj} + \underline{w}_{qj} = 1 \qquad \text{for all } q \text{ and } j \tag{4.21c}$$

$$A_{qsi} = \sum\nolimits_{h=1}^{i} \sum\nolimits_{j=1}^{\overline{P}} a_{qj} x_{shj} \qquad \text{for all } q, s, \text{ and } i \tag{4.21d}$$

$$L_{si} + K(\lambda_{qs} - 1) \leq A_{qsi} \leq U_{si} + K(1 - \lambda_{qs}) \qquad \text{for all } q, s, \text{ and } i \tag{4.21e}$$

$$\sum\nolimits_{s=1}^{S} \lambda_{qs} + \sum\nolimits_{j=1}^{\overline{P}} \overline{w}_{qj} \geq 1 \qquad \text{for all } q \tag{4.21f}$$

$$\sum\nolimits_{k=1}^{\overline{P}-i+1} x_{sik} + \sum\nolimits_{k=i}^{\overline{P}} x_{sik} = 1 \qquad \text{for all } s \text{ and } i \geq \lfloor \overline{P}/2 \rfloor + 2 \tag{4.22a}$$

$$n_{sld}, \mathscr{U}_{sl}, \mu_{si}, A_{qsi}, m'_{sij} \geq 0, \quad \mathscr{L}_{sl} \geq -R \qquad \text{for all } q, s, i, j, l, \text{ and } d \tag{4.23a}$$

$$L_{si}, U_{si}, m_j \geq 0 \quad \text{Integer} \qquad \text{for all } s, i, \text{ and } j \tag{4.23b}$$

$$x_{sij}, \lambda_{qs}, \overline{y}_{sil}, \underline{y}_{sil}, \overline{z}_{sld}, \underline{z}_{sld}, t^{+}_{sij}, t^{-}_{sij}, \overline{w}_{qj}, \underline{w}_{qj} \in \{0, 1\} \qquad \text{for all } q, s, i, j, l, \text{ and } d \tag{4.23c}$$

Similar to the MIP from section (3.2), we are minimizing the number of edges to find the optimal search scheme. Here, we have no restriction on the size of partitions and we only set an upper bound for the number of partitions, $\overline{P}$. The optimal solution would provide us with searches, the size of partitions, and the optimal number of partitions. The number of partitions, in optimal solution, equals the number of partitions with non-zero sizes.

Constraints (4.15a)-(4.15d) set the order of partitions in search s and ensure their connectivity. Constraints (4.15a) and (4.15b) make sure for search s, there is a one to one assignment between partitions and iterations. These constraints together determine the order of partitions in the search. Constraints (4.15c) and (4.15d) ensure the connectivity of the partitions for search s throughout all iterations, for a detailed discussion refer to section 3.2.

Constraint (4.16a) calculates the cumulative length of the partitions searched at iteration i. Let μ_{si} represent the cumulative length at iteration i for search s and define μ_{s0} as 0. m_j denotes the length of partition j and m'_{sij} is an integer that equals m_j if partition j is assigned to iteration i, otherwise m'_{sij} is zero. Equation (4.16b) ensures that the sum of partition sizes add up to the length of the read. Inequality (4.16c) forces m'_{sij} to be zero except for one pair of i and j. Since only one m'_{sij} is nonzero, (4.16d) assigns m_j to that variable.

Since the size of partitions are unknown, we cannot establish a straightforward connection between level l of a trie and its corresponding iteration. Therefore, there is no direct relation between the lower bound and upper bound for the number of errors in the iterations, and the nodes located at a level l of the search trie. Constraints (4.17a)-(4.17b) detect the iteration that contains level l by setting the values of y_{sil} such that $y_{sil} - y_{s,i-1,l}$ becomes 1 if and only if iteration i contains level l.

Constraints (4.17c) and (4.17d) set the lower bound and upper bound for the number of errors for levels of the search tries. Let $\mathscr{L}_{sl}$ represent the lower bound for cumulative mismatches of nodes, substrings, at level l of the search s. With the help of binary variable y_{sil} in constraints (4.17a)-(4.17b), when $\mu_{s,i-1} < l \leq \mu_{si}$, constraint (4.17c) sets the lower bound $\mathscr{L}_{sl}$ to $L_{si} - (\mu_{si} - l)$ which reaches to L_{si} at the end of iteration i. Let $\mathscr{U}_{sl}$ denote the upper bound for cumulative mismatches for nodes at level l of the search s. Constraint (4.17d) sets $\mathscr{U}_{sl}$ to U_{si} when iteration i contains level

l.

Constraints (4.18a)-(4.18c) enforce the recursive equation in Kucherov et al [7] for calculating the number of nodes in a search trie. With the help of binary variables $\overline{z}_{sld}$ and $\underline{z}_{sld}$ in constraints (4.18a) and (4.18b), when $\mathscr{L}_{sl} \leq d \leq \mathscr{U}_{sl}$ the recursive formula

$$n_{sld} = n_{s,l-1,d} + (\sigma - 1)n_{s,l-1,d-1}$$

is enforced by constraint (4.18c). Otherwise, (4.18c) becomes a trivial inequality.

Similar to the discussion for the MIP from section 3.2, constraints (4.19a)-(4.19b) enforce the non-decreasing property of cumulative L_{si} and U_{si} and constraints (4.20a)-(4.20b) are to eliminate symmetry.

Constraints (4.21a)-(4.21c) make sure that the number of mismatches in a partition is less than the size of the partition. Binary variable $\overline{w}_{qj}$ is 1 if and only if the number of mismatches for an error pattern q, a_{qj}, is greater than the length of at least one partition. $\underline{w}_{qj}$ would be 1 if in partition j of error pattern q, a_{qj} was not greater than m_j.

Constraints (4.21d)-(4.21f) make sure that every mismatch pattern is covered by at least one search. Constraint (4.21d) calculates A_{qsi} which is the cumulative number of mismatches up to and including iteration i of search s. If $\lambda_{qs} = 1$ in constraint (4.21e), then A_{qsi} is forced to be between lower bound L_{si} and upper bound U_{si} at every iteration i which means mismatch pattern q is covered by s. Constraint (4.21f) makes sure all applicable error patterns are covered. When the length of partition is smaller than errors in the partition, (4.21f) will not enforce (4.21e).This enables the MIP to produce solutions with partition sizes of zero which can be interpreted as determining the optimal number of partitions.

4.2.1 Optimal search scheme for variable size partition

In this section we present the result of the variable size partition MIP and the computational advantages achieved by implementing the variable size partition versus equal size partition MIP from section 3.2.

Table 4.1, for a number of relevant parameter values, $\overline{S} = 3$ and $R = 24$, shows the optimal number of edges found by the variable size MIP from section 4.2 in comparison to the equal size partition formulation from section 3.2. The highest reduction occurred for $\overline{P} = 3$ which was between 5% and 10%. The corresponding optimal search schemes are presented in Table 4.2. Column *partition size* demonstrates the size of partitions in the optimal search scheme. Partitions with a size of zero should be treated as they do not exist.

Table 4.1: Comparison between the objective values from equal size partition MIP and variable size partition MIP for $\overline{S}$=3, $R = 24$, and different values of K and $\overline{P}$.

	$K = 1$		$K = 2$		$K = 3$	
	variable-size obj	equal-size obj	variable-size obj	equal-size obj	variable-size obj	equal-size obj
$\overline{P} = 3$	515	563	12592	13189	222552	236887
$\overline{P} = 4$	515	515	12592	12834	222552	233956
$\overline{P} = 6$	515	515	12592	12731	222552	222552

Table 4.2: Search schemes found by variable size partition MIP for $K = 1, 2, 3$ and $\overline{P} = 3, 4, 5, 6$. $\overline{P}$ denotes the upper bound on number of partitions. This searches are used for experiments in Table 4.1. The schemes for $K = 1$ and $\overline{P} = 3, 4, 5$ plus $K = 2, 3$ and $\overline{P} = 3$ are optimal schemes. To control the running time of the MIP, the rest are best solutions found by running the MIP for 3 hours with $\overline{S} = 3$ and $R = 24$. These schemes are most probably optimal.

	$K = 1$		$K = 2$		$K = 3$	
	search scheme	partition size	search scheme	partition size	search scheme	partition size
$\overline{P} = 3$	$(213, 001, 001)$ $(321, 000, 011)$	$(6, 6, 12)$	$(123, 002, 012)$ $(231, 000, 012)$ $(321, 011, 022)$	$(5, 9, 10)$	$(123, 003, 023)$ $(231, 000, 123)$ $(321, 022, 033)$	$(4, 12, 8)$
$\overline{P} = 4$	$(1234, 0000, 0011)$ $(4321, 0001, 0011)$	$(6, 6, 11, 1)$	$(1234, 0011, 0222)$ $(3214, 0000, 0012)$ $(4321, 0002, 0112)$	$(10, 7, 2, 5)$	$(1234, 0003, 0223)$ $(3241, 00000, 1123)$ $(4321, 0022, 0333)$	$(4, 7, 5, 8)$
$\overline{P} = 5$	$(12345, 00001, 01111)$ $(54321, 00000, 00001)$	$(12, 1, 5, 5, 1)$	$(12345, 00002, 01222)$ $(23451, 00000, 01112)$ $(54321, 00011, 00022)$	$(5, 10, 2, 5, 2)$	$(12345, 00033, 02233)$ $(23415, 00000, 11233)$ $(43521, 00022, 03333)$	$(4, 3, 9, 8, 0)$
$\overline{P} = 6$	$(321456, 000001, 000011)$ $(654321, 000000, 001111)$	$(1, 0, 0, 11, 11, 1)$	$(231456, 000000, 000222)$ $(453216, 000011, 001112)$ $(654321, 000002, 011222)$	$(7, 0, 3, 3, 6, 5)$	$(321456, 000222, 000333)$ $(432156, 000000, 122233)$ $(543216, 000033, 023333)$	$(5, 2, 1, 12, 4, 0)$

4.3 Fixed General Edit Distance

In addition to mismatches, deletion and insertion are two important types of errors in DNA sequencing. Few million small insertion and deletion errors have been discovered in human genomes and the genetic variation caused by those indels is significant [44]. In this section we will introduce an MIP formulation to solve the optimal search scheme problem under edit distance assuming the size of the partitions are arbitrary and given.

For given K, R, P, $\overline{S}$, and sizes of partitions ,m_j, this MIP finds the search scheme with minimum number of sub-strings that have at most $\overline{S}$ searches. The optimal solution to the MIP provides the (π, L, U) and the number of nonempty searches in the optimal search scheme for edit distance.

4.3.1 Tries

To accommodate for edit distance error, we expand a read into another string and write the formulation for the extended read. First, we add K dummy placeholders prior to a *real character* for insertion. Second, we add another dummy placeholder after the *real character* to represent the deletion of that character. Figure 4.3 shows how to expand a read into an *extended read*. A solid line represents a *real character*, "I, M, and D" mark levels associated to the occurrence of an insertion, mismatch, or deletion, respectively. The set of K placeholders of type "I", an "M", and a "D" is called an *extended character*.

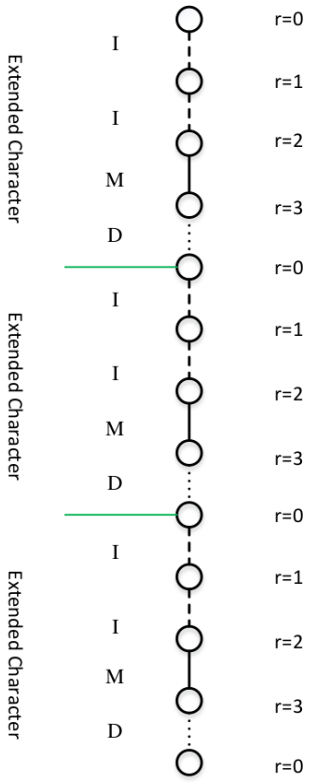


Figure 4.3: Extended characters of a read of size 3, accommodate for insertion and deletion errors.

Furthermore, we generalize the concept of a search trie to accommodate for edit distance by using extended characters and a new concept called node's *group type*. Wherever an error occurs we draw a diagonal edge. The position where an error occurs, determines the type of the error. If an error occurs at levels dedicated to insertion, deletion, or real character (the level for mismatch) the type of error will be designated as insertion, deletion, and mismatch, respectively.

Let g denote the *group type* for a node. $g = -K, \ldots, K$ and represents the net number of insertions. For instance, a node associated to a sub-string with 3 mismatches, 2 insertions, and 1 deletion would have a $g = 2 - 1 = 1$. Let l denote the depth of a node in a trie, we define n^g_{sld} as the number of nodes with exactly d cumulative errors with g net insertions (negative values represent net deletion) at the lth level of the trie of search s. Moreover, $n^0_{s00} = 1$, $n^g_{s00} = 0$ $g \neq 0$, $n^g_{s0d} = 0$

for $d \geq 1$, $n^g_{sld} = 0$ for $d < |g|$, and $n^g_{sl,-1} = 0$ for all values of g.

In order to determine of the type of an error designated to a level l, we define $r = l \ mod \ (K+2)$. $r = 0$ denotes deletion, $r = 1$ to K represent insertion, and $r = K + 1$ denotes mismatch. After an insertion, a parent node of n^g_{sld} type generates a child node of $n^{g+1}_{s,l+1,d+1}$ type. Similarly, when a deletion occurs a node of n^g_{sld} type generates a node of $n^{g-1}_{s,l+1,d+1}$ type. In order to have a unique way to represent $m < K$ number of insertions at an extended character, we establish a convention whereby for an error pattern with m insertions, the errors occur only at the very last m levels dedicated to insertion (the last m levels denoted by I in an extended characters).

Figure 4.4 represents the trie for a string of length 3 with maximum error $K = 1$ and $\sigma = 2$. The type of each node is written next to it. Also, the nodes in blue represent sub-strings of length 2. The number of those nodes times c_2 will be used in the objective function. c_2 denotes the probability of occurrence of a sub-string in the text (we will use 1 for computational experiments).

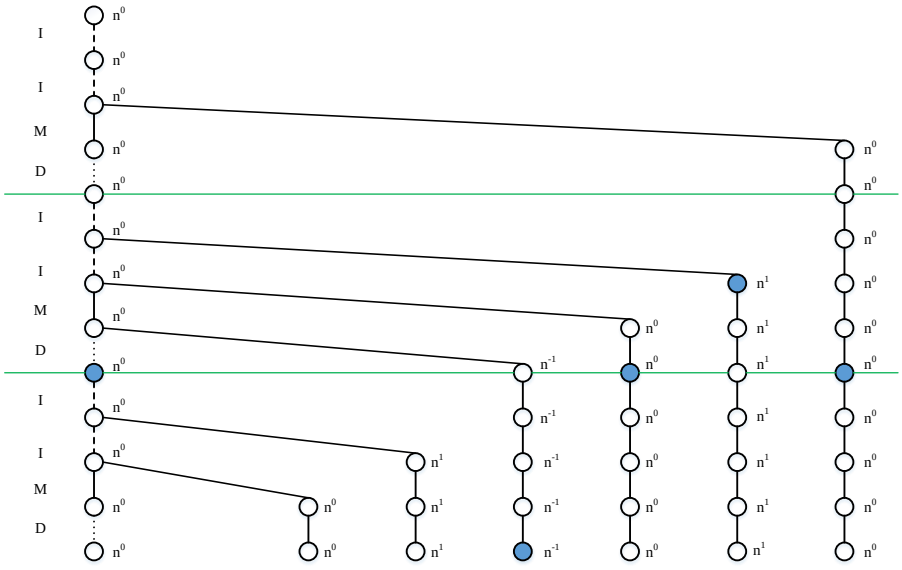


Figure 4.4: Generalized trie accommodates indel and mismatch errors for a read of size 3 with $K = 1$. The type of nodes are written next to them.

Due to occurrence of insertion and deletion, an error pattern can be produced in different ways through various combinations of insertion, deletion, and mismatch errors. We need to eliminate those strings in a trie that lead to the same pattern but with more errors. For instance, pattern "AAACA" after adding "T" right before "C" and then dropping "C" is the same as occurrence of only one mismatch at C's position where "C" would alter to "T". The following cases can be represented by their counterpart mismatch-only strings with fewer errors:

- Deletion at i^{th} extended character and insertion at i^{th} extended character.
- Deletion at i^{th} extended character and insertion at $(i+1)^{th}$ extended character.
- Deletion at i^{th} extended character and insertion at $(i-1)^{th}$ extended character.
- Deletion at i^{th} extended character and insertion at $(i+2)^{th}$ extended character.
- In addition to the cases mentioned above, we cannot delete a character and allow a mismatch to occur.

4.3.2 Formulation

The MIP formulation for edit distance, presented below, solves the optimal search scheme problem assuming P is given and pieces have arbitrary but given sizes. More specifically, for given maximum edit distance error K, R, P, and $\overline{S}$, this MIP finds the search scheme with minimum number of sub-strings with at most $\overline{S}$ searches.

$$\min \sum_{s=1}^{S} \sum_{h=1}^{R} c_h \left(\sum_{d=0}^{K} n^0_{s,h(K+2),d} + \sum_{g=1}^{K} \sum_{d=g}^{K} n^g_{s,h(K+2)-g(K+2)+K,d} + \sum_{v=h+1}^{R} \sum_{d=v-h}^{K} n^{h-v}_{s,v(K+2),d} \right) \tag{4.24}$$

subject to

$$\sum_{i=1}^{P} x_{sij} = 1 \qquad \text{for all } s \text{ and } j \tag{4.25a}$$

$$\sum_{j=1}^{P} x_{sij} = 1 \qquad \text{for all } s \text{ and } i \tag{4.25b}$$

$$\sum_{h=1}^{i} x_{shj} - \sum_{h=1}^{i} x_{sh,j-1} = t^+_{sij} - t^-_{sij} \qquad \text{for all } s,\ i = 2, \ldots, P-1,\ j = 1, \ldots, P+1 \tag{4.25c}$$

$$\sum_{j=1}^{P+1} (t^+_{sij} + t^-_{sij}) = 2 \qquad \text{for all } s,\ i = 2, \ldots, P-1 \tag{4.25d}$$

$$\sum_{h=1}^{i} \sum_{j=1}^{P} (K+1) m_j x_{shj} = \mu_{si} \qquad \text{for all } s \text{ and } i \tag{4.26a}$$

$$l - \mu_{s,i-1} \leq l y_{sil} \qquad \text{for all } s, i, \text{ and } l$$

(4.26b)

$$\sum_{l=1}^{R(K+2)} y_{sil} = R(K+2) - \mu_{s,i-1} \qquad \text{for all } s \text{ and } i$$

(4.26c)

$$\mathscr{L}_{sl} - L_{si} + (\mu_{si} - l) \leq (K + R(K+2))(1 - y_{sil} + y_{s,i+1,l}) \qquad \text{for all } s, i, \text{ and } l$$

(4.26d)

$$K(-1 + y_{sil} - y_{s,i+1,l}) \leq \mathscr{U}_{sl} - U_{si} \qquad \text{for all } s, i, \text{ and } l$$

(4.26e)

$$d - \mathscr{L}_{sl} + 1 \leq (R(K+2)+1)\underline{z}_{sld} \qquad \text{for all } s, l, \text{ and } d$$

(4.27a)

$$\mathscr{U}_{sl} + 1 - d \leq (K+1)\overline{z}_{sld} \qquad \text{for all } s, l, \text{ and } d$$

(4.27b)

$$\begin{aligned}\binom{l}{d}(\sigma)^d(\overline{z}_{sld} + \underline{z}_{sld} - 2) \leq\ & n^g_{sld} - n^g_{s,l-r,d} - \sigma n^{g-1}_{s,l-1,d-1} \\ & + \sigma n^g_{s,l-(r+K+2),d-2} + \sigma n^g_{s,l-(r+2(K+2)),d-2} \\ & + \sigma(\sigma-1) n^g_{s,l-(r+2(K+2)),d-3} \qquad \text{for all } s\text{ , } g,\ r = 1, ..., K,\ d \leq r\end{aligned}$$

(4.27c)

$$\binom{l}{d}(\sigma)^d(\overline{z}_{sld} + \underline{z}_{sld} - 2) \leq n^g_{sld} - n^g_{s,l-r,d} \qquad \text{for all } s\text{ , } g,\ r = 1, ..., K-1,\ d \geq r+1$$

(4.27d)

$$\binom{l}{d}(\sigma)^d(\overline{z}_{sld} + \underline{z}_{sld} - 2) \leq n^g_{sld} - n^g_{s,l-1,d} - (\sigma-1) n^g_{s,l-1,d-1} \qquad \text{for all } s,\ r = K+1$$

(4.27e)

$$\begin{aligned}\binom{l}{d}(\sigma)^d(\overline{z}_{sld} + \underline{z}_{sld} - 2) \leq\ & n^g_{sld} - n^g_{s,l-1,d} - n^{g+1}_{s,l-2(K+2),d-1} \\ & - (\sigma-1)\, n^{g+1}_{s,l-2(K+2),d-2} - n^{g+2}_{s,l-2(K+2),d-2} \qquad \text{for all } s\text{ , } g,\ r = 0\end{aligned}$$

(4.27f)

$$L_{si} \leq L_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1, \ldots, P-1 \tag{4.28a}$$

$$U_{si} \leq U_{s,i+1} \qquad \text{for all } s, \text{ and } i = 1, \ldots, P-1 \tag{4.28b}$$

$$L_{si} + K(\lambda_{qs} - 1) \leq \sum_{h=1}^{i} \sum_{j=1}^{P} a_{qj} x_{shj} \leq U_{si} + K(1 - \lambda_{qs}) \qquad \text{for all } q, s, \text{ and } i \tag{4.29a}$$

$$\sum_{s=1}^{S} \lambda_{qs} \geq 1 \qquad \text{for all } q \tag{4.29b}$$

$$x_{1PP} = 1 \tag{4.30a}$$

$$(S - s + 1)\, x_{s,1,j} + \sum_{t=s}^{S} \sum_{k=1}^{j-1} x_{t,1,k} \leq S - s + 1 \qquad \text{for all } s \text{ and } j = 2, \ldots, P \tag{4.30b}$$

$$n^{g}_{sld}, \mathscr{U}_{sl}, \mu_{si}, A_{qsi} \geq 0, \quad \mathscr{L}_{sl} \geq -R \qquad \text{for all } q, s, i, j, l, d, \text{ and } g \tag{4.31a}$$

$$L_{si}, U_{si} \geq 0 \quad \text{Integer} \qquad \text{for all } s \text{ and } i \tag{4.31b}$$

$$x_{sij}, \lambda_{qs}, y_{sil}, \overline{z}_{sld}, \underline{z}_{sld}, t^{+}_{sij}, t^{-}_{sij} \in \{0, 1\} \qquad \text{for all } q, s, i, j, l, \text{ and } d \tag{4.31c}$$

The objective function minimizes the expected number of sub-strings to be searched. The terms in the parenthesis count only the edges associated to sub-strings and of same length (as discussed for Figure 4.4). The first term counts sub-strings associated with mismatch and insertion. The second term counts the sub-strings associated to deletion.

x_{sij} in constraints (4.25a) and (4.25b) capture the assignment of pieces to iterations. In other words, at optimality, these variables determine π_s for the optimal search scheme. Constraints (4.25c) and (4.25d) ensure the connectivity of the pieces. More specifically, together these two constraints are essentially the linearization of the following constraint

$$\sum_{j=1}^{P} \left| \sum_{h=1}^{i} x_{shj} - \sum_{h=1}^{i} x_{sh,j-1} \right| = 2 \qquad \text{for all } s \text{ and } i \tag{4.32}$$

The term $\sum_{h=1}^{i} x_{shj}$ will have a binary value which denotes whether in search s partition j has been searched at any of iterations 1 to i of search s. The term $\sum_{h=1}^{i} x_{sh,j-1}$ captures the same notion for partition $j-1$. If all partitions form a connected block on the read at any iteration i, $\sum_{h=1}^{i} x_{shj} - \sum_{h=1}^{i} x_{sh,j-1}$ equals 1 only for one j, equals -1 only for one j, and 0 for all others. This is ensured by equation 4.32 and therefore for its linearization.

(4.26a)-(4.26e) determine the relationship between the levels of the trie and their corresponding iterations. Constraint (4.26a) calculates μ_{si}, the cumulative length of the pieces traversed in search s up to and including iteration i. With the help of binary variable y_{sil} in constraints (4.26b)-(4.26c), if $\mu_{s,i-1} < l \leq \mu_{si}$, the lower and upper bounds $\mathscr{L}_{sl}$ and $\mathscr{U}_{sl}$ are set to L_{si} and U_{si} by constraints (4.26d) and (4.26e), respectively.

Constraints (4.27a)-(4.27f) enforce the recursive equation needed to calculate the number of sub-strings. With the help of binary variables $\overline{z}_{sld}$ and $\underline{z}_{sld}$ in constraints (4.27a) and (4.27b), if $\mathscr{L}_{sl} \leq d \leq \mathscr{U}_{sl}$ the recursive formula is enforced by constraints (4.27c) to (4.27f). Otherwise, (4.27c) to (4.27f) become trivial inequalities.

More precisely, constraints (4.27c) to (4.27d) construct the trie for levels associated to insertion. The recursive function for these levels ($r = 1, ..., K$) resembles the recursive function in [7].

By convention, when the number of insertions m is less than K, we only consider the sub-string for which the insertions happen only at the last m places assigned to insertion. We add $d \leq r$ to guarantee this in (4.27c).

Furthermore, we eliminate the strings associated to error patterns that can be represented in

different ways through various occurrences of insertion, deletion, and mismatch errors. To exclude deletion at extended character i and insertion at $i+1$ we subtract $\sigma n^g_{s,l-(r+k+2),d-2}$ from the number of nodes in constraint (4.27c). To exclude deletion at extended character i and insertion at $i+2$, we subtract $n^g_{s,l-(r+2(k+2)),d-2}$ and $(\sigma-1)n^g_{s,l-(r+2(k+2)),d-3}$ from the number of nodes in constraints (4.27c). Constraint (4.27e)is associated to mismatch and it is basically the recursive function in [7].

Constraint (4.27f) generates the nodes at levels that deletion is allowed. We do not allow deletion and insertion/mismatch to happen at an extended character. Also, to prevent deletion at extended character i and insertion at $i-1$, we use $n^{g+1}_{s,l-2(K+2),d-1} + (\sigma-1)\ n^{g+1}_{s,l-2(K+2),d-2} + n^{g+2}_{s,l-2(K+2),d-2}$ instead of $n^{g+1}_{s,l-1,d-1}$. The first three terms calculate the number of nodes at level $l-(K+2)$ which do not have insertion at extended character $i-1$ (basically nodes at level $l-2(K+2)$ plus their children generated via either only mismatch or deletion).

Constraints (4.28a)-(4.28b) ensure L_{si} and U_{si} are non-decreasing as they are cumulative values. Constraints (4.29a)-(4.29b) ensure that every mismatch pattern is covered by at least one search. λ_{qs} is a binary variable that determines whether error pattern q is covered by search s. Constraint (4.29a) forces $\lambda=0$ if search s does not cover error pattern q and constraint (4.29b) ensures every error pattern is at least covered by one search. Constraints (4.30a)-(4.30b) are not necessary for the formulation , however, they eliminate some symmetries from the feasible region.

4.3.3 Optimal Search Schemes

In this section we present the result of the fixed size partition MIP for the edit distance. The optimal search schemes are presented in Table 4.3 for a variety of relevant parameter values K and P. The optimal solutions are identical to those of Hamming distance except for $K=2,\ P=3$ and $K=3,\ P=4$ plus all the cases with $K=4$. Even for those cases the optimal Hamming distance produces an objective function very close to the optimal solutions for edit distance. Consequently, using Hamming distance optimal search schemes for edit distance is justifiable (see section 3.5).

Table 4.3: Search schemes found by our edit distance MIP for $\overline{S} = 4$, $K = 1, 2, 3$ and $P = K + 1$, $P = K + 2$ and $P = K + 3$. To control the running time of MIP, the schemes for $K = 3$ and 4 are best solutions found by running the MIP for 3 hours with $\overline{S} = 4$. The solution for $K = 3$ and $P = 4$ is optimal . The rest of the schemes are most probably optimal for $\overline{S} = 4$.

	$K = 1$	$K = 2$	$K = 3$	$K = 4$
Optimal ($P = K + 1$)	$(12, 00, 01)$ $(21, 01, 01)$	$(321, 002, 022)$ $(213, 000, 012)$ $(123, 011, 012)$	$(1234, 0111, 1223)$ $(1234, 0000, 0033)$ $(2341, 0002, 0013)$ $(4321, 0003, 0233)$	$(12345, 01111, 33334)$ $(12345, 00000, 00444)$ $(54321, 00004, 03344)$
Optimal ($P = K + 2$)	$(123, 001, 001)$ $(321, 000, 011)$	$(2134, 0011, 0022)$ $(3214, 0000, 0112)$ $(4321, 0002, 0122)$	$(12345, 00022, 00333)$ $(43215, 00000, 11223)$ $(54321, 00003, 02233)$	$(123456, 000002, 033344)$ $(234561, 000000, 222334)$ $(564321, 000034, 004444)$
Optimal ($P = K + 3$)	$(1234, 0000, 0011)$ $(4321, 0001, 0011)$	$(21345, 00011, 00222)$ $(43215, 00000, 00112)$ $(54321, 00002, 01122)$	$(123456, 000003, 022233)$ $(234561, 000000, 111223)$ $(654321, 000022, 003333)$	$(1234567, 0000004, 0334444)$ $(6754321, 0000000, 3333334)$ $(6574321, 0000044, 0000444)$

4.4 Concluding Remarks

The formulation in section 3.2 solves OSS problem with the assumption of having equal size partitions for Hamming distance. In this chapter, we provided three formulations to relax those assumptions. These formulations also answered the open ended questions in [7]. However, either due to substantial computational resources needed for real world reads or the slight improvement of the solutions of these formulations over equal size partition formulation, we chose to use the results of equal size partition MIP for computational purposes throughout chapter 5.

5. TOWARDS A FULL-FLEDGED ALIGNER

Pure in-index search using standard backtracking is very slow for larger values of K. However, the drastic improvement over standard backtracking, gained by using our optimal search schemes in a bidirectional index, as observed in Section 3.5, demonstrate that there is a potential for significant improvement in the performance of any read mapper that utilizes in-index search. Some well-known full-fledged aligners, such as BWA-aln [21] and Bowtie1 [4], perform the search entirely in FM-index. Since Bowtie1 only performs the search for Hamming distance, in section 5.1 we compare OSS, BWA-aln, plus Bowtie1 and demonstrate that our optimal search schemes are superior to those aligners for Hamming distance.

Due to the exponential complexity of ASM using FM-index in terms of K, many state-of-the-art aligners do not perform ASM completely in index but rather use a combination of search in the index and verification in text using dynamic programming (DP).

Although OSS performs the search entirely in index, because of its tremendous performance in mapping Illumina reads to human genome, as demonstrated in Table 3.3, we will challenge it against full-fledged aligners benefiting from in-text verification. In order to cast a glance at the potential of combining OSS and in-text verification and its usage in full-fledged aligners, in section 5.2, we will gauge the performance of an implementation of OSS combined with in-text verification for Hamming distance versus OSS purely performed in index, also against backtracking plus in-text verification. Furthermore in section 5.3, we compare the performance of OSS, carrying out the search solely in index, against full-fledged aligners benefiting from combing in-index and in-text verification.

5.1 Computational Performance of OSS vs Full-Fledged In-Index Aligners

State-of-the-art aligners such as BWA-aln and Bowtie1, carry out the approximate string matching solely in FM-index. In this section, we observe that our optimal search schemes, OSS, outperforms those aligners. To get a better sense of the potential of OSS, we have compared our

optimal search schemes with Bowtie1 and BWA (bwa-aln) under Hamming distance. Since Bowtie1 is only capable of performing the search in Hamming distance, this comparison has been carried out for Hamming distance.

Bowtie1 has been set up to search for all alignments with at most K mismatches (`-v <K> -y -a`). We have also compared optimal search schemes against BWA in all-mapping mode (`bwa aln -N -O 0 -l<R> -n <K>`). The results shown in Table 5.1 demonstrate that our optimal search scheme is faster than Bowtie1 for $K = 1, 2$, and 3. In addition, OSS outperforms BWA for all values of K but $K = 1$, for which BWA is slightly faster.

An auxiliary data structure called array D, is the reason why BWA performs better than OSS for $K = 1$. Let us assume that we are mapping read R of size $|R|$, we define array D of size $|R|$ such that $D[i]$ represents a lower bound of the number of differences between string $R[0, i]$, first $i + 1$ characters of R, and the reference genome. Array D makes BWA faster by avoiding descending into some branches of the search trie. Imagine $D[0] = 0, D[1] = 1$ and we are performing depth first search, for the first character we go down the search trie but for the second character we will not go down because $D[1] = 1$. We ignore that branch and its child nodes and instead go into another branch of the search trie [21]. This makes the search space smaller and BWA faster. However, for $K = 1$ compared to $K \geq 2$ the number of edges avoided because of D, is a great number compared to the entire search trie. As K increases, the number of edges avoided during depth first search becomes significantly smaller in comparison to the number of edges in the search trie which deteriorate the effect of D.

Table 5.1: Running time comparison of searching all approximate matches of $100,000$ Illumina reads ($R = 101$) using OSS, Bowtie1, and BWA-aln for $K = 1, 2, 3$ and Hamming distance. The factor column is the speed-up ratio versus OSS in each category.

Dist.	Search Tool	$K = 1$		$K = 2$		$K = 3$	
		Time	Factor	Time	Factor	Time	Factor
Hamm.	OSS	5.23s	1	13.98s	1	50.33s	1
	Bowtie1	24.00s	4.58	92.00s	6.58	243.00s	4.82
	BWA	4.24s	0.81	14.93s	1.07	118.86s	2.36

5.2 Promising Combination of OSS and In-Text Verification

Approximate string matching using FM-index has exponential complexity in terms of K. Therefore, full-fledged aligners do not carry out the search completely in index. They employ a combination of in-index search and in-text verification. In order to take a glance at the possible improvement gained by combining OSS and in-text verification, we have implemented OSS and in-text verification for Hamming distance. In general, OSS combined with in-text verification accelerated the search by a factor of two. We also observed that OSS preserved its computational advantages compared to backtracking even in the presence of in-text verification. This indicated that apart from in-text verification, we gained performance solely through OSS.

In our execution of OSS and in-text verification, we traverse the search trie in a depth first manner, in index, till the number of occurrences for a sub-string in the search trie is small enough. We then switch into text verification simply by comparing genome and the read for all candidate locations corresponding to the sub-string. Moreover, in another strategy we perform the in-text verification only at the last iteration of the search. Table 5.2 shows the running time of optimal search schemes and in-text verification switching strategies whereby the criterion is set to having 25 and 50 occurrences of the sub-string as well as a switch criterion where the last block of each search is verified in the text. Using two different data sets for human genome, we observed that switching to in-text verification speeds up the mapping for all values of $K = 1, 2,$ and 3. Although switching at the last block speeds up the search for many full fledged aligners, when used along OSS, it is inferior to the other two criteria. Table 5.2 shows that switching to in-text verification wherever a node in the trie represents a sub-string with at most 25 or 50 occurrences, leads to a promising bi-fold speed up for the search.

Although we have shown that combining OSS and in-text verification leads to a superior performance in comparison to OSS alone, but we need to investigate whether the improvement gained only from in-text verification casts a shadow on OSS. In other words, can in-text verification alone compensate for the speed up gained through optimal search schemes? In order to answer this question, we have combined backtracking with in-text verification and compared it against

OSS plus in-text verification. Table 5.2 shows that OSS, without in-text verification, is much faster than backtracking plus in-text verification for both data sets and for all values of $K = 1, 2,$ and 3. This is a clear sign of the effectiveness of OSS for all sorts of aligner software packages. Surprisingly, Table 5.2 shows that adding in text verification slows down the backtrack search. This is due to the numerous candidate locations which need to be verified in text, therefore performing the search in index is a better choice. On average, a read of size 20 and 30 occurs about 50, and 25 times in human genome, respectively. In such a depth of a backtracking search tree, there are countless nodes which result in numerous candidate locations for in-text verification, making in-text verification significantly expensive. As a result of these observations, we infer that combing OSS and in-text verification is a viable proposition. Our observations in this section suggest that a full-fledged aligner that employs an intelligent combination of search in bidirectional FM-index using our optimal search schemes and in-text verification can outperform today's best approximate read mappers.

Table 5.2: Running time of optimal search schemes with $P = K + 1$ pieces for one mismatch and $P = K + 2$ pieces for two and three mismatches with in-text verification.

Strategy	$K = 1$		$K = 2$		$K = 3$	
	Time	Total in-text Verifications	Time	Total in-text Verifications	Time	Total in-text Verifications
	ERX1959065 (100K reads, 101 bps, hg38)					
OSS	5.87s	0	15.7s	0	55.96s	0
OSS-ITV$_{occ_{25}}$	3.38s	0.3 $\times 10^6$	7.74s	5.0 $\times 10^6$	33.77s	24.8 $\times 10^6$
OSS-ITV$_{occ_{50}}$	3.37s	0.4 $\times 10^6$	7.58s	6.2 $\times 10^6$	35.2s	34.5 $\times 10^6$
OSS-ITV$_{block}$	4.62s	9.3 $\times 10^6$	13s	9.7 $\times 10^6$	50.67s	16.0 $\times 10^6$
Backtrack	19.1s	0	226.83s	0	2032.14s	0
Backtrack-ITV$_{occ_{25}}$	18.64s	2.6 $\times 10^6$	293.02s	125.4 $\times 10^6$	3450.76s	2396.0 $\times 10^6$
Backtrack-ITV$_{occ_{50}}$	20.36s	5.2 $\times 10^6$	342.63s	244.8 $\times 10^6$	4348.43s	4509.2 $\times 10^6$
	SRR5365378 (1M reads, 125 bps, hg38)					
OSS	59.12s	0	126.82s	0	429.85s	0
OSS-ITV$_{occ_{25}}$	29.26s	2.4 $\times 10^6$	54.47s	22.7 $\times 10^6$	258.39s	127.8 $\times 10^6$
OSS-ITV$_{occ_{50}}$	28.99s	2.9 $\times 10^6$	53.34s	27.9 $\times 10^6$	274.78s	181.2 $\times 10^6$
OSS-ITV$_{block}$	33.38s	39.5 $\times 10^6$	105.45s	39.5 $\times 10^6$	382.89s	68.1 $\times 10^6$
Backtrack	169.47s	0	2019.67s	0	19917.23s	0
Backtrack-ITV$_{occ_{25}}$	163.88s	25.7 $\times 10^6$	2776.03s	1228.9 $\times 10^6$	36889.54s	23827.3 $\times 10^6$
Backtrack-ITV$_{occ_{50}}$	184.05s	52.2 $\times 10^6$	3457.74s	2405.5 $\times 10^6$	51580.86s	45040.1 $\times 10^6$

5.3 OSS implemented in index vs full fledged aligners

Although we witnessed performance gain from combining OSS and in-text verification in previous section, the combination was developed for Hamming distance. Therefore, to acquire a sense of the effectiveness of optimal search schemes, we decided to even challenge our optimal

search schemes by using them in a pure index-based search and compare the results against the full-fledged state-of-the-art aligners, that have the advantage of using a combination of in-index search and in-text verification for edit distance. We executed two sets of comparisons, all and strata mapping. We used 3 different data sets, two from human genome (100K reads of length 101 bps from ERX1959065 and another 1M reads of length 125 bps from SRR5365378) plus a data set for house mouse (500K reads of length 40 bps from SRR1270201).

For the first set of comparison, we compare OSS with BWA, Yara [45], as well as an available implementation of the 01^*0-filter scheme combined with dynamic programming (named Bwolo) [37] in all-mapping mode. We did these comparisons for edit distance only as all these tools work for edit distance. BWA was run with options `bwa aln -N -O` <K> `-l`<R> `-n` <K> `-i 0` (we did not use `bwa mem` because there is no way to impose maximum <K> edit distance error). Yara was run with options `-e` <K> `-s` <K> `-y full -t 1`. We note that Bowtie2 [46] is not designed with all-mapping in mind (for our data set, it did not terminate in 3 hours with default configuration and `-a` option). Moreover, imposing an all-mapping with maximum K errors in Bowtie2 in a way that its results are comparable to other tools is difficult. Bowtie2 settings used in [37] do not enforce this, and nonetheless, led to a very long running time. Consequently, we did not use Bowtie2 in this study.

For the first set of comparisons, we compared OSS with BWA, Yara, and Bwolo in all mapping mode. For the two data sets from human genome and $K = 1$ and 2, OSS outperformed other aligners with the exception of Bwolo for $K = 2$. This demonstrates that our optimal schemes are so effective that although the search is performed completely in index, the running times are comparable to full-fledged state-of-the-art aligners, which use a combination of in-index search and in-text verification. For $K = 3$, the benefit of using in-text verification in full-fledged aligners catches up and thus outperform OSS which carries out the search entirely in index. Specifically, Bwolo takes advantage of DP in an efficient way. Its search tries produce long seeds with considerably low number of leaves which reduces the number of DP performed. Furthermore, its search tries are basically $01 * 0$ seeds that can be searched in FM-index without suffering from function calls overheads caused via

a recursive implementation.

Bwolo search schemes are not optimal in terms of total number of edges but work better when combined with in-text verification. This implies that, in order to design search schemes to be utilized in conjunction with in-text verification, the objective function of an MIP should incorporate the in-index and in-text computational expenses. Consequently, a new optimization problem needs to be solved.

For the reads of length 40 from mouse genume and $K = 1$, BWA and Yara outperformed OSS while for $K = 2$, only BWA outperformed OSS. According to Table 5.3, for $K = 3$ BWA and Yara lost their edge over OSS and only Bwolo, benefiting from dynamic programming, outperformed OSS by a factor of 2. Since mouse genome is smaller than human genome, for $K = 1$ and 2, there are fewer candidate locations to be verified using dynamic programming saving considerable time for BWA and Yara. This edge diminishes when K increases which in turn, increases the number of verifications in the text. For $K = 3$, OSS is faster than other aligners and with incorporating text verification it can outperforms Bwolo. For this data set and $K = 3$, Yara was not able to conduct the task due to consuming more than $112\ GB$ of memory.

We find these results for our optimal search schemes very impressive. To our knowledge, this is the first time that, for $K = 1$ and $K = 2$, ASM of reads of these sizes performed completely in index has been reported to compete in running time with the best full-fledged aligners, which use combination of index search and in-text DP verification. This implies the power of our optimal search schemes. We note that the results are even better for strata mode in Table 5.4. We performed a second set of comparisons with BWA and Yara in strata mode. The 0-strata search means we first search the reads with 0 errors, then search all the reads with no exact match, with 1 error, and so on, until K is reached. This strategy can be generalized to s-strata, where $s \leq K$. This means that, for $b = 0$ to $K - s$, for all reads with a b-error best match, we compute all occurrences with up to $b + s$ errors. We ran Yara in 1-strata mode using `-e <K> -s 1 -y full -t 1` and BWA with `bwa aln -O <K> -l<R> -n <K> -i 0` arguments and compared the running times with OSS. BWA approximately mimics 1-strata mode because it disregards reads with too many

Table 5.3: Running time, all mapping.

Dist.	Search Tool	$K=1$		$K=2$		$K=3$	
		Time	Factor	Time	Factor	Time	Factor
		ERX1959065 (100K reads, 101 bps, hg38)					
Edit.	OSS	7.8s	1.00	90.09s	1.00	1064.42s	1.00
	BWA	11.19s	1.43	225.71s	2.51	5101.78s	4.79
	Yara	910.31s	116.71	1013.18s	11.25	1073.68s	1.01
	Bwolo	14.79s	1.90	47.52s	0.53	153.71s	0.14
		SRR5365378 (1M reads, 125 bps, hg38)					
	OSS	66.55s	1.00	693.59s	1.00	9467.15s	1.00
	BWA	79.63s	1.20	1651.04s	2.38	43321.95s	4.58
	Yara	848.74s	12.75	1294.09s	1.87	1538.84s	0.16
	Bwolo	115.50s	1.74	300.21s	0.43	834.71s	0.09
		SRR1270201 (500K reads, 40 bps, mm10)					
	OSS	153.81s	1.00	2706.32s	1.00	25898.90s	1.00
	BWA	40.96s	0.27	1211.58s	0.45	36923.71s	1.43
	Yara	47.07s	0.31	9268.10s	3.42	*	*
	Bwolo	690.55s	4.49	3023.83s	1.12	12498.60s	0.48

$\star$ represents an incomplete run due to memory overflow.

possible alignments on the genome which in return makes it faster. For both data sets from human genome and for all values of K OSS outperformed Yara. In contrast, for the same data sets and $K = 1, 2,$ and 3 BWA , benefiting from disregarding repetitive reads, outperformed OSS.

Although Yara benefits from in-text verification, for $500K$ reads of size $40bps$ from mouse genome, OSS was multiple times faster for $K = 1, 2,$ and 3. For $K = 1$ and 2 BWA performs better than OSS. This superior performance happens because of two reasons: 1) shorter reads align to many places and BWA ignores reads mapping to numerous location on genome 2) the use of array D (for a detailed discussion see section 5.1). For $K = 3$ the effectiveness of array D diminishes and therefore OSS outperformed BWA by a factor of 1.85.

Of course, we did not expect to be able to outperform full-fledged aligners for larger values of K, because for larger K and these read lengths, verification in index is too costly, especially if the number of successful verifications is low. Although Vroland et al.[37] raised the option of using a bidirectional index for verification, they only used in-text DP verification for the last pieces as they had only a unidirectional index at hand. Nevertheless, for their data set (40bp, exactly 3

Table 5.4: Running time, strata mapping .

Dist.	Search Tool	$K = 1$		$K = 2$		$K = 3$	
		Time	Factor	Time	Factor	Time	Factor
		ERX1959065 (100K reads, 101 bps, hg38)					
Edit.	OSS	7.97s	1.00	12.14s	1.00	26.57s	1.00
	BWA	4.05s	0.51	9.90s	0.82	18.15s	0.68
	Yara	706.44s	88.64	982.03s	80.89	1033.51s	38.90
		SRR5365378 (1M reads, 125 bps, hg38)					
	OSS	65.73s	1.00	142.83s	1.00	569.99s	1.00
	BWA	30.78s	0.47	74.95s	0.52	264.21s	0.46
	Yara	817.53s	12.44	926.18s	6.48	1322.64s	2.32
		SRR1270201 (500K reads, 40 bps, mm10)					
	OSS	149.95s	1.00	187.27s	1.00	414.82s	1.00
	BWA	9.51s	0.06	62.72s	0.33	768.59s	1.85
	Yara	843.50s	5.63	1765.60s	9.43	15500.00s	37.37

errors), pure index-based search using our optimal search schemes outperforms Bwolo by a factor of almost 1.5 (data not shown), so the read length matters. Although the optimal scheme found by our MIP is superior to the 01^*0 scheme in [37] for search in the index, for a larger K, one has to take into account the number of remaining verifications versus the number of edges in the trie for the remaining pieces. If that ratio is low, it does not pay off to verify in the index instead of verification in the text as our comparisons showed.

6. CONCLUSION AND FUTURE RESEARCH

In this chapter, we highlight the main achievements of this research and suggest intresting research topics for the future.

6.1 Conclusion

In this research, we contributed to the approximate string matching research as follows:

1. We proposed, for the first time, a method to solve the optimal search scheme problem for ASM-B for Hamming distance (using a MIP formulation).

2. We demonstrated that our MIP approach can solve the optimum search scheme problem to optimality in a reasonable amount of time for input parameters of considerable size, and enjoys very quick convergence to optimal or near-optimal solutions for input parameters of larger size.

3. We showed that approximate search in a bidirectional FM-index can be performed significantly faster if the optimal schemes obtained from our MIP are used. This was demonstrated based on number of edges in the search tries as well as actual running time of in-index search on real Illumina reads (up to 35 times faster than standard backtracking for 3 errors). We also showed that although our MIP solutions are for Hamming distance they perform equally well for edit distance.

4. We showcased the power of our optimal search schemes by demonstrating that for $K = 1$ and 2 errors, approximate string matching of reads of size $R = 101$ performed completely in index compete in running time with the best full-fledged aligners, which benefit from combining search in index with in-text dynamic programming verification. This suggests that a full-fledged aligner that intelligently combines search in bidirectional index using our optimal search schemes with in-text verification using DP can outperform today's best approximate aligners.

6.2 Future Research

Moreover, our approach in this research has raised some interesting open problems:

1. Our computational experiments in Section 3.4 showed that our current MIP has two attractive properties: the early solutions it finds are optimal or near-optimal, and its optimal search scheme is insensitive to the value of R (we ask: *"is this insensitivity to R a theoretically provable fact?"*). This makes our current MIP quite powerful in practice because, even if all input parameters K, R, P, $\overline{S}$ are quite large, we can run the MIP for a short time with a much smaller R to get a solution that is most probably optimal or near-optimal for the original problem. Nevertheless, solving the MIP completely to ascertain optimality is of great interest and currently consumes considerable computational resources for large instances, especially when $\overline{S} > 5$, $K > 4$, $P > 6$, $R > 100$. We ask *"can the solution time be improved by introducing other MIP formulations, or strengthening the current formulation using strong cutting planes or further elimination of symmetric solutions?"*

2. We demonstrated that the verification of few occurrences with high errors in the index is worse than in-text DP verification. We ask *"what is the best point to stop verification in the index and start verifying in the text instead?."* This can be individually decided for each pattern.

3. We have provided the MIP that solves OSS for Hamming distance. Those search schemes might not be optimal when in-index search is combined with in-text verification. In order to design search schemes to be utilized in conjunction with in-text verification, the objective function of an MIP should incorporate the in-index and in-text computational expenses. We ask *"what is the MIP formulation that produces the optimal searches when in-index search is combined with in-text verification?."*

REFERENCES

[1] https://www.news-medical.net/life-sciences/What-is-Transcription.aspx. (Accessed on Nov. 18, 2018).

[2] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter, "The structure and function of dna," in *Molecular Biology of the Cell. 4th edition*, Garland Science, 2002.

[3] Y. Lu, Y. Shen, W. Warren, and R. Walter, "Next generation sequencing in aquatic models," *Next Generation Sequencing-Advances, Applications and Challenges*, pp. 61–79, 2016.

[4] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.

[5] C. Pockrandt, M. Ehrhardt, and K. Reinert, "EPR-Dictionaries: A Practical and Fast Data Structure for Constant Time Searches in Unidirectional and Bidirectional FM Indices," in *RECOMB '17*, pp. 190–206, 2017.

[6] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu, "High throughput short read alignment via bi-directional bwt," in *IEEE BIBM '09*, pp. 31–36, 2009.

[7] G. Kucherov, K. Salikhov, and D. Tsur, "Approximate string matching using a bidirectional index," *Theoretical Computer Science*, vol. 638, pp. 145–158, 2016.

[8] K. A. Wetterstrand, "Dna sequencing costs: data from the nhgri genome sequencing program (gsp)," 2013.

[9] S. Goodwin, J. D. McPherson, and W. R. McCombie, "Coming of age: ten years of next-generation sequencing technologies," *Nature Reviews Genetics*, vol. 17, no. 6, p. 333, 2016.

[10] E. R. Mardis, "Next-generation sequencing platforms," *Annual review of analytical chemistry*, vol. 6, pp. 287–303, 2013.

[11] H. Buermans and J. Den Dunnen, "Next generation sequencing technology: advances and applications," *Biochimica et Biophysica Acta (BBA)-Molecular Basis of Disease*, vol. 1842, no. 10, pp. 1932–1941, 2014.

[12] M. L. Metzker, "Sequencing technologies—the next generation," *Nature reviews genetics*, vol. 11, no. 1, p. 31, 2010.

[13] K. Reinert, B. Langmead, D. Weese, and D. J. Evers, "Alignment of next-generation sequencing reads," *Annual review of genomics and human genetics*, vol. 16, pp. 133–151, 2015.

[14] R. Baeza-Yates and G. Navarro, "Fast approximate string matching in a dictionary," in *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No. 98EX207)*, pp. 14–22, IEEE, 1998.

[15] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[16] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of discrete algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[17] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 390–398, IEEE, 2000.

[18] G. L. Nemhauser and L. A. Wolsey, *Integer and combinatorial optimization*. New York: Wiley, 1988.

[19] L. A. Wolsey, *Integer programming*. New York: Wiley, 1998.

[20] K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, G. Urgese, and D. Weese, "The SeqAn C++ template library for efficient sequence analysis: A resource for programmers," *Journal of Biotechnology*, vol. 261, pp. 157–168, 2017.

[21] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[22] https://www.acegamsat.com/gamsat-biology-the-cell. (Accessed on Nov. 18, 2018).

[23] L.-Y. Fu, G.-Z. Wang, B.-G. Ma, and H.-Y. Zhang, "Exploring the common molecular basis for the universal dna mutation bias: Revival of löwdin mutation model," *Biochemical and biophysical research communications*, vol. 409, no. 3, pp. 367–371, 2011.

[24] https://www.genome.gov/27541954/dna-sequencing-costs-data/. (Accessed on Nov. 18, 2018).

[25] J. Zhang, R. Chiodini, A. Badr, and G. Zhang, "The impact of next-generation sequencing on genomics," *Journal of genetics and genomics*, vol. 38, no. 3, pp. 95–109, 2011.

[26] C. Meldrum, M. A. Doyle, and R. W. Tothill, "Next-generation sequencing for cancer diagnostics: a practical perspective," *The Clinical Biochemist Reviews*, vol. 32, no. 4, p. 177, 2011.

[27] K. Frese, H. Katus, and B. Meder, "Next-generation sequencing: from understanding biology to personalized medicine," *Biology*, vol. 2, no. 1, pp. 378–398, 2013.

[28] https://www.illumina.com/documents/products/illumina_sequencing_introduction.pdf. (Accessed on Nov. 18, 2018).

[29] U. Manber and E. W. Myers, "Suffix arrays: a new method for on-line string searches," in *SODA '90*, pp. 319–327, 1990.

[30] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[31] M. G. Maaß, "Linear bidirectional on-line construction of affix trees," *Algorithmica*, vol. 37, no. 1, pp. 43–74, 2003.

[32] D. Strothmann, "The affix array data structure and its applications to rna secondary structure analysis," *Theoretical Computer Science*, vol. 389, no. 1-2, pp. 278–294, 2007.

[33] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *FOCS '00*, pp. 390–398, 2000.

[34] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep. 124, Digital SRC Research Report, 1994.

[35] K. Reinert, B. Langmead, D. Weese, and D. J. Evers, "Alignment of Next-Generation Sequencing Reads.," *Annual review of genomics and human genetics*, vol. 16, pp. 133–151, 2015.

[36] J. Karkkainen and J. C. Na, "Faster filters for approximate string matching," in *ALENEX '07*, pp. 84–90, 2007.

[37] C. Vroland, M. Salson, S. Bini, and H. Touzet, "Approximate search of short patterns with high error rates using the 01*0 lossless seeds," *Journal of Discrete Algorithms*, pp. 3–16, 2016.

[38] https://www.cs.jhu.edu/ langmea/resources/lecture_notes/bwt_and_fm_index.pdf. (Accessed on Nov. 18, 2018).

[39] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 841–850, Society for Industrial and Applied Mathematics, 2003.

[40] C. Pockrandt, M. Ehrhardt, and K. Reinert, "Epr-dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional fm indices," in *International Conference on Research in Computational Molecular Biology*, pp. 190–206, Springer, 2017.

[41] T. Schnattinger, E. Ohlebusch, and S. Gog, "Bidirectional search in a string with wavelet trees," in *Annual Symposium on Combinatorial Pattern Matching*, pp. 40–50, Springer, 2010.

[42] D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen, "Versatile succinct representations of the bidirectional burrows-wheeler transform," in *European Symposium on Algorithms*, pp. 133–144, Springer, 2013.

[43] IBM-ILOG, "Cplex 12.7.1, https://www.ibm.com/support/knowledgecenter/en/sssa5p_12.7.1/ilog.odms.studio.help/ optimization_studio/topics/cos_home.html." (Accessed on Nov. 2, 2017).

[44] J. M. Mullaney, R. E. Mills, W. S. Pittard, and S. E. Devine, "Small insertions and deletions (indels) in human genomes," *Human molecular genetics*, vol. 19, no. R2, pp. R131–R136, 2010.

[45] E. Siragusa, *Approximate string matching for high-throughput sequencing*. PhD thesis, Freie Universität Berlin, 2015.

[46] B. Langmead and S. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

APPENDIX A

FINDING OPTIMUM SEARCH SCHEMES FOR APPROXIMATE STRING MATCHING USING MIXED INTEGER PROGRAMMING

This program/code finds the optimum search schemes for approximate string matching using bidirectional FM-index using the approach described in the paper Kianfar, K., Pockrandt, C., Torkamandi, B., Luo, H., Reinert, K., Optimum Search Schemes for Approximate String Matching Using Bidirectional FM-Index, 2018. Any commercial use is prohibited.

The MIP can have different solutions with the same objective values. It is also possible that multiple runs of one problem, which do not reach optimality (i.e., run time equals run time limit), result in slightly different solutions and objective values.

There are two options to run the program in Linux: run the executive file or build from source code.

For both options first execute the following:

```
$ git clone https://github.com/kianfar77/OptimumSearchSchemes.git
$ cd OptimumSearchSchemes/MIPCode/ && ls
```

Run the executive file

- The executable has been generated using CPLEX 12.7.1.
- Before running the program, make sure there is a folder named "output" in application directory.
- Do not delete or modify parameters.txt
- In application directory, type the following commands in terminal

```
$ chmod +x EqualFix
$ ./EqualFix -s <upperbound on number of searches> -p <number of
```

```
parts> -k <maximum error> -r <read length> [-t <time limit>]
[-lowerK= <minimum number of errors>][-lp][-verbose][-sigma]
[-h|-help]
```

Build from source code

- Make sure you have the following files in "libs" folder (you can acquire the latest version on IBM download page for students/faculties, see the section "required libraries" below)

1. cplex.h
2. cpxconst.h
3. libcplex.a
4. libilocplex.a
5. libcplexdistmip.a

- Before running the program, make sure there is a folder named "output" in source directory.
- In source files' directory, type the following commands in terminal

```
$ make
$ ./EqualFix -s <upperbound on number of searches> -p <number of
parts> -k <maximum error> -r <read length> [-t <time limit>]
[-lowerK= <minimum number of errors>][-lp][-verbose][-sigma]
[-h|-help]
```

Required libraries

- CPLEX 12.7.1 is used for the paper. If not available, the latest academic version, i.e., CPLEX 12.8, available at IBM download page for students/faculties, can be used with no significant difference in run times.

- Install CPLEX under super user

```
$ chmod +x cplex_file_you_have_downloaded.bin

$ ./cplex_file_you_have_downloaded.bin
```

- After installations, say, in the path /opt/ibm/ILOG/CPLEX_Studio1271,
  - cplex.h and cpxconst.h would be located at /opt/ibm/ILOG/CPLEX_Studio1271/cplex/include/ilcplex
  - libcplex.a, libilocplex.a, and libcplexdistmip.a would be located at /opt/ibm/ILOG/CPLEX_Studio1271/cplex/lib/x86-64_linux/static_pic/

Arguments of the program

-s:	Upperbound on the number of searches. [required]
-p:	Number of parts (partitions). [required]
-k:	Number of maximum errors. [required]
-r:	Length of read. [required]
-t:	Upperbound on run time. [optional, defualt= 3.0 hours]
–lowerK:	Number of minimum errors. [optional, defualt= 0]
–lp:	Prints out the MIP formulation. [optional]
–verbose:	Prints out the progress of objective value, rows starting with * correspond to feasible solutions. [optional]
–sigma:	Alphabet size. [optional, defualt= 4]
-h, –help:	Displays help.

Usage

```
$ ./EqualFix -help

$ ./EqualFix -s 3 -p 3 -k 2 -r 12

$ ./EqualFix -s 3 -p 3 -k 2 -r 12 -t 0.5
```

```
$ ./EqualFix -s 3 -p 3 -k 2 -r 12 -t 0.5 -verbose -lowerK=1 -sigma=2
$ ./EqualFix -s 3 -p 3 -k 2 -r 12 -t 0.5 -verbose -lp > ./output/verbose.txt
```

Output

- The objective value and execution time (only for solver) will be printed on the screen.
- The program generates "output.txt" in "output" folder.
- "output.txt" contains sigma (alphabet size), S (upper bound on number of searches), P (number of parts), lowerK (minimum number of errors), K (maximum number of errors), R (read length), run time (sec), and Objective (number of edges in all tries). Also for each search, there are U's (upper bounds on errors in different parts), x's (the order in which a search processes the parts), and L's (lower bounds on errors in different read's parts). There would be P elements in x's , L's and U's. First element of x shows the partition (part) being searched at iteration 1, second element of x shows the partition covered at iteration 2, and so forth. Similarly for L and U, the ith elements represent the number of errors in the ith element of x, the part of the read, processed at ith iteration. In other words, the ith elements of x, L, and U are associated with iteration i.
- The program generates "LP.lp" in "output" folder if –lp is recieved. "LP.lp" contains the mixed integer program.
- If detailed verbose is requested, The progress of feasible solutions toward the optimal solution will be printed on the screen (results in a longer run time). In order to save detailed verbose messages you may run the program by typing ./EqualFix -s -p -k -r > ./output/verbose.txt. In detailed verbose setting, rows starting with "*" represent a feasible integer solution. By looking at those rows one can observe how fast the program converges to the optimal solution (see verboseExample.png).