STUDYING THE EFFECTS OF SAMPLING ON THE EFFICIENCY AND ACCURACY OF K-MER INDEXES

By

Meznah Almutairy

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2017

ABSTRACT

STUDYING THE EFFECTS OF SAMPLING ON THE EFFICIENCY AND ACCURACY OF K-MER INDEXES

By

Meznah Almutairy

Searching for local alignments is a critical step in many bioinformatics applications and pipelines. This search process is often sped up by finding shared exact matches of a minimum length. Depending on the application, the shared exact matches are extended to maximal exact matches, and these are often extended further to local alignments by allowing mismatches and/or gaps. In this dissertation, we focus on searching for all maximal exact matches ($MEMs$) and all highly similar local alignments ($HSLAs$) between a query sequence and a database of sequences. We focus on finding $MEMs$ and $HSLAs$ over nucleotide sequences.

One of the most common ways to search for all $MEMs$ and $HSLAs$ is to use a k-mer index such as BLAST. A major problem with k-mer indexes is the space required to store the lists of all occurrences of all k-mers in the database. One method for reducing the space needed, and also query time, is sampling where only some k-mer occurrences are stored.

We classify sampling strategies used to create k-mer indexes in two ways: how they choose k-mers and how many k-mers they choose. The k-mers can be chosen in two ways: *fixed sampling* and *minimizer* sampling. A sampling method might select enough k-mers such that the k-mer index reaches full accuracy. We refer to this sampling as *hard sampling*. Alternatively, a sampling method might select fewer k-mers to reduce the index size even further but the index does not guarantee full accuracy. We refer to this sampling as *soft sampling*. In the current literature, no systematic study has been done to compare the

different sampling methods and their relative benefits/weakness.

It is well known that fixed sampling will produce a smaller index, typically by roughly a factor of two, whereas it is generally assumed that minimizer sampling will produce faster query times since query k-mers can also be sampled. However, no direct comparison of fixed and minimizer sampling has been performed to verify these assumptions. Also, most previous work uses *hard sampling*, in which all similar sequences are guaranteed to be found. In contrast, we study *soft sampling*, which further reduces the k-mer index at a cost of decreasing query accuracy.

We systematically compare fixed and minimizer sampling to find all $MEMs$ between large genomes such as the human genome and the mouse genome. We also study soft sampling to find all $HSLAs$ using the NCBI BLAST tool with the human genome and human ESTs. We use BLAST, since it is the most widely used tool to search for $HSLAs$. We compared the sampling methods with respect to index size, query time, and query accuracy.

We reach the following conclusions. First, using larger k-mers reduces query time for both fixed sampling and minimizer sampling at a cost of requiring more space. If we use the same k-mer size for both methods, fixed sampling requires typically half as much space whereas minimizer sampling processes queries slightly faster. If we are allowed to use any k-mer size for each method, then we can choose a k-mer size such that fixed sampling both uses less space and processes queries faster than minimizer sampling. When identifying $HSLAs$, we find that soft sampling significantly reduces both index size and query time with relatively small losses in query accuracy. The results demonstrate that soft sampling is a simple but effective strategy for performing efficient searches for $HSLAs$. We also provide a new model for sampling with BLAST that predicts empirical retention rates with reasonable accuracy.

To my parents, my husband, and my children.

ACKNOWLEDGMENTS

It is my earnest desire to express my gratitude and appreciation for my dissertation adviser Dr. Eric Torng who has provided constant guidance and encouragement throughout my graduate studies. His guidance was essential to the completion of this dissertation and has taught me innumerable lessons and insights on the academic research.

I would like to acknowledge my dissertation committee members Dr. Yanni Sun, Dr. Kevin Liu, and Dr. Jason Gallant for their expertise and suggestions that have significantly improved this dissertation. Also, I would like to acknowledge the support of the Michigan State University High Performance Computing Center and the Institute for Cyber Enabled Research.

I am truly grateful to my parents Rasheed and Haya for their everlasting love and prayers. I wish to give my heartfelt thanks to my husband, Nasser, who always believed in me. Lastly, I am deeply thankful to my children Joud, Jenna, and Ayah for bringing smiles to my heart and joys to my spirit.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Sequence similarity search

With each passing year, advances in sequencing technologies, such as ROCHE/454, Illumina/Solexa, and Pacific Biosciences (PacBio), are producing DNA sequences both faster and cheaper. Right now, the average number of sequences generated from one sequencing run is on the order of hundreds of millions to billions.

While this explosive growth in DNA datasets yields exciting new possibilities for biologists, the vast size of the datasets also presents significant challenges for many compute-intensive biology applications. These applications include homogenous search [69, 6, 104, 56], detection of single nucleotide polymorphisms (SNP) [34, 78, 65], mapping cDNA sequences against the corresponding genome [36, 66, 99]. sequence assembly [82, 70, 71], sequence clustering [25, 28, 46], and sequence classification [98, 7, 23]. A core operation in all these applications is to search the dataset for sequences that are similar to a given query sequence. The similarity between a dataset sequence and a query sequence may be defined at the sequence to sequence, sub-sequence to sequence, or sub-sequence to sub-sequence level. These similarity search levels are also known as searching for global alignments, semi-global alignments, and local alignments, respectively. In this dissertation, we focus on applications that rely on finding sub-sequence to sub-sequence similarity levels such as homogenous search,

sequence mapping, and SNP detections.

1.2 Maximal exact matches and highly similar local alignments

Searching for local alignments is a critical step in many bioinformatics applications and pipelines. This search process is often sped up by finding shared exact matches of a minimum length. The value of minimum length is usually set to ensure all the desired similar sequences are recognized. Depending on the application, the shared exact matches are extended to maximal exact matches [91, 38], and these are often extended further to local alignments by allowing mismatches and/or gaps [92, 56, 75, 36].

In this dissertation, we focus on searching for all *maximal exact matches* ($MEMs$) and all *highly similar local alignments* ($HSLAs$) between a query sequence and a database of sequences. We focus on finding $MEMs$ and $HSLAs$ over nucleotide sequences where nucleotides are represented by A, C, G, and T. The $MEMs$ and $HSLAs$ are commonly used in applications that compare sequences within the same species or closely related species.

1.3 Overview of searching methods

1.3.1 Sequence alignments types

In bioinformatics, a sequence alignment is a procedure of arranging nucleotide or protein sequences to determine regions of similarity. These alignments are usually used to find functional, structural, or evolutionary relationships between the sequences. Sequence alignments

are also used for non-biological sequences, such as computing the similarity level between strings in a natural language, text mining, and financial data.

Sequence alignments can be classified into global and local alignments. Global alignments try to align every residue in every sequence. The Needleman-Wunsch algorithm [64] is a traditional global alignment method that is based on dynamic programming. On the other hand, local alignments try to find regions of similarity between sequences. The Smith-Waterman algorithm [84] is a traditional local alignment method that is also based on dynamic programming. Semi-global alignments [14, 53] are hybrid methods that try to align one sequence entirely within the other sequence. These are especially useful when one sequence is short, and the other is very long. Many of semi-global alignment algorithms are extended from global alignment algorithms such as the modified Needleman-Wunsch algorithm [53]. In these methods, gaps at the beginning and/or end of an alignment (also known as starting or trailing spacing) are ignored.

Sequence alignments can also be classified into pairwise sequence alignment and multiple sequence alignment. In pairwise sequence alignment, we align two sequences. Some standard pairwise sequence alignment methods are the Needleman-Wunsch algorithm [53], the Smith-Waterman algorithm [84], FASTA [69], BLAST and its improved versions [6, 104, 57, 17]. The pairwise sequence alignments are usually computed using a 2-dimensional matrix. Multiple sequence alignment methods align more than two sequences at a time. A common way to perform multiple sequence alignments is to generalize pairwise sequence alignment methods. One way to generalize pairwise sequence alignment methods is to use an n-dimensional matrix instead of a 2-dimensional matrix to align n sequences instead of 2 sequences. This technique is computationally expensive but is guaranteed to find a global optimum solution and is only used when for small n. A more practical solution for large n is to use heuristics

such as progressive alignments as in ClustalW [88] MUSCLE [24] MAFFT [35]. Progressive alignment methods use a guide tree to reduce the multiple sequence alignments problem into a series of 2-dimensional pairwise sequence alignment problems.

Another way to align sequences is to do structural alignments such as DALI [32], SSAP [87], and Combinatorial Extension methods [81]. Structural alignments are commonly used for protein sequences and sometimes RNA sequences. They align sequences using information about the secondary and tertiary structure. These methods typically find local alignments and can be applied to two or more sequences.

1.3.2 Online searching methods

Searches can be online or offline. We first describe the online searching methods. The online methods produce the best alignments but are slow use dynamic programming. The Needleman-Wunsch algorithm [53] and Smith-Waterman algorithm [84] are based on dynamic programming and find global and local alignments, respectively.

Faster heuristic methods are often used as dynamic programming requires quadratic time complexity. One heuristic, known as seed-end-extend, first finds seeds, word-to-word matches of a given length, and then tries to extend these seeds into local alignments. FASTA[50, 69] was one of the earliest seed-and-extend searching tools. Today, the dominant method for sequence comparison is the Basic Local Alignment Search Tool, or BLAST [5]. Given a query, BLAST performs a linear scan over the sequence database searching for a set of seeds/words shared with some substrings of the query. It then extends them in both directions until the accumulated similarity score begins to decrease. Finally, BLAST reports these matching regions with high statistical significance. Further improvements of BLAST include MegaBLAST [104], MPBLAST [40], and miBLAST [39]. MegaBLAST is a greedy algorithm

that can efficiently align sequences that are highly similar. MPBLAST and miBLAST are different versions of BLAST used for parallel queries.

The running time of all online search methods is proportional to the size of the dataset, which is increasingly prohibitive. Therefore, much work has been done to develop searching methods that rely on dataset pre-processing to create a search index. These search methods are known as offline search methods.

1.3.3 Offline searching methods

In offline search, we first preprocess a dataset by building a persistent data structure, called an index, to support fast search. This data structure is often designed to store the occurrences of patterns in the dataset to support fast search later on. The patterns might be strings or substrings of fixed or variable lengths. Most previous works fall into one of two categories: suffix-array methods and seed-and-extend heuristic methods.

The suffix-array methods convert the dataset into a suffix-array data structure, which simulates a suffix-tree of the dataset. In the case of DNA datasets, edges of this suffix-tree are labeled with one of the four nucleotides. To form a unique suffix of the dataset, the algorithms traverse the tree from the root to a leaf while concatenating all the nucleotides on the edges along the path. Every leaf node stores all matching locations of this unique suffix in the dataset. Searching for a query read is performed by traversing through the dataset suffix-tree from the root to a leaf node following the query sequence. If there exists a path from the root to a leaf such that the corresponding suffix of the path matches the query read, then all the locations stored in the leaf node are returned as matching locations. Suffix arrays simulate the suffix-tree traversal process with a much smaller memory footprint by using the Ferragina- Manzini index [27] and the Burrows-Wheeler Transform [16]. There

are two significant problems with the suffix array approach: it does not manage mismatches efficiently, and it uses lots of memory, even with these space saving optimizations.

The offline seed-and-extend method, similar to the online seed-and-extend method, is developed based on the observation that for a correct matching, a query sequence and a database sequence must share some small regions of exact matches or seeds. By identifying the seeds of a query, the search methods limit the search range from the whole dataset to only the neighborhood of each seed. In offline search, seeds are found by preprocessing the dataset and storing the locations of their occurrences in a separate data structure called an index. During the search process, a seed-and-extend method first identifies the seeds in a query. Then the search method tries to extend the matches at each of the seed locations using dynamic programming. SSAHA [66] and BLAT [36] are early examples of indexed seed-and-extend search methods. In both methods, hash tables are used to save seeds extracted from a database. In 2008, Morgulis *et al.* [56] allow creating and passing an index to MegaBLAST to perform indexed search. Instead of hash tables, the index used by Morgulis *et al.* is an array of all possible seeds associated with the lists of occurrences of each seed.

1.3.3.1 Defining k-mer indexes

Searching with k-mer indexes is one of the earliest and major types of indexed seed-and-extend search methods. In this search method, the seeds are substring of length exactly k which is known as k-mers. A k-mer index consists of two major parts: the k-mer dictionary and the inverted lists. The k-mer dictionary is composed of all/some possible k-mers that can be extracted from the dataset sequences. Each k-mer in the k-mer dictionary has an inverted list which is the list of all/some if its occurrences in the dataset.

Finding maximal exact matches $MEMs$ and highly similar local alignments $HSLAs$ is

often sped up using k-mer indexes. Similar to any seed-and-extend offline method, a k-mer index supports quickly finding shared k-mers. That is, when a query sequence is given, we extract k-mers from the query, check if that k-mer appears in the dictionary, and find the corresponding shared k-mers by using the stored list of occurrences. We typically extend these shared k-mers in two stages. In the first stage, we try to extend every shared k-mer into an exact matche of a given minimum length. If the first stage is successful, we then further extend the match into an MEM. In the context of searching for $HSLAs$, every MEM is extended even further by allowing mismatches and/or gaps. We describe this search process more precisely in chapter 3 and chapter 4.

In this dissertation, we only focus on indexed seed-and-extend searching methods that are used to find $MEMs$ and $HSLAs$. More precisely, we focus only on k-mer index search methods where seeds are k-mers.

1.4 Overview of k-mer sampling strategies

One of the biggest problems with using k-mer indexes is that the size of the index is significantly larger than the underlying database/ datasets. As biological databases/datasets rapidly increase in size, the size of the resulting k-mer indexes increases which makes using a k-mer index infeasible for many applications. Furthermore, query time increases rapidly as the database's size and/or the number of queries increases. To ensure k-mer indexes remain viable, we must mitigate k-mer index size and query time.

One of the most effective and widely used ways of mitigating k-mer index size and query time is to perform sampling, in which we omit some k-mer occurrences from the index. We classify sampling strategies in two ways: how they choose k-mers and how many k-mers they

choose.

1.4.1 Fixed vs. minimizer sampling

There are two major ways to choose k-mers: fixed sampling and minimizer sampling. In *fixed sampling*, for a given sampling step $w \geq 1$, we choose the k-mers that occur at every wth position [56, 66, 36]; this is the sampling option supported by Indexed BLAST. When a query sequence is given, all the k-mers from the query are used during the search process.

On the other hand, in *minimizer sampling*, we choose a "minimum" k-mer within a given window [75, 76, 101, 18, 59, 43]. More specifically, for a window of w adjacent k-mers, the k-mer that is alphabetically minimum is selected. The next window then starts one position to right from the previous window. When a query sequence is given, we also sample the k-mers from query sequence in the same fashion. Thus, only the sampled k-mers from the query are used during the search process.

It has widely been assumed that fixed sampling produces smaller indexes and has less index construction times than minimizer sampling, but that minimizer sampling leads to faster query processing than fixed sampling. For example, Roberts *et al.* [75] highlight the importance of sampling k-mers at query time during the search procedure saying that "the procedure would still be more efficient if we could compare only a fraction of the k-mers in T to the database" where T is their notation for a set of query strings. However, these beliefs have never been empirically verified. In fact, few studies have empirically tested either method on its own [75, 56]. In chapter 3, we fill this gap by systematically evaluating and comparing fixed sampling and minimizer sampling to assess how these methods perform with respect to index construction time, index size, and query processing time. Specifically, we compare and contrast the construction time, index size, and query processing time required

to find all $MEMs$ using both fixed and minimizer sampling.

1.4.2 Hard fixed vs. soft fixed sampling

In fixed sampling, for a given sampling step $w \geq 1$, a k-mer that occurs at every wth position is saved. We distinguish between two types of fixed sampling: *hard sampling* [56, 66] and *soft sampling* [36]. In hard sampling, we choose w small enough such that we are guaranteed to find all desired $HSLAs$. On the other hand, in soft sampling, we consider large w values, and thus we risk missing some $HSLAs$.

In chapter 4, we study how best to sample a k-mer index to manage index size, query time, and accuracy where accuracy refers to finding all desired $HSLAs$. We evaluate a broad range of sampling rates w that includes existing choices and new sampling choices. In particular, we study *soft sampling*, or sampling especially sparsely to reduce further index size and query time at the risk of missing some $HSLAs$. We show that using soft sampling, which has largely been ignored in previous studies, significantly reduces index size and computation times with very little loss in accuracy.

1.5 Research questions and major results

We study the problem of trying to find the best sampling strategy to create simultaneously efficient and accurate k-mer indexes in the context of finding $MEMs$ and $HSLAs$ using DNA datasets. In particular, we want to answer two main questions. The first question is to compare the effectiveness of the two major sampling strategies, fixed sampling and minimizer sampling. The second question is to study the effectiveness of soft or sparse fixed sampling. To address these two major questions, we systematically investigate the effect of

sampling on k-mer indexes.

Most biological applications use one of two major types of sampling: fixed sampling and minimizer sampling. It is well known that fixed sampling will produce a smaller index, typically by roughly a factor of two, whereas it is generally assumed that minimizer sampling will produce faster query times since query k-mers can also be sampled. However, no direct comparison of fixed and minimizer sampling has been performed. In chapter 3, we systematically compare fixed and minimizer sampling with respect to index size and query processing time. We use the resulting k-mer indexes for fixed sampling and minimizer sampling to find all $MEMs$.

In chapter 4, we study the effects of fixed soft sampling with NCBI indexed BLAST when searching for $HSLAs$. Our work is unique in that there is little prior work that has considered soft sampling, and the little work that has considered it has not systematically studied how sampling parameter w affects accuracy. We specifically study the effect of w on the size, accuracy, and query time of the k-mer index. We also extend previous analytical models to work with a wider range of w and k values. Then, we compare our empirical results with predictions from both the original and the extended analytical models.

To assess the effectiveness of soft sampling in a real biological application, we use soft sampling k-mer index in mapping human ESTs on the human genome. We focus on k-mer-based mappers [30, 3, 77, 2, 33, 96] that are typically fully sensitive mappers that "can detect sequences missed by other tools" [31] but may be relatively slow. We study whether soft sampling k-mer indexes might increase the speed of these mappers with relatively little loss in sensitivity. These methods work in two stages. First, they find the set of all $HSLAs$ between an EST and a genome. Then they map the EST to the genome by selecting and linking these $HSLAs$. The mappers usually differ in how to modify, evaluate, and use the resulting

$HSLAs$ to assess the final mapping process. In chapter 4, we assess precisely whether the correct mapping is retained when we use soft-sampled k-mer indexes to complete the first stage of finding $HSLAs$. We only simulate the mapping process because we want our results to be general and independent from the details of the final mapping process of a mapper.

We summarize our major contributions.

1. We systematically compare fixed sampling with minimizer sampling using real biological datasets to assess how well they find all $MEMs$ with respect to index construction time, index size, and query processing time. Our results show the surprising result that fixed sampling typically answers queries at least as fast as minimizer sampling and often is faster when both methods use the same space regardless of k.

2. We evaluate the impact of the k value on the effectiveness of fixed and minimizer sampling methods to find all $MEMs$. Previous studies usually focus on only one value of k. We show that the value of k has a significant impact on a sampling method's index size and query processing time. When the value of k decrease, fewer number of k-mer occurrences are saved resulting in smaller indexes. However, when the value of k increases, the index processes queries much faster. On average, the reduction in query times for all query sets when $k = 32$ compared to $k = 12$ is 37 and 136 times faster for fixed sampling and minimizer sampling respectively.

3. We systematically assess how well BLAST can find $HSLAs$ when using soft sampling when working with the human genome as our database. In particular, we study how accurate BLAST is in retaining all $HSLAs$. We show that BLAST's accuracy is high, even for large choices of w. Furthermore, the false positive rate, in the form of shared $MEMs$ that do not extend into $HSLA$, is significantly reduced, leading

to a corresponding significant reduction in query time. This demonstrates that soft sampling is a simple but effective method to increase index efficiency with surprisingly little loss of accuracy.

4. We design a new analytical model that we call the BLAST model by extending previously developed analytical models to work with our choices of k and w. We compare the theoretical predictions from our new BLAST model and old models with our empirical results. We show that the new BLAST model is reasonably accurate whereas other analytical models are not accurate in our context.

5. Finally, we study the effects of using soft sampling for the problem of mapping human ESTs against the human genome. We conservatively simulate the process because either existing mapping tools do not support soft sampling or do not allow us to replace the first phase of finding $HSLAs$. We show that we are able to map more than 98% of the query ESTs perfectly while reducing index size by 3-5 times and query time by 23.3% when compared to hard sampling.

Chapter 2

Related work

Over the last decade, there has been a dramatic increase in the use of k-mer indexes in biological applications and pipelines to accelerate the search for maximal exact matches ($MEMs$) and/or highly similar local alignments ($HSLAs$). Since k-mer indexes require a lot of memory, sampling has been widely used to reduce the index size. We classify sampling strategies used to create k-mer indexes in two ways: how they choose k-mers and how many k-mers they choose. The k-mers can be sampled in two ways: fixed sampling and minimizer sampling. The number of sampled k-mers depends on the size of a sampling parameter w. If w is relatively small, and thus enough k-mers are sampled, then the k-mer index reaches full accuracy, and we refer to this sampling as hard sampling. If w is large, then fewer k-mers are sampled, then the index does not guarantee full accuracy, and we refer to this sampling as soft sampling. In the current literature, no systematic study has been done to compare the different sampling methods and their relative benefits/weakness.

In this dissertation, we study the problem of trying to find the best sampling strategy to create simultaneously efficient and accurate k-mer indexes. We study k-mer indexes in the context of finding all $MEMs$ and all $HSLAs$ between a query sequence and a database of sequences, which are our two primary motivating applications.

2.1 Fixed sampling versus minimizer sampling

Two sampling methods are commonly used to build k-mer indexes: fixed sampling and minimizer sampling. No previous work has carefully compared these two sampling methods and their relative benefits/weakness.

Both Roberts *et al.* [75] and Morgulis *et al.* [56] use sampling to reduce the size of k-mer indexes with application to find $HSLAs$. In 2004, Roberts *et al.* were the first to propose minimizer sampling to reduce the size of a k-mer index (in particular the k-mer lists) to find $HSLAs$. Assuming the data is random, Roberts *et al.* computed the expected theoretical reduction in k-mer lists size to be $2/(w+1)$ compared to saving complete lists. In 2008, Morgulis *et al.* proposed indexed mega-BLAST to accelerate searching for $HSLAs$. Morgulis *et al.* used fixed sampling to build k-mer indexes and reported the reduction in list size to be $1/w$, compared to saving all complete lists. Morgulis *et al.* did not conduct any comparisons between fixed and minimizer sampling.

Finding $MEMs$ is the first step in finding local alignments. In seed-and-extend searching methods, shared seeds, which are exact matches, are extended to $MEMs$, and then to local alignments. In comparative genomics studies, $MEMs$ define anchor points for comparing different genomes, In a recent paper, $MEMs$ are also used to map long NGS reads against a genome [92]. Although suffix trees have been the traditional data structure of choice when searching for $MEMs$ [41, 1, 37, 91], Khiste and Ilie recently showed that k-mer indexes outperform suffix trees when searching for $MEMs$ in large genomes when the MEM size is relatively large [38]. Specifically, Khiste and Ilie use fixed sampling to build a memory-efficient k-mer index to search for $MEMs$. Although minimizer sampling has been actively used as an alternative way to reduce k-mer index size in other studies and allows for sampling

k-mers from a query sequence, Khiste and Ilie do not compare fixed sampling to minimizer sampling.

Roberts *et al.* [75] defined low complexity regions as one possible complication for minimizer sampling. Roberts *et al.* write "If a string contains many consecutive zeros (or As in the case of genomic data), then several consecutive k-mers may be minimizers. While this is not a major problem, it counteracts our goal of sampling a fraction of the k-mers." The low complexity regions, as defined by Roberts *et al.*, is one simple type of repeats. In general, repeats, or repeated sequences, are short substrings that occur multiple times in a genome. To handle the low complexity regions, Roberts *et al.* write "In general, we want to devise our ordering to increase the chance of rare k-mers being minimizers." They propose strategies to try to accomplish this task. However, it is unclear how effective these schemes are for dealing with repeats. In particular, it is not clear if the proposed schemes work for all types or repeats, all numbers of repeats, and all possible choices of k.

Schleimer *et al.* [80] independently introduced minimizer sampling calling it *winnowing sampling*. While winnowing is identical to minimizer, winnowing has been studied and used for for different problems and domains than minimizer. Minimizer sampling has been used to sample k-mers, called minimizers, to solve the problem of searching for local alignments using biological datasets while winnowing method has been used to solve the problem of k-mer counting and ranking using text documents datasets for document plagiarism detection. A key difference between how minimizer and winnowing sampling is that minimizer sampling requires saving sampled k-mer positions while winnowing method has not. This is different areas because in MEM and $HSLA$ search problems the positions define anchor points to compare sequences whereas in k-mer counting and ranking problems only the number of occurrences is used to estimate the similarity between two documents. Similar to Roberts

et al., Schleimer *et al.* observed that in low complexity regions (or low-entropy strings in text mining literature), a k-mer might occur more than once and all of its occurrences are sampled. Schleimer *et al.* suggested a simple fix to this problem by not sampling duplicate k-mers in both the indexing and the querying phase and call this version *robust winnowing*. If it is desired to find positional information, in robust winnowing, we should inspect all the w adjacent k-mers to find the correct matching positions. In the context of k-mer counting and plagiarism detection, Schleimer *et al.* did not explore more winnowing and robust winnowing methods. Also, Schleimer *et al.* did not compare the performance of winnowing and robust winnowing.

The k-mer sampling method has been used in sequence assembly. Ye *et al.* [101] proposed using sampled k-mers, instead of all k-mers, to reduce the memory requirements for De Bruijn graph (DBG) based assemblers. Ye *et al.* used fixed sampling to sample k-mers; the penalty is that links or edges between k-mers are longer and slightly more complex. Ye *et al.* report that fixed sampling with step w reduces their dictionary by roughly $1/w$ compared to tools that use a full list of k-mers [102, 83, 45]. Ye *et al.* note the existence of minimizer sampling and express interest in comparing minimizer sampling to fixed sampling in future work but did not compare the two in their work. In genome assembly, we only use the dictionary (the list of k-mers); we do not use the lists of k-mer occurrences. In other applications, where we use the lists of k-mer occurrences, the size of these lists is the dominant factor in index size. Therefore it is important to understand how fixed and minimizer sampling affect both the number of k-mers and the number of k-mer occurrences. Li *et al.* [48] and Movahedi *et al.* [59] both proposed disk-based DBG assemblers to avoid loading the whole graph into RAM. They load small segments of the graph incrementally and complete the assembly in this fashion. Since completing the assembly requires identifying adjacent exact

matches, both papers use minimizer sampling as a hashing mechanism, to find adjacent exact matches and group them into the same segment. We do not focus on these applications for two reasons. First, our goal is to focus on applications that use k-mer indexes in RAM, which means that index size is critical. Second, minimizer sampling, in the above context, is used as a hashing function to minimize disk I/O operations rather than reducing the list of k-mers.

The k-mer sampling method has been used to solve other problems in bioinformatics. In the k-mer counting problem, the task is to build a histogram of occurrences of every k-mer in a given dataset where k is relatively large ($k > 20$), and it is infeasible to list all k-mers in RAM. Similar to disk-based DBG assemblers, minimizer sampling is used to select m-mers ($m < k$) from every k-mer. These m-mers are later used to reduce disk I/O operations in disk-based counting k-mers tools such as MSPKmerCounter [47] and KMC2 [21]. Again, this problem is significantly different than our motivating problems which are searching for $MEMs$ and $HSLAs$. In the MEM and $HSLA$ search problems, the location of sampled k-mers is important.

In metagenomic sequence classification, Kraken program [98] uses the idea of minimizer to accelerate the classification process in large data sets. Kraken starts with creating a database that contains entries of a L-mer and the lowest common ancestor LCA of all organisms whose genomes contain that L-mer. When a query sequence is given, Kraken searches the database for each L-mer in a sequence, and then using the resulting set of LCA taxa to determine an appropriate label for the sequence. To finding all L-mers effectively, Kraken builds a k-mer index, ($k < L$) where each k-mer is associated with all L-mers containing this k-mer as a its minimizer. Since a simple lexicographical ordering of k-mers can be biased to sample more minimizers over low-complexity regains, Kraken uses the exclusive-or (XOR) operation to

scrambles the standard ordering of each k-mer's before comparing the k-mers to each other using lexicographical ordering.

2.2 Hard fixed versus soft fixed sampling

To find $HSLAs$, we use a sampled k-mer index with sampling step $w \geq 1$. We focus on $HSLAs$ that have at least one MEM with a minimum of length L. In chapter 4, we discuss in detail how to compute the right value of L to find all desired $HSLAs$.

Most previous studies of sampled k-mer indexes have focused on hard fixed sampling (or hard sampling for short) with limited study of soft fixed sampling (or soft sampling for short) and thus have not studied the effect of choosing a large sampling step w on index accuracy, query time, or false positive rate. For example, Morgulis *et al.* [56] built Indexed BLAST, which uses $w = L - k + 1$ and supports k values up to 15. Ning *et al.* [66] built the index in SSAHA with $k = 1/2(L + 1)$ and $w = k$. Morgulis *et al.* and Ning *et al.* use hard sampling to find the desired $HSLAs$. In both cases, the value of w is small enough such that for each MEM of length L, there is a sampled k-mer and thus we can find all $MEMs$ and thus all $HSLAs$.

Kent [36] has performed the main previous study of soft sampling. Kent developed an analytical model for estimating the likelihood of retaining matches and creating false positives for a variety of indexed search strategies. These include searching with one k-mer, two nearby small k-mers, and one large k-mer with one allowed error. In all cases, he built a soft sampled k-mer index where $w = k$. Kent computed the best choice of k such that the expected accuracy to find all $HSLAs$ was above a given threshold and the number of shared k-mers that did not lead to $HSLAs$ were as small as possible.

Kent's work differs from ours in several key ways. First, we consider only $k = 12$ so that we can use BLAST to perform our searches, whereas Kent considered multiple k values. Second, we consider a wide range of w values, whereas Kent only considered $w = k$. Thus, Kent's work does not allow a true study of the effect of w on index performance since, in his work, k is always changing in addition to w.dataset We extend Kent's analytical model to work with our choices of $k < L$ and w and we call this new model the BLAST model. We compare our empirical accuracy with the predicted accuracies of both Kent's original model and our BLAST model. Our results show that our BLAST model is reasonably accurate in predicting$HSLA$ retention rate. On the other hand, Kent's model significantly underestimates $HSLA$ retention rate in our experiments with the human genome. This is expected since Kent's model is not designed to handle the case when $k < L$.

2.3 Soft fixed sampling in EST mapping

We apply soft fixed sampling to the problem of EST mapping on a genome, which builds upon finding $HSLAs$. Mapping ESTs on a genome is a fundamental procedure in genome research. These mappings are used to discover the intron-exon structure of genes, SNPs, and cDNA insertions and deletions, to name just a few applications. Many different mapping tools are available, each with their own advantages [31]. We focus on k-mer–based mappers such as mrFAST/mrsFAST [30, 3], SHRiMP [77], Hobbes [2], drFAST [33], and RazerS [96]. These mappers are typically fully sensitive mappers that "can detect reads missed by other tools" [31] but may be relatively slow. In all these mappers, only hard sampling is used when building a k-mer index. We study whether soft sampling k-mer indexes might increase the speed of these mappers with relatively little loss in sensitivity when working with the human

genome as our database. Specifically, we assess whether the correct mapping is retained when we use soft-sampled k-mer indexes to complete the first stage of finding $HSLAs$. We measure the effect of sampling on both the index size and the query time. We only simulate the mapping process because we want our results to be general and independent from the details of the final mapping process of a mapper.

Finally, Xin *et al.* [100] proposed two general techniques to accelerate k-mer based mappers. The first technique is to use the set of adjacent k-mers as supporting evidence for the existence of a true match. The second is to use shared infrequent k-mers to select the best mapping location. Similar to other studies, they only used $w = k$ while evaluating these techniques. In contrast, we test a broader range of w values and demonstrate that using a larger w greatly reduces query time and index size while suffering only a small loss of sensitivity.

Chapter 3

Fixed versus minimizer sampling

Searching for similar sequences is a critical step in many bioinformatics applications and pipelines, including identifying homologues, sequence classifications, sequence mapping, and sequence assembly. This search process is often sped up by finding shared exact matches (EM) of a minimum length L. The value of L is usually set to ensure all the desired similar sequences are recognized. Depending on the application, the shared EMs can be extended to longer EMs [98]. In some applications, the EMs are extended to maximal exact matches ($MEMs$) [91, 38], and $MEMs$ are often extended further to local alignments by allowing mismatches and/or gaps [92, 56, 75, 36].

Finding EMs, $MEMs$, and local alignments is often sped up using k-mer indexes ($k < L$). One of the biggest problems with using k-mer indexes is that the size of the index is significantly larger than the underlying database/ datasets. One of the most effective and widely used ways of mitigating k-mer index size and query time is to perform sampling, in which we omit some k-mer occurrences from the index. A sampling strategy can be classified based on how it chooses k-mers. There are two major ways to choose k-mers: fixed sampling and minimizer sampling.

It has widely been assumed that fixed sampling produces smaller indexes and has smaller index construction times than minimizer sampling but that minimizer sampling leads to faster query processing than fixed sampling. However, these beliefs have never been empiri-

cally verified. In fact, few studies have empirically tested either method on its own [75, 56]. In this chapter, we fill this gap by systematically evaluating and comparing fixed sampling and minimizer sampling to assess how these methods perform with respect to index construction time, index size, and query processing time. Specifically, we compare and contrast the construction time, index size, and query processing time required to find all MEMs using both fixed and minimizer sampling.

We start by formalizing the problem of finding $MEMs$ between two sequences. Then, we illustrate how k-mer indexes are used to find $MEMs$. Next, we formally describe the two k-mers sampling methods: fixed sampling and minimize sampling. We highlight the key similarities and differences between these two sampling methods. Finally, we set the comparison framework and conclude with the comparison results.

3.1 The MEM enumeration problem

Let Σ be a finite ordered alphabet. We focus on the alphabet for nucleotide databases $\Sigma = \{A, C, G, T\}$. Let s be a string over Σ of length $|s|$. We use $s[i]$ to denote the character at position i in s, for $0 \leq i < |s|$. We use $s[0]$ to denote the first character in string s. We use the ordered pair $s(i, j)$ to denote the substring in s starting with the character at position i and ending with the character at position j for $0 \leq i < j < |s|$. We note that substring $s(i, j)$ is also denoted as $s[i..j]$ in some papers, but we only use $s(i, j)$ in this chapter.

Definition 1 ***Exact Match* (*EM*)** *For any two strings s_1 and s_2, a pair of substrings $(s_1(i_1, j_1), s_2(i_2, j_2))$ is an exact match if and only if $s_1(i_1, j_1) = s_2(i_2, j_2)$. Also, a substring ω is an exact match if there is an exact match pair $(s_1(i_1, j_1), s_2(i_2, j_2))$ such that $\omega =$*

$s_1(i_1, j_1)$. *The length of an exact match is* $j_1 - i_1 + 1$.

Definition 2 ***Maximal Exact Match*** **(*MEM*)** *An exact match* $(s_1(i_1, j_1), s_2(i_2, j_2))$ *is called maximal if* $s_1[i_1 - 1] \neq s_2[i_2 - 1]$ *and* $s_1[j_1 + 1] \neq s_2[j_2 + 1]$. *Also, a substring* ω *is a maximal exact match if there is a maximal exact match pair* $(s_1(i_1, j_1), s_2(i_2, j_2))$ *such that* $\omega = s_1(i_1, j_1)$.

We now formalize the problem of finding $MEMs$ between two datasets.

Definition 3 ***MEM Enumeration Problem*** *Given two datasets of sequences* D_1 *and* D_2 *and an integer* L, *the* MEM *enumeration problem is to find the set of all* $MEMs$ *of length at least* L *between all sequences in* D_1 *and all sequences in* $D2$. *We denote this set as* $MEM(D_1, D_2, L)$. *We use* $MEM(L)$ *if* D_1*and* D_2 *are clear from the context.*

We illustrate many of these and later definitions using the following example where $D_1 = \{s_1\}$ and $D_2 = \{s_2\}$ and s_1 and s_2 are as follows:
$s_1 = GTAC\ T\ AGG\ CTA\ CTA\ GGGG$ with length $|s_1| = 18$
$s_2 = GTAC\ A\ AGG\ CTA\ CTA\ CTA\ TTTT$ with length $|s_2| = 21$

The two string s_1 and s_2 have two $MEMs$ of length at least 6: $AGGCTACTA = (s_1(5, 13), s_2(5, 13))$ with length 9 and $CTACTA = (s_1(8, 13), s_2(11, 16))$ with length 6. Thus, $MEM(\{s_1\}, \{s_2\}, 6) = \{(s_1(5, 13), s_2(5, 13)), (s_1(8, 13), s_2(11, 16))\}$ whereas
$MEM(\{s_1\}, \{s_2\}, 8) = \{(s_1(5, 13), s_2(5, 13))\}$.

In this study, we focus on finding $MEMs$ of a minimum length L between a query sequence and a database of sequences because it is a critical step in searching for local alignments with tools such as NCBI BLAST.

3.2 Using k-mer indexes to find $MEMs$

Finding $MEMs$ with a minimum length L is often sped up using a k-mer index, $k \leq L$, at the cost of additional space. A k-mer index supports quickly finding EMs of length k which are also known as shared k-mers. We typically extend these shared k-mers in two stages. In the first stage, we try to extend every shared k-mer into an EM of length $L \geq k$. If the first stage is successful, we then further extend the match into an MEM. In the context of searching for local alignments, every MEM is extended even further by allowing mismatches and/or gaps.

It is possible to skip the first extension step and build an L-mer index to find all shared L-mers. However, due to technical limitations and the huge memory requirements necessary for building an L-mer index for $L > 32$, it is common to build the index using $k \leq 32 < L$ [38, 98, 56, 75].

We typically work with a k-mer index as follows. We save the list of k-mers present in the database and we refer to this list as dictionary. For each saved k-mer, we save some of its occurrences into a list. When given a query sequence, we extract k-mers from the query, see if that k-mer appears in the dictionary, and find the corresponding shared k-mers by using the stored list of occurrences.

We describe this search process more precisely as follows.

Definition 4 (k-mer and k-mer occurrence) *Consider any length k substring $s(j-k+1, j)$ of string s where $k-1 \leq j \leq s-1$. We call that substring a k-mer and more concisely represent this k-mer occurrence using the ordered pair (s, j).*

Definition 5 (Shared k-mers and shared k-mer occurrences) *Consider any two strings s_1 and s_2 that have an exact match $(s_1(i_1, j_1), s_2(i_2, j_2))$ of length k. We call the common*

substring $s_1(i_1, j_1)$ (equivalently $s_2(i_2, j_w)$) a shared k-mer and more concisely represent the corresponding shared k-mer occurrence using the quadruple (s_1, j_1, s_2, j_2).

Any string s of length $|s|$ contains exactly $|s| - k + 1$ k-mer occurrences. Using our previous example with $k = 3$, s_1 and s_2 have 16 and 19 3-mer occurrences, respectively. Furthermore, s_1 and s_2 have exactly seven shared 3-mers: ACT, AGG, CTA, GCT, GGC, GTA, and TAC. These shared 3-mers result in 24 different shared 3-mer occurrences as follows. GCT, GGC, and GTA appear exactly once in s_1 and s_2, and thus each of them has exactly one shared 3-mer occurrence: $(s_1, 9, s_2, 9)$, $(s_1, 8, s_2, 8)$, and $(s_1, 2, s_2, 2)$. The 3-mer AGG occurs 2 times in s_1 and 1 time in s_2, and thus AGG is part of two different shared 3-mer occurrences: $(s_1, 5, s_2, 7)$ and $(s_1, 15, s_2, 7)$. Since shared 3-mer ACT occurs 2 times in both s_1 and in s_2, the shared 3-mer ACT is part of 2×2 different shared 3-mer occurrences: $(s_1, 4, s_2, 12)$, $(s_1, 4, s_2, 15)$, $(s_1, 12, s_2, 12)$, and $(s_1, 12, s_2, 15)$. Similarly, TAC occurs 2 times in s_1 and 3 times in s_2, so shared 3-mer TAC is part of 3×2 different shared 3-mer occurrences: $(s_1, 3, s_2, 3)$, $(s_1, 3, s_2, 11)$, $(s_1, 3, s_2, 14)$, $(s_1, 11, s_2, 3)$, $(s_1, 11, s_2, 11)$, and $(s_1, 11, s_2, 14)$. Finally, CTA occurs 3 times in s_1 and 3 times in s_2, leading to 3×3 different shared 3-mer occurrences: $(s_1, 5, s_2, 10)$, $(s_1, 5, s_2, 13)$, $(s_1, 5, s_2, 16)$, $(s_1, 10, s_2, 10)$, $(s_1, 10, s_2, 13)$, $(s_1, 10, s_2, 16)$, $(s_1, 13, s_2, 10)$, $(s_1, 13, s_2, 13)$, and $(s_1, 13, s_2, 16)$.

Since $k \leq L$, it is possible that a shared k-mer occurrence is not part of an MEM of length at least L; we call such a shared k-mer occurrence a **false positive**. In general, decreasing the value of k increases the chance that a shared k-mer occurrence is a false positive.

Using the above 24 shared 3-mers occurrences and assuming $L = 6$, the $MEM =$

$AGGCTACTA$ can be found by extending any of the following seven 3-mer occurrences: $(s_1, 7, s_2, 7)$, $(s_1, 8, s_2, 8)$, $(s_1, 9, s_2, 9)$, $(s_1, 10, s_2, 10)$, $(s_1, 11, s_2, 11)$, $(s_1, 12, s_2, 12)$, or $(s_1, 13, s_2, 13)$. Similarly the $MEM = CTACTA$ can be found by extending any of the following four 3-mer occurrences: $(s_1, 10, s_2, 13)$, $(s_1, 11, s_2, 14)$, $(s_1, 12, s_2, 15)$, or $(s_1, 13, s_2, 16)$. The remaining twelve shared 3-mer occurrences are false positives. If $L = 8$, then the seven shared 3-mer occurrences that can be extended to $AGGCTACTA$ are not false positives. The remaining sixteen shared 3-mer occurrences are false positives.

Every EM of length L has $L - k + 1$ shared k-mer occurrences. Finding and extending one of these shared k-mer occurrences is sufficient for finding that EM. Therefore, when building k-mer indexes, we can store a sampled subset of k-mer occurrences in the index and still find every possible MEM of length at least L. With sampling, we not only reduce the index's memory requirements, we also reduce query time by not discovering the same MEM multiple times. Sampling, therefore, is a very effective method for improving a k-mer index's efficiency (reducing construction time, space, and query time).

3.3 Fixed sampling versus minimizer sampling

In bioinformatics, two sampling methods are commonly used to build k-mer indexes: **fixed sampling** [38, 56] and **minimizer sampling** [75, 98]. To ensure that a k-mer index achieves 100% sensitivity which means that it finds all shared $MEMs$ of length at least L, both methods ensure that within every MEM of length at least L, at least one k-mer occurrence is saved to the index. We now define both sampling methods comparing and contrasting their relative strengths and weaknesses.

Fixed sampling is simple greedy sampling strategy that minimizes the number of k-mer occurrences stored in the index. The goal is to ensure we choose one complete k-mer from every possible substring of length L from each database sequence s. For example, we must choose one k-mer from $s(0, L-1)$ to store in the index; we greedily choose the k-mer that ends at $s[L-1]$ since it not only covers this substring but also the next $L-k-1$ substrings up to but not including $s(L-k+1, 2L-k)$. To cover that substring $s(L-k+1, 2L-k)$, we again greedily choose the k-mer that ends at $s[2L-k]$ since it again covers the next $L-k-1$ substrings. In general, the jth k-mer occurrence that we sample ends at position $L-1+(j-1)w$ where $w = L-k+1$. We typically refer to $w = L-k+1$ as our sampling step or sampling window for fixed sampling. During the query phase, we extract every k-mer from the query sequence q to search for shared k-mer occurrences. Since every k-mer is extracted from q, if s and q have an MEM of length at least L, then some shared k-mer from that MEM will be in the k-mer index and the MEM can be recovered. Fixed sampling has several advantages. First, it is very fast to construct the index. Second, it stores the minimum possible number of k-mer occurrences in the index to guarantee 100% sensitivity and thus minimizes index size. The disadvantage is that all k-mers from the query sequence need to be processed which may slow query time.

Minimizer sampling uses a more sophisticated sampling strategy that allows sampling of both the database sequence s and the query sequence q. We must again choose one k-mer from $s(0, L-1)$ to store in the index. This time, we choose to store the **minimum** k-mer from $s(0, L-1)$ in our index where we order substrings in some canonical order. For simplicity, one can use the alphabetical order where $A < C < T < G$ which implies $AAA < AGA < AGG < TAA$. To improve performance and increase the probability that rare k-mers become minimizers, different orderings are often used. For example, Roberts *et*

al. proposed using $C < A < T < G$ in odd numbered bases and the reverse ordering in even-numbered bases [75]. Alternatively, Wood *et al.* [98] suggest using the exclusive-or (XOR) operation to scramble the standard ordering of each k-mer before comparing the k-mers to each other using lexicographical ordering. Once an ordering is established, we process each length L substring of s (equivalently each window of $w = L - k + 1$ k-mers) in turn storing the minimum k-mer occurrence in the index. The first view focusing on substrings of length L seems more intuitive; the second view focusing on windows of w k-mers is useful when predicting the expected size reduction from using minimizer sampling. We note a few things. First, there may be multiple occurrences of a minimum k-mer within a length L substring; in this case for the original minimizer algorithm but not the one we focus on in this chapter, each occurrence is stored in the index. Second, minimizer sampling is likely to store more occurrences than fixed sampling as it does not maximize the distance between k-mer occurrences stored in the index. Third, the time to construct the index is greater as more substrings need to be considered. The advantage that minimizer has comes at query time. Rather than choosing all k-mers from query q, it does the same sampling. That is, we consider every substring of length L and choose only the minimum k-mers from within each length L substring to consider for extension. Since both the query and each database string extract the minimum k-mer(s) from every substring of length L, if there is an EM of length L, the same minimum k-mer will be extracted and then extended into the EM and then MEM. In summary, minimizer sampling requires more time to construct its index and builds a larger index than fixed sampling, but it processes fewer k-mers from the query sequence and thus may have faster query processing times.

We illustrate the two algorithms using our previous example where we use $k = 3$ and $L = 8$ so $w = L - k + 1 = 6$, and we use $D_1 = \{s_1\}$ as our database and $D_2 = \{s_2\}$ as our

query dataset. The goal is to return $MEM(\{s_1\}, \{s_2\}, 8)$. With fixed sampling, we store the 3-mer AGG with its occurrence $(s_1, 7)$ and the 3-mer CTA and its occurrence $(s_1, 13)$ in the 3-mer index.

All 19 3-mers from s_2 are extracted with both AGG and CTA being shared 3-mers. Since CTA occurs three times in s_2, we consider four shared 3-mer occurrences for extension: $(s_1, 7, s_2, 7), (s_1, 13, s_2, 10), (s_1, 13, s_2, 13), (s_1, 13, s_2, 16)$. The first and third can be extended to the same MEM of length 9 whereas the other two cannot be extended to a length 8 MEM and thus are false positives.

With minimizer sampling, we store three 3-mer occurrences to the index, two with ACT and one with AGG: $(s_1, 4)$, $(s_1, 7)$, and $(s_1, 12)$. Minimizer sampling is also applied to the query s_2 and three 3-mer occurrences are chosen to test for extension, two with ACT and one with AAG: $(s_2, 7)$, $(s_2, 12)$, and $(s_2, 15)$. The only shared 3-mer is ACT, and since ACT appears twice in both sequences, we consider four shared 3-mer occurrences for extension: $(s_1, 4, s_2, 12)$, $(s_1, 4, s_2, 15)$, $(s_1, 12, s_2, 12)$, and $(s_1, 12, s_2, 15)$. Only $(s_1, 12, s_2, 12)$ can be extended to an MEM of length at least 8; the other three shared k-mer occurrences are false positives.

Schleimer *et al.* [80] independently introduced minimizer sampling as winnowing sampling. While winnowing is identical to minimizer, winnowing has been studied and used for different problems and domains than minimizer. As in [75], Schleimer observed that in low complexity regions, a k-mer might occur more than once and all of its occurrences are sampled. Schleimer *et al.* suggested a simple fix to this problem which they called robust winnowing where they do not sample duplicate k-mers. The idea is that for each window, we select the minimizer. If there is more than one occurrence of the minimizer, we select the same occurrences as the previous window. If not, we select the rightmost minimizer

occurrence. This results in a possible loss of the correct minimizer occurrence match. They suggested looking at all w adjacent minimizers to find the correct match.

As we described, there are two possible optimizations that can be used to improve the performance of minimizer sampling. The first one is to avoid using lexicographical ordering. The second one is not to sample duplicate minimizers. In this chapter, we study the effects of each optimization individually and combined. To avoid lexicographical ordering, we use the randomization method suggested by Wood [98]. To prevent sampling duplicate minimizers, we use the robust winnowing method proposed by Schleimer [80], but only apply robust winnowing to the index, not to the query sequences. We sample all minimizers from a query sequence window. In this scenario, the correct minimizer occurrence matches are guaranteed to be found. We finally apply both methods together to test the effectiveness of using both optimizations simultaneously.

We represent nucleotides using two bits and store k-mers for $k \leq 32$ in a 64-bit block. All indexes are saved as hash tables where a key is a k-mer and its value is a pointer to that k-mer's list of occurrences. We store each k-mer's list of occurrences in a set data structure; each occurrence is an ordered pair of 64-bit positive integers (s, j) where s is a sequence ID and j is the ending position of this k-mer occurrence in s.

3.3.1 Sampling methods

We compare fixed sampling and minimizer sampling. Minimizer sampling can be improved using two different optimizations: randomized ordering and duplicate minimizer removal. We list all possible combinations of our two optimizations for minimizer sampling in the following table.

Using both optimizations will result in the most effective minimizer method $min_{rand,one}$.

Table 3.1: **Possible minimizer sampling versions based on the optimization techniques.**

Duplicate handling \Ordering schema	Lexicographical	Randomized
Sample all duplicate minimizers	$min_{lex,many}$	$min_{rand,many}$
Remove duplicate minimizers	$min_{lex,one}$	$min_{rand,one}$

Thus, we compare $min_{rand,one}$ with fixed sampling (fix) to test the effectiveness of the two major sampling methods. We compare $min_{lex,one}$ with min $min_{rand,one}$ to determine how much effect the randomization optimization has, and we compare $min_{rand,many}$ with min $min_{rand,one}$ to determine how effective duplicate removal is. We will not consider $min_{lex,many}$ in any comparison since it is the worst version of minimizer and known to be inefficient. Next, we formally describe each sampling method. Note that we choose parameters that ensure we achieve 100% sensitivity which means we will find all MEMs.

In fixed sampling (fix), as we described earlier, we build the k-mer index for a database of sequences by sampling from every database sequence s the k-mer occurrences ending at positions $L-1+(j-1)w$ where $w = L-k+1$ and $0 \leq j \leq \lfloor(|s|-L+1)/w\rfloor$. We refer to $w = L-k+1$ as our sampling step. During the query phase, we extract all k-mer occurrences from each query sequence q and consider them for extension. Since every k-mer is extracted from q, if a database sequence s and q have an MEM of length at least L, then some shared k-mer from that MEM will be in the k-mer index and the MEM can be recovered.

In standard minimizer sampling without any optimization ($min_{lex,many}$), for every substring of length L in a database sequence s, we store the minimum k-mer occurrence in our index; if there is more than one minimum k-mer within any length L substring, all minimum k-mer occurrences are stored. We use the normal lexicographical ordering where $A < C < T < G$ to define minimum k-mers in our work. Unlike fixed sampling, during the

query phase, we use the same sampling for a query sequence q. That is, we consider only the minimum k-mers from each substring of length L in q for extension. Since both the query and each database string extract the minimum k-mer(s) from every substring of length L, if there is an EM of length L, the same minimum k-mer will be extracted and then extended into the EM and then MEM.

We find minimum k-mers from s and q using a sliding window approach where we use the minimum k-mer from the previous window to speed up the search for the minimum k-mer(s) from the new window. For the first substring $s(0, L-1)$ or $q(0, L-1)$, we must examine all $L-k+1$ k-mers and choose the minimum one(s). We store the rightmost minimum k-mer and its position as our current minimum k-mer. If the current minimum k-mer belongs to the next window, then we can find the minimum k-mer for the next window by simply comparing the new k-mer in that window with the current minimum k-mer. If the new k-mer is no larger than the current minimum k-mer, we update the current minimum k-mer to be the new k-mer. Otherwise, the new current minimum k-mer is the same as the old minimum k-mer. If the current minimum k-mer is not part of the next window, then we must again examine all $L-k+1$ k-mers and choose the minimum one(s).

Now, we describe how to apply optimizations to reduce the number of k-mers sampled from the database which leads to a significant speedup of query time. The first optimization is to use randomized ordering instead of lexicographical ordering. To do this, we first create a random k-mer mask by uniformly selecting k letters from A, C, G, or T in each positions. We then view a k-mer as a 2k bit string. For any k-mer or equivalently 2k bit string, we create a new scrambled 2k bit string by doing an exclusive-or (XOR) operation between every bit of the 2k bit string and the 2k bit mask. We then sort all the scrambled 2k bit strings to identify a minimizer. For example, the bit string for the 4-mer $AACC$ is 00000101

using lexicographical ordering, where $A = 00$, $C = 01$, $G = 10$, and $T = 11$. Let the random 4-mer mask be $CGAT$ which is equivalent to the 8-bit string 01100011. After applying the XOR operation between $AACC$ (00000101) and $CGAT$ (01100011), the resulting scrambled new bit string for $AACC$ is 01100110. We refer to minimizer sampling that uses randomized ordering as min_{rand}.

The second optimization prevents sampling duplicate minimizers in the indexing phase. There are two occasions where standard minimizer stores duplicate minimizers in the index. The first occurs when we the current minimizer is not part of the next window and we must examine all $w = L - k + 1$ k-mers in that window. If we find multiple minimizers, all are stored in the index using the standard minimizer sampling strategy. To apply the duplicate removal optimization, we store only the rightmost minimizer in the index. The second possibility for storing duplicate minimizers occurs when the current minimizer for the previous window still lies within the next window and is identical to the one new k-mer for that window. In this scenario, to remove duplicates, we do not store this duplicate copy at this time in the index. However, we do track its position so that if no new minimizer is found before the current minimizer moves out of the current window, we can use this k-mer to replace the current minimizer at that time and still do only one comparison for that window. At that time, we would have to store this minimizer in the index if it is the minimizer of that window. We refer to minimizer sampling that uses duplicate removal as min_{one}.

3.3.2 Indexing and querying

In the indexing phase, we create a k-mer index for a given database and sampling method as follows. We sample k-mers and their occurrences from each sequence based on the selected

sampling method (fix, $min_{rand,one}$, $min_{rand,many}$, and $min_{lex,one}$). We save the sampled k-mers into the index dictionary, and for each k-mer occurrence, we update the corresponding list of k-mer occurrences.

We then proceed to the query phase where we sequentially process each query sequence. If we use fixed sampling (fix), we extract all k-mer occurrences from query sequence q. For all minimizer methods $min_{rand,one}$, $min_{rand,many}$, and $min_{lex,one}$, we extract the minimum k-mer occurrences including duplicates from each window in q.

Once we extract the k-mer occurrences from q, we use the index to find shared k-mer occurrences and then $MEMs$ as follows. For every k-mer occurrence in q, we check if the k-mer is in the index dictionary. If the k-mer is found, then we use the k-mer's associated list of occurrences to find all shared k-mer occurrences between q and the database of sequences.

We perform this search in a manner similar to NCBI BLAST with the goal of minimizing the number of database read operations. Specifically, we group the shared k-mer occurrences between q and DB by database sequence ID s. For the list of k-mer occurrences shared between q and s, we sort them in alphabetical order and then positional order. We store this information in a hash table with key s where the hash table entries are pointers to the sorted lists of shared k-mer occurrences. We then read in each relevant database sequence s exactly once and process all the corresponding shared k-mers in alphabetical order of k-mer.

For every query sequence q and a database sequence s, we report any shared k-mer occurrence that can be extended to length at least L as an MEM. Our extension method is similar to that of Khiste and Ilie [38]. Before we try to extend a shared k-mer occurrence, we first check if it is contained within our list of discovered $MEMs$ which is, of course, initially empty. If so, we skip this shared k-mer occurrence and move on to the next one. If not, then we try to extend the k-mer in both directions to see if it is part of an MEM with

length at least L. If the extension succeeds, we add the new MEM to our list of discovered $MEMs$. If the extension fails, we report this shared k-mer occurrence as a false positive. This ensures we only extend one shared k-mer occurrence within any MEM.

We check if a shared k-mer occurrence is part of a discovered MEM using the following properties. A shared k-mer occurrence (q, j'_q, s, j'_s) is part of a shared MEM $MEM = (q(i_q, j_q), s(i_s, j_s))$ if the following conditions hold: (1) $i_q \leq j'_q - k + 1 < j'_q \leq j_q$, (2), $i_s \leq j'_s - k + 1 < j'_s \leq j_s$, and (3) $(j'_q - k + 1) - i_q = (j'_s - k + 1) - i_s$. Checking these conditions can be done in constant time per discovered MEM, and typically the number of discovered MEMs per pair of sequences q and S is small, so this verification step typically takes constant time.

3.3.3 Experimental setting and evaluation metics

3.3.3.1 Database and query sets

We consider only nucleotide datasets. Our database is the human genome. We use three query sets: the mouse genome, the chimp genome and an NGS dataset. All the datasets are publicly available. The genome datasets can be downloaded from UCSC (http://hgdownload.cse.ucsc.edu). The NGS dataset can be downloaded from Sequence Read Archive (SRA) on the NCBI website (https://www.ncbi.nlm.nih.gov/sra) Table 4.1 describes each dataset used. According to Koning *et al.* [20], two third of the human genome consists of repetitive sequences. It is also known that the mouse genome contains many repeats too. [56, 58]. It is unclear if this is the case for the chimp and NGS datasets.

Performing the queries using each query set directly would require large amounts of computing power, memory and time. To support parallelization of queries on MSU's High

Performance Computing Cluster, we pre-process all the datasets as follows. We first divide every sequence into non-overlapping sequences of length 1000. The number of resulting sequences is show in Table 4.1. This allows us to run our experiments in parallel and to ensure that the set of shared k-mers and MEMs can be saved in RAM. We also only save letters in $\{A, C, G, T\}$; that is, ambiguous characters are removed. For each pre-processed query set, we partition the set into 1000 query sets of equal size (except the last set my be slightly smaller). We recognize that we may not be able to find $MEMs$ that extend across the pre-processed sequences, but this should not significantly change our results.

Table 3.2: **datasets used for testing.**

Datasets	Size (Mbp)	#Seq.	Type	# Processed Seq.
Homo sapiens (Human)	3137	93	Database	2,897,341
Mus musculus (Mouse)	2731	66	Query set	5,306
Pan troglodytes (Chimp)	3218	24,132	Query set	5,818
SRA:SRR003161 (NGS)	788.5	1,376,701	Query set	2,792

The database set is only one large set. The query sets are partitioned into 1000 small query sets where each small set has the indicated number of processed sequences (except the last set may have fewer sequences). The processed sequences are of length 1000 (except the last processed sequence for every sequence may be shorter).

For each of our pre-processed query sets, we compute the actual number of MEMs between that query set and the pre-processed human genome for both choices of L. These results are shown in Table 3.3.

Table 3.3: **The number of MEMs for each query set and the human genome for both choices of L given our pre-processing into sequences of length 1000.**

L	Mouse	Chimp	NGS
50	838,857,328	2,077,183,744	940,731
100	428,609	101,868,611	457,512

3.3.3.2 Index parameters and metrics

We study the impact of the sampling methods on the k-mer index creation phase. We consider the following sampling methods fix, $min_{rand,one}$, $min_{lex,one}$, and $min_{rand,many}$. We use the index to find all $MEMs$ of length at least L where $L \in \{50, 100\}$. For each sampling method, we create set of indexes for $k \in [12, 32]$. We consider $L = 50$ and $L = 100$, because it frequently used in biological applications that compare mouse and chimp against human genome [91, 38] or map the NGS dataset against human genome [92, 36].

For each index, we report the dictionary size, lists size and the total index size which is the sum of dictionary and lists sizes. The dictionary size is measured by counting the number of k-mers. The lists size is measured by counting the number of k-mers occurrences in all lists. We also report the index construction time.

3.3.3.3 Querying parameters and metrics

The index is used to find all $MEMs$ of length at least L where $L \in \{50, 100\}$. For L and for each sampling method, we used set of k-mer indexes where $k \in \{12, 16, 20, 24, 28, 32\}$. The total number of indexes considered is $2 \times 3 \times 6 = 48$ indexes. All the indexes give the same final results, namely all $MEMs$ of length at least L.

For each query phase, we report the time and the number of "false positives". The number of false positives for a query set is the number of shared k-mer occurrences that failed to be extended to a MEM of length at least L. Recall that we partition each query set into 1000 query set partitions. The reported time is the sum of times that an index needs to answer all queries in all query set partitions. Likewise, the number of false positives for a query set is the sum of the number false positives for all queries in all query set partitions..

3.3.3.4 System specification/configuration

We run the experiments on a cluster that runs the Community Enterprise Operating System (CentOS) 6.6. The cluster has 24 nodes where each node has two 2.5Ghz 10-core Intel Xeon E5-2670v2 processors, 256 GB RAM, and 500 GB local disk.

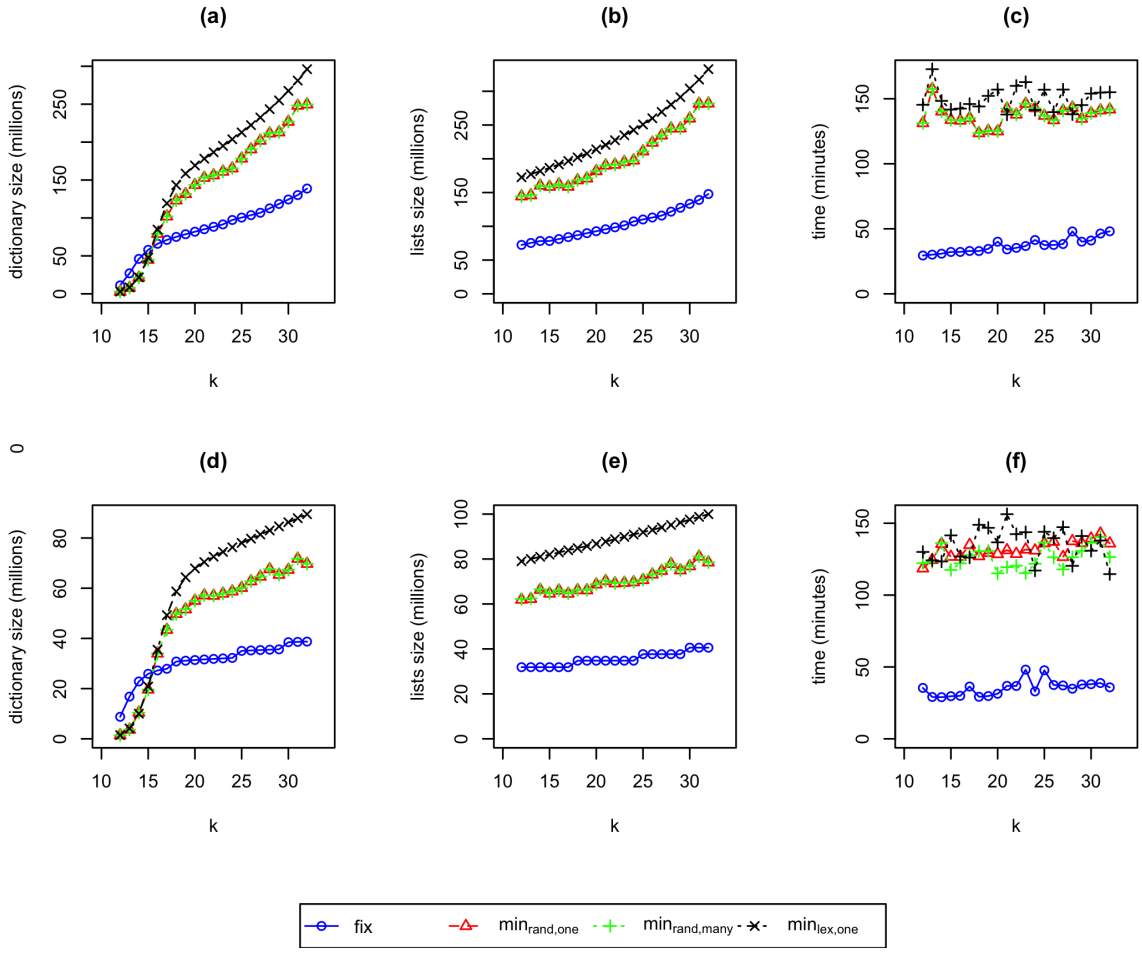
3.4 Results and discussion

3.4.1 Index size and index construction time

Fixed sampling (fix) *produces indexes that are less than half the size of those produced by all minimizer sampling methods (*$min_{rand,one}$, $min_{rand,many}$, *and* $min_{lex,one}$*) for almost all choices of* k. Likewise, we can construct fixed sampling's index roughly 3 to 4 times as fast as we can construct minimizer's index. We provide full index size and construction time results in Figure 4.2.

We now explore why fixed sampling produces indexes that are roughly half the size of indexes produced by minimizer sampling $min_{rand,one}$. We start with the size of the occurrence lists. For a fixed value of k, we can accurately predict the size of fixed sampling's k-mer occurrence lists because $1/w$ of the total number of k-mer occurrences will be sampled. For minimizer, Roberts *et al.* showed that for random sequences, the number of minimizers would be roughly $2/(w+1)$ of the total number of k-mer occurrences [75]. Basically, each minimizer would cover roughly half a window of length w rather than a full window of length w as we get from fixed sampling. Thus, we would expect fixed sampling to produce occurrence lists that are roughly $(w+1)/2w$ the size of the occurrence lists produced by minimizer sampling; that is, the occurrence lists should be just more than half the size.

Figure 3.1: **The dictionary sizes, lists sizes, and construction times for k-mer indexes built using fixed sampling (fix) and minimizer sampling $min_{rand,one}$, $min_{rand,many}$, and $min_{lex,one}$.** For parts (a), (b), and (c), we use $L = 50$. For parts (d), (e), and (f), we use $L = 100$. For all graphs, $12 \leq k \leq 32$.



Note, Roberts *et al.* observed that the actual proportion of minimizers in practice can be a few percent above $2/(w+1)$ for several reasons. In our experiments, we see that the number of sampled occurrences for fixed sampling divided by the number of sampled occurrences for minimizer sampling actually ranges from 48% to 55% for both $L = 50$ and $L = 100$. This is consistent with the observation of Roberts *et al.* that the actual proportion of minimizers can be a few percent above $2/(w+1)$.

Minimizer sampling $min_{rand,many}$ has essentially identical results to minimizer sampling $min_{rand,one}$ with respect to the size of occurrence lists; the optimization to remove duplicate minimizers from a window does not have much effect on the total number of sampled occurrences. Minimizer sampling $min_{lex,one}$ produced indexes that are larger than minimizer sampling $min_{rand,one}$ with respect to the size of occurrence lists; the optimization to use randomized ordering, instead of lexicographical ordering, effectively reduces over sampling the same k-mer in regions with many repeats resulting in 15% to 20% reduction for $L = 50$ and $L = 100$, respectively.

We now consider the dictionary size. For the smallest values of k that we consider, mainly 12-15, minimizer typically has much smaller dictionaries than fixed sampling. For $k = 12$ and $L = 100$, minimizer's dictionary is almost 6 times smaller than fixed sampling's dictionary. For these small values of k, many of the sampled k-mers are chosen many times, and this is especially true for minimizer which leads to its smaller dictionary. However, for these k values, because many of the sampled k-mers are chosen many times, the dictionaries are much smaller than the occurrence lists, so fixed sampling still has a total index size that is roughly half that of minimizer. For example, for $k = 12$ and $L = 50$, for fixed sampling, each k-mer in the dictionary appears roughly 6.5 times in the occurrence lists whereas for minimizer sampling $min_{rand,one}$, each k-mer in the dictionary appears roughly 52 times in the occurrence lists.

Once we consider $k \geq 16$, for fixed sampling, each dictionary consists of mostly unique k-mers. For example, for $k = 16$ and $L = 50$, each k-mer in fixed sampling's dictionary appears roughly 1.23 times in the occurrence lists. For minimizer sampling $min_{rand,one}$, this starts to happen around $k = 21$. For example for $k = 21$ and $L = 50$, each k-mer in minimizer sampling's dictionary appears roughly 1.24 times in the occurrence lists. By the

time $k = 32$, each dictionary k-mer appears less than 1.13 times in the occurrence lists for both fixed sampling and minimizer sampling. This implies that for large k, the dictionary size is comparable to the occurrence lists size. Specifically, we see that fixed sampling's dictionaries are roughly half the size of minimizer sampling's dictionaries for $k \geq 21$ for both $L = 50$ and $L = 100$. Finally, we note that the dictionary size for minimizer sampling $min_{rand,many}$ is identical to that of minimizer sampling $min_{rand,one}$ as $min_{rand,many}$ only omits some repeated occurrences for the same k-mer. Minimizer sampling $min_{lex,one}$ produced dictionaries that are larger than minimizer sampling $min_{rand,one}$, again because we reduce oversampling the same k-mer in regions with many repeats. For example, when $k > 16$ the reduction ranges from 12% to 23% for $L = 50$ and $L = 100$, respectively.

We note that for all sampling methods, increasing the value of k increases the size of the index. This is expected, since the sampling step $w = L - k + 1$ decreases as k increases.

Finally, fixed sampling's faster construction time is easily explained. First, the number of sampled occurrences is less than half as many as minimizer sampling. Second, no comparisons are needed; fixed sampling simply grabs every wth k-mer whereas minimizer sampling needs to consider every k-mer and do comparisons to determine if new k-mers are minimizers. However, as we show in later, the reduction has significant impact on reducing query time.

3.4.2 Query time

We first start with our query time results. Our full query time results for each of our three sampling methods for $L = 50$ and $L = 100$ are shown in Tables 3.4 and 3.5.

Our key query time result is that for the same k values and query data with many repeats, such as in the mouse genome, $min_{rand,one}$ processes queries significantly faster than fixed sampling, especially for commonly used small k values like 12 and 16 [56, 30, 3, 77, 2, 33,

Table 3.4: **Query times (in hours) for all sampling methods and all choices of k when $L = 50$.**

Query set	k	fix	$min_{rand,one}$	$min_{rand,many}$	$min_{lex,one}$
Mouse	12	447.00	284.50	437.16	1008.30
	16	106.49	36.08	142.22	204.39
	20	52.91	26.96	107.46	86.13
	24	33.31	17.36	58.38	48.54
	28	20.84	15.48	48.77	28.87
	32	13.75	12.43	32.79	18.58
Chimp	12	1493.09	1294.66	1407.43	2340.63
	16	661.67	510.99	641.86	932.27
	20	340.00	471.84	859.09	455.38
	24	180.67	184.19	221.33	205.94
	28	98.20	116.06	146.56	116.46
	32	55.25	71.34	79.92	56.58
NGS	12	237.04	197.73	205.70	486.43
	16	88.02	79.71	72.66	134.30
	20	41.75	49.40	47.99	59.96
	24	21.33	22.72	22.21	26.69
	28	11.83	14.98	15.73	14.38
	32	6.77	10.68	9.80	7.36

96, 98]. For example, when $k = 12$ and $k = 16$, $min_{rand,one}$ answer the queries 126.14% and 369.14%, on average, faster than fixed sampling for $L = 50$ and $L = 100$, respectively. For large k, $k > 16$, $min_{rand,one}$ processes the queries 58.36% and 271.64%, on average, faster than fixed sampling for $L = 50$ and $L = 100$, respectively. When there are only a few repeats in the query data, such as in the chimp genome and NGS datasets, and for small k values, $min_{rand,one}$ is 15.15% to 37.65% faster than fixed sampling, on average. On the other hand, when the value of $k > 16$, $min_{rand,one}$ is slower than fixed by 7.73% to 19.82%.

While we observe that minimizer sampling process queries faster than fixed sampling for the same choice of k, we also observe that minimizer sampling uses more space than fixed sampling for the same choice of k. We will later compare minimizer sampling with fixed sampling when they are restricted to indexes of the same size to determine which is indeed

Table 3.5: **Query times (in hours) or all sampling methods and all choices of k when $L = 100$.**

Query set	k	fix	$min_{rand,one}$	$min_{rand,many}$	$min_{lex,one}$
Mouse	12	198.46	101.05	102.06	692.17
	16	41.74	5.63	7.25	142.50
	20	21.31	3.59	4.52	40.38
	24	12.37	3.76	4.29	15.60
	28	7.28	2.17	2.48	7.49
	32	4.92	2.16	2.35	4.89
Chimp	12	732.51	524.91	505.38	1744.26
	16	276.38	203.60	203.90	560.79
	20	131.61	160.61	254.88	242.23
	24	61.68	50.29	50.03	98.51
	28	33.20	41.01	40.88	53.60
	32	17.78	30.91	31.53	21.94
NGS	12	111.03	79.95	82.78	364.51
	16	33.33	27.39	28.17	83.38
	20	15.58	13.93	14.47	27.85
	24	7.27	5.91	5.85	10.90
	28	4.12	5.01	5.34	6.08
	32	2.32	4.47	4.43	2.52

faster. When exploring this tradeoff, we find that fixed sampling is faster than minimizer sampling when both methods have equal sized indexes.

Our next query time result is that increasing k significantly decreases the query processing time of all methods. For all query sets, increasing k from 12 to 16 reduces the query time of fixed sampling and $min_{rand,one}$ by roughly 3-5 times; the one exception is $min_{rand,one}$ with the mouse genome query set for $L = 50$ and $L = 100$ where the reduction is only 7.89% and 17.96% times, respectively. For all query sets and all methods, increasing k by an additive factor of 4 above 16 roughly halves the method's query processing time with a couple of outliers in both directions.

Our final query time result is that the optimization using randomized ordering is significantly more effective than the optimization that removes duplicate minimizers; especially

for large L and small k values. For the mouse genome and for $k = 12$ and $k = 16$, the randomized ordering is, on average, 360.46% to 1508.86% faster than lexicographical ordering. For the chimp and NGS datasets and $k = 12$ and $k = 16$, the randomized ordering is, on average, 81.62% to 280.17% faster. On the other hand, duplicate minimizer removal improves minimizer sampling using the mouse genome by 14.94% to 173.93%, on average, for $k = 12$ and $k = 16$. For the chimp and NGS datasets, duplicate minimizer removal does not improve minimizer sampling; the one exception is chimp data when $L = 100$, it gives 17.16% faster query processing for $k = 12$ and $k = 16$.

3.4.2.1 The theoretical vs. empirical query time expectation

Because minimizer sampling only tests some k-mers extracted from the query sequence to see if they are shared k-mers, one might expect that minimizer sampling would process queries as much as w times faster than fixed sampling. However, the query time results show this is not the case; the speedup is typically much less than w and often less than *twice* as fast. This is explained by counting the total number of shared k-mer occurrences found by both fixed and $min_{rand,one}$. These counts are shown in Tables 3.6 and 3.7.

Recall fixed sampling will test all $q - k + 1$ k-mers from a query sequence q. On the other hand, the expected number of query k-mers that minimizer sampling will test is $2(q - k + 1)/(w + 1)$ or roughly $2/(w + 1)$ times smaller if the sequences are generated uniformly at random [75]. Each tested k-mer will generate x shared k-mer occurrences where x is the length of that k-mer's occurrence list in the index. Theoretically, if $x = c$ for fixed sampling, then we expect that $x = 2c$ for minimizer sampling. Then fixed sampling will test $c(q-k+1)$ k-mers occurrences and minimizer sampling will test $2c(q-k+1)/(w+1)$ of k-mers occurrences; since minimizer sampling produce lists large by a factor of two. However, this

Table 3.6: **The number of shared k-mer occurrences (in billions) for all sampling methods and all choices of k when $L = 50$.**

Query set	k	fix	$min_{rand,one}$	$min_{rand,many}$	$min_{lex,one}$
Mouse	12	281.90	100.58	348.92	870.72
	16	103.19	25.02	202.80	242.16
	20	55.96	20.84	161.16	101.93
	24	33.28	11.33	83.47	52.93
	28	19.55	11.68	73.18	29.27
	32	11.74	9.36	47.14	16.98
Chimp	12	714.46	445.52	515.02	1699.86
	16	310.33	176.65	223.43	601.94
	20	155.08	255.68	520.61	221.16
	24	75.52	62.58	85.37	87.17
	28	37.92	46.97	109.50	45.74
	32	22.12	27.64	45.33	23.79
NGS	12	101.81	64.62	71.33	390.05
	16	30.29	24.05	26.56	86.30
	20	12.95	17.20	20.26	28.24
	24	6.41	7.24	8.24	11.00
	28	3.44	5.11	5.95	5.86
	32	2.09	3.42	3.91	3.06

is not always the case. For example, for $q = 1000$ and $L = 100$, then we expect minimizer sampling to be 95% faster than fixed sampling when $20 \leq k \leq 32$. For chimp and NGS datasets, the empirical results show that minimizer sampling is 7.73% to 14.24% slower than fixed sampling.

To understand the query time, we need to compute the average size of x for k-mers that are in the index dictionary. We show this in Tables 3.8 and 3.9 for each sampling method; specifically, these tables show the mean and the standard deviation of occurrence list lengths in each index. For $L = 100$ and $k = 12$, the mean length of a minimizer sampling occurrence list is just over 13 times larger than the mean length of a fixed sampling occurrence list. For $k = 16$, this falls to roughly 1.55 times larger, and for larger k, this falls to just a bit larger. Even more dramatic, the standard deviation for minimizer's occurrence list lengths

Table 3.7: **The number of shared k-mer occurrences (in billions) for all sampling methods and all choices of k when $L = 100$.**

Query set	k	fix	$min_{rand,one}$	$min_{rand,many}$	$min_{lex,one}$
Mouse	12	124.51	25.91	35.00	663.35
	16	40.72	1.49	4.24	156.26
	20	20.98	0.69	2.33	44.94
	24	10.94	0.57	1.27	13.70
	28	6.04	0.10	0.36	5.09
	32	3.25	0.10	0.50	2.82
Chimp	12	355.18	157.24	160.65	1499.97
	16	129.05	61.04	61.30	418.83
	20	58.20	87.84	174.00	125.58
	24	24.31	13.01	13.05	37.12
	28	11.72	10.77	11.62	15.88
	32	6.08	8.09	8.10	6.35
NGS	12	44.95	22.98	23.41	318.71
	16	11.89	7.81	7.86	59.64
	20	4.86	4.67	4.71	14.79
	24	2.08	1.54	1.55	4.17
	28	1.06	1.36	1.37	1.82
	32	0.58	1.09	1.10	0.73

ranges from 6.58 times up to 21.5 times larger than the standard deviation of fixed samplings occurrence list lengths.

What this shows is that some k-mers in minimizer sampling have very large occurrence lists. Furthermore, the k-mers that have large occurrence lists are exactly the k-mers that are most likely to be extracted from a query sequence since the sampling method is biased to choose them. This explains why, despite testing relatively few query k-mers, minimizer sampling have much larger query times than expected theoretically.

3.4.3 Space and speed

We summarize our comparison of the the sampling methods by plotting the space and speed of the resulting index for each query set and both choices of L in Figure 4.3.

Table 3.8: **The mean and standard deviation of the length of a k-mer's list of occurrences using the human genome for all sampling methods and all choices of k when $L = 50$.**

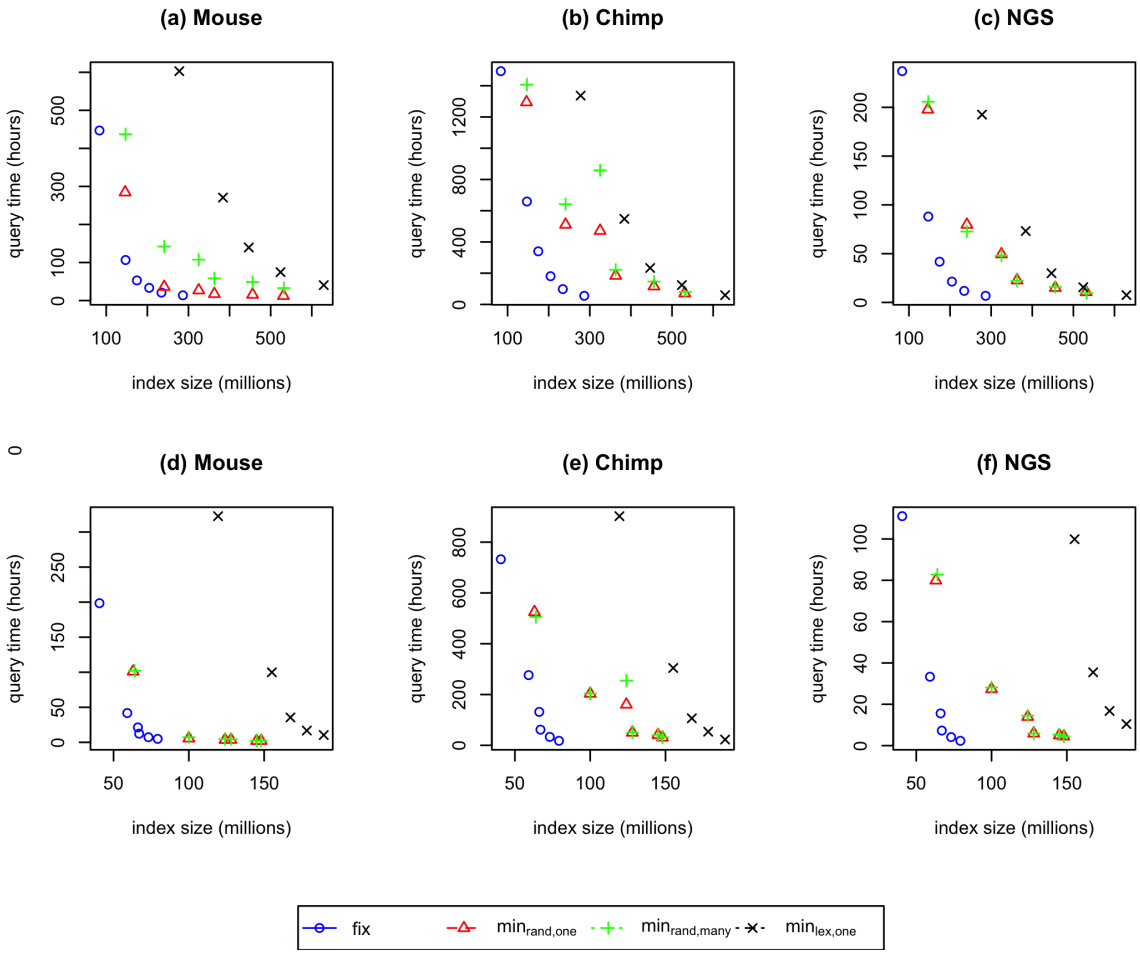
	k	fix	$min_{rand,one}$	$min_{rand,many}$	$min_{lex,one}$
Mean	12	6.50	51.72	53.20	58.70
	16	1.23	2.01	2.05	2.26
	20	1.13	1.26	1.27	1.26
	24	1.10	1.19	1.19	1.19
	28	1.08	1.16	1.16	1.15
	32	1.07	1.13	1.13	1.12
Std. Dev.	12	37.61	419.57	483.85	616.48
	16	10.17	67.30	66.02	69.38
	20	6.65	32.14	36.96	32.79
	24	4.66	20.06	22.73	19.98
	28	3.41	17.56	16.91	13.74
	32	2.65	10.61	11.84	9.09

Table 3.9: **The mean and standard deviation of the length of a k-mer's list of occurrences using the human genome for all sampling methods and all choices of k when $L = 100$.**

	k	fix	$min_{rand,one}$	$min_{rand,many}$	$min_{lex,one}$
Mean	12	3.63	45.20	44.70	53.02
	16	1.17	1.81	1.95	2.34
	20	1.11	1.24	1.25	1.28
	24	1.08	1.19	1.19	1.19
	28	1.06	1.16	1.15	1.15
	32	1.05	1.13	1.13	1.12
Std. Dev.	12	18.68	401.81	402.42	712.16
	16	6.21	67.08	61.54	80.71
	20	4.02	25.38	30.99	36.05
	24	2.62	17.51	19.63	20.47
	28	1.87	16.71	14.84	13.85
	32	1.37	9.02	10.11	8.61

If we ask both methods to use the same space regardless of k, we find that fixed sampling typically answers queries at least as fast as $min_{rand,one}$ and often is faster. For example, the index created using fixed sampling with $k = 16$ has roughly the same size as the index created

Figure 3.2: **Comparing the space and speed of fixed sampling (fix), minimizer sampling $min_{rand,one}$, $min_{rand,many}$, and $min_{lex,one}$.** We use $L = 50$ for the three upper figures and $L = 100$ for the three lower figures. The values for $min_{lex,one}$ when $k = 12$ are very large and removed from figure below.



using minimizer sampling with $k = 12$. However, fixed sampling is 37.43%, 51.11%, and 44.52% faster than minimizer sampling for the mouse, chimp, and NGS datasets, respectively. Combined with the fact that fixed sampling is much simpler than minimizer sampling, it is clear that fixed sampling is always the best choice if we optimize both time and space regardless of k.

Finally, we conclude by observing the randomized ordering optimization is more effective

than the duplicate removal optimization. We can see that for all k values we consider, the effectiveness of $min_{rand,many}$ and $min_{rand,one}$ are very similar. On the other hand, $min_{lex,one}$ is significantly worst than both $min_{rand,one}$ and $min_{rand,many}$.

Chapter 4

Soft fixed sampling

Finding highly similar local alignments ($HSLAs$) is an important step in bioinformatics and computational biology. Finding $HSLAs$ is often sped up using k-mer indexes. NCBI BLAST, and in particular indexed MegaBLAST [56], is the classic and the most widely used program to find $HSLAs$ using k-mers. Since the size of k-mer indexes is large, sampling k-mer occurrences has been used as an effective way to reduce index size and query time. Based on the results of the last chapter, and also because BLAST only supports fixed sampling, we focus on fixed sampling. We distinguish between two types of fixed sampling: *hard sampling* [56, 66] and *soft sampling* [36]. In hard sampling, we choose w small enough such that we are guaranteed to find all desired $HSLAs$. On the other hand, in soft sampling, we consider large w values, and thus we risk missing some $HSLAs$.

In this chapter, we study how best to sample a k-mer index to manage index size, query time, and accuracy where accuracy refers to finding all desired $HSLAs$. We show that using soft sampling, which has largely been ignored in previous studies, significantly reduces index size and computation times with very little loss in accuracy. We study soft sampled k-mer indexes in the context of finding $HLSAs$ between a query sequence q and a database of sequences DB. We also study that soft sampling can be effectively used in mapping ESTs to a genome for mapping tools that first find $HSLAs$.

We start by formalizing the problem of finding $HSLAs$. Then we illustrate how to

use NCBI BLAST k-mer indexes to find $HSLAs$. Next, we formally define hard and soft sampling. We then describe how to use soft sampling to improve EST mapping. Next, we illustrate the experimental settings and analytical modeling. We conclude with results that demonstrate that soft sampling is a simple but effective strategy for performing efficient searches for $HSLAs$.

4.1 Highly similar local alignments ($HSLA$)

We now formally define the first application, finding $HSLAs$. We start by defining a local alignment $A(s, q)$ between two sequences s and q. For simplicity, we denote $A(s, q)$ as just A.

Definition 6 (Local alignment) *A local alignment $A(s, q)$ between any two sequences s and q is a triple (x, y, m) where x is a contiguous subsequence of s, y is a contiguous subsequence of q, and m is an injective and monotonically increasing mapping from positions in x to positions in y.*

Figure 4.1: **Example database sequence s and query sequence q and two local alignments A_1 and A_2.** The symbol (|) identifies two mapped identical positions and ($*$) is an inserted gap position in one of the two sequences.

$$
\begin{array}{rl}
s: & CCAACGATACCCCCCTTTTTCTGCGTCC** \\
 & \quad|\,|\,|\,|\,|\;\;|\qquad\quad|\,|\,|\,|\,|\;\;|\,|\,|\,|\,| \\
q: & **AACGA*AGGGGGGTTTTTGTGCGTGGGG \\
 & \quad\underbrace{\qquad\qquad}_{A1}\qquad\quad\underbrace{\qquad\qquad\qquad\qquad}_{A2}
\end{array}
$$

Within an alignment $A = (x, y, m)$, some positions in x may map to no positions in y and vice versa. Let $map(A)$ denote the number of positions in x that map to positions in

y, and let $match(A)$ denote the number of mapped positions that are identical. We then define the length of A to be $|A| = |x| + |y| - map(A)$, and we define the match percentage to be $mp(A) = match(A)/|A|$. Finally, we define $E(A) = |A| - match(A)$ to be the number of errors in alignment A.

To illustrate these definitions, consider the example in Fig 4.1 with two local alignments $A1$ and $A2$. We have $map(A1) = 6$, $match(A1) = 6$, $|A1| = 7$, $mp(A1) = 85.7\%$, and $E(A) = 1$ whereas $map(A2) = 11$, $match(A2) = 10$, $|A2| = 11$, $mp(A2) = 91\%$, and $E(A) = 1$.

When searching for local alignments, our goal is to find all $HSLAs$ that have a minimum length and match percentage. We formally define our targeted $HSLAs$, which we also refer to as true matches, as follows:

Definition 7 (True match or $HSLA$) *For a database of sequences DB, a query sequence q, an alignment length threshold l, and a match threshold t, we define $HSLA(DB, q, l, t) = \{A(s, q) \mid s \in DB, |A(s, q)| \geq l$ and $mp(A(s, q)) \geq t\}$.*

We use $HSLA(q, t)$ when DB and l are clear from context. We also define short $HSLAs$ to represent $HSLAs$ that are barely in $HSLA(DB, q, l, t)$ and which are the hardest to find.

Definition 8 (Short $HSLA$) *For a database of sequences DB, a query sequence q, an alignment length threshold l, and a match threshold t, we define $HSLA_{\text{short}}(DB, q, l, t) = \{A(s, q) \mid s \in DB, l \leq |A(s, q)| \leq (2 - t)l$ and $mp(A(s, q)) \geq t\}$.*

For example, $HSLA(\{s\}, q, 6, 85\%) = \{A1, A2\}$ whereas $HSLA(\{s\}, q, 11, 85\%) = \{A2\}$; $A1$ is omitted because it does not meet the length threshold of 11. Likewise, $HSLA(\{s\}, q, 6, 90\%) = \{A2\}$; $A1$ is dropped because it does not meet the match percentage threshold. Focusing on short $HSLAs$, $HSLA_{\text{short}}(\{s\}, q, 6, 85\%) = \{A1\}$. $A2$ is dropped because it is too long. Note

that $HSLA(\{s\}, q, 6, 90\%)$ actually includes several alignments that overlap significantly with $A1$; we follow standard practice and only include the longest alignment with highest match percentage from any group of highly overlapping alignments in $HSLA(s, q, l, t)$.

4.2 Using NCBI BLAST k-mer indexes to finding $HSLAs$

We now describe how indexed BLAST [56] is typically used to find $HSLAs$ in $HSLA(DB, q, l, t)$. Specifically, indexed BLAST uses a seed-and-extend search process where we have one *seed* phase and two *extension* phases. In the seed phase, for a given k value k', indexed BLAST uses a k'-mer index to find shared k'-mers, where a shared k'-mer is a substring formed by k' consecutive letters that appear in both a database sequence $s \in DB$ and in the query q. More specifically, indexed BLAST identifies the locations or occurrences of these shared k'-mers. Once shared k'-mer occurrences are found, BLAST performs the first extension phase. In this phase, each occurrence is extended in both directions to find a maximal exact match (MEM), which is an exact match that cannot be extended in either direction without introducing mismatches. If a found MEM has length at least some threshold k^* (defined below), BLAST performs the second extension phase where it tries to extend the MEM into an $HSLA$. BLAST's extension process in this second phase is slightly more complex than the process from its first phase since BLAST must allow some mismatches and gaps in this second phase.

To illustrate this process, consider our example from Fig 4.1 and suppose we use BLAST to search for $HSLA(\{s\}, q, 11, 90\%)$ with $k' = 4$ and $k^* = 5$. Suppose the seed phase returns the shared 4-mers AACG, TTTT, and TGCG. When BLAST performs the first extension phase, it would find the MEMs AACGA, TTTTT, and TGCGT. Since $k^* = 5$, BLAST

would then try to extend the three MEMs to $HSLAs$. The latter two would extend to $A2$ whereas the MEM AACGA cannot be extended into an $HSLA$.

We now describe BLAST's first two phases in more detail starting with the seed phase. BLAST constructs a k'-mer index as follows. The k'-mer index saves a list of database k'-mers in a lookup table of all possible k'-mers, which is $4^{k'}$ entries. We refer to this lookup table as a dictionary. For each k'-mer in the dictionary, BLAST saves some of its occurrences in an inverted list (also known as an offset list). A k'-mer occurrence is an ordered pair (s, i) where s is the string containing this occurrence and i is the position of the last character in this occurrence.

BLAST then finds shared k'-mers as follows. BLAST extracts all k'-mers from query sequence q. BLAST then searches for each extracted k'-mer in the dictionary. If the extracted k'-mer is in the dictionary, it represents a shared k'-mer for q and some $s \in DB$. BLAST uses that k'-mer's inverted list to find occurrences of that k'-mer in DB.

A key choice is what value of k' should be used. Typically, k' is chosen to be at most 16 so that the list of all possible k'-mers (which has $4^{k'}$ entries) can be stored as an array in RAM. Since we use BLAST to perform our experiments, we use BLAST's default value of $k' = 12$.

We next describe the first extension phase where BLAST searches for MEM_{k^*}s which are MEMs of length at least k^*. The extension itself is straightforward since mismatches and gaps are not allowed. The key issue for this phase is what k^* should be. We want k^* to be as large as possible to reduce the number of false positives, which are MEM_{k^*}s that cannot be extended into $HSLAs$. It is well known how to compute k^* given a target alignment length L and a maximum number of errors E [94, 95, 19]. Specifically, $k^* = \lfloor L/(1+E) \rfloor$. The basic idea is that the worst case is when the errors are evenly spaced. The question then is

what value of L and E should be used. The hardest $HSLAs$ to find are the short $HSLAs$ defined in Definition 8; basically those of length exactly l and match percentage t. Thus, we use $L = l$ and $E = \lfloor (1-t)l \rfloor$, which leads to $k^* = \lfloor l/(1 + \lfloor (1-t)l \rfloor) \rfloor$.

4.3 Hard versus soft Sampling

The fundamental issue with using k'-mer indexes to search for $HSLAs$ is that the k'-mer index can be very large. Most systems including BLAST control dictionary size by limiting k' to a small value such as 12. With this choice of k', the problem is that there are too many k'-mer occurrences because the total number of k'-mer occurrences is roughly the total length of all the sequences in the database DB. The human genome is roughly 3 billion base pairs, so this would mean roughly 3 billion k'-mer occurrences.

For this reason, k'-mer indexes are typically sampled where we only save some k'-mer occurrences rather than all of them. We focus on *fixed sampling* where for a given sampling step $w \geq 1$, a k-mer that occurs at every wth position is saved. We distinguish between two types of fixed sampling: *hard sampling* [56, 66] and *soft sampling* [36].

In **hard sampling**, we choose $w \leq k^* - k' + 1$ so that we are guaranteed to find a k'-mer within every MEM_{k^*}. Thus, when we apply the first extension step, we will find the resulting MEM_{k^*}. Since we find all MEM_{k^*}s after the first extension step, we are guaranteed to find all $HSLAs$ after the second extension step. Without loss of generality, for hard sampling, we assume $w = k^* - k' + 1$ since this maximizes the space savings with no loss in accuracy. We refer to this w value as w_0 ($w_0 = k^* - k' + 1$).

In **soft sampling**, we consider $w > w_0$. Because we no longer are guaranteed to choose a k'-mer from every MEM_{k^*}, when we apply the first extension phase, we may miss some

MEM_{k^*}s which may lead to missing some $HSLAs$ in the next extension phase. Thus, if we use soft sampling, we risk missing some $HSLAs$.

4.4 Retention rates and false positives

Recall our goal is to find $HSLA(DB, q, l, t)$. We denote the $HSLAs$ and short $HSLAs$ found by using indexed BLAST with parameter values k' and w to be $HSLA(DB, q, l, t, k', w)$ and $HSLA_{\text{short}}(DB, q, l, t, k', w)$, respectively. With hard sampling ($w = w_0$), we know $HSLA(DB, q, l, t, k', w_0) = HSLA(DB, q, l, t)$. With soft sampling ($w > w_0$), $HSLA(DB, q, l, t) - HSLA(DB, q, l, t, k', w) \neq \emptyset$ is possible. We define the retention rate of $HSLAs$ as a function of w as follows.

Definition 9 (Retention Rate) *For a k'-mer index with a sampling step w (SI(w)), the retention rate for HSLA ($RR(w, w_0)$) and the retention rate for $HSLA_{\text{short}}$ $RR_{short}(w, w_0)$ are:*

$RR(w, w_0) = |\, HSLA(DB, q, l, t, k', w)|/|\, HSLA(DB, q, l, t)|$,

$RR_{short}(w, w_0) = |\, HSLA_{\text{short}}(DB, q, l, t, k', w)|/|\, HSLA_{\text{short}}(DB, q, l, t)|$.

We typically express these ratios as percentages. We will study how $RR(w)$ and $RR_{short}(w)$ change as a function of w. Because short $HSLAs$ are the hardest true matches to find, we expect $RR(w) > RR_{short}(w)$ in most cases.

We present a new analytical model to compute the expected retention rate of $HSLAs$ in $HSLA_{\text{short}}(DB, q, l, t)$. The new model is an extension to Kent's analytical model [36] where he essentially assumed $w = k' = k^*$ in his model. On the other hand, we propose a new model where we assume $k' < k^*$ and $w \geq 1$. We refer to the new model as the BLAST model since it accounts for typical parameters used in BLAST searches.

Searching with sampled k'-mer index produces two intermediate results: shared k'-mers and MEM_{k}*s. The second extension process, extending MEM_{k}*s into $HSLAs$, is more complex and costly than the first extension process since we are allowing some mismatches and gaps. We thus define MEM_{k}*s that do not extend into $HSLAs$ to be **false positives**. We will also study how the number of false positives changes as a function of w.

4.5 Using NCBI BLAST k-mer indexes in EST mapping

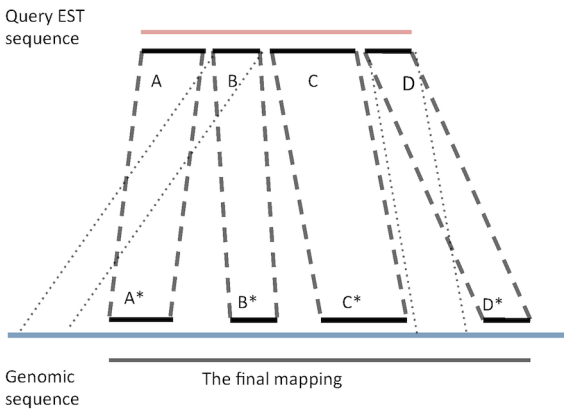
Our second motivating application, which builds upon the first, is mapping ESTs on a genome, a fundamental procedure in genome research. Many different mapping tools are available, each with their own advantages [31]. We focus on hash table–based, seed-and-extend mappers such as mrFAST/mrsFAST [30, 3], SHRiMP [77], Hobbes [2], drFAST [33], and RazerS [96]. These mappers are typically fully sensitive mappers that "can detect reads missed by other tools" [31] but may be relatively slow.

We study whether soft sampling k-mer indexes might increase the speed of these mappers with relatively little loss in sensitivity when working with the human genome as our database. These methods work in two stages. First, they find the set of all $HSLAs$ between an EST and a genome. Then they map the EST to the genome by selecting and linking these $HSLAs$. The mappers usually differ in how to modify, evaluate, and use the resulting $HSLAs$ to assess the final mapping process. Fig 4.2 illustrates the mapping procedure.

In this chapter, we assess the effectiveness of soft sampling in mapping human ESTs on a human genome. Specifically, we assess whether the correct mapping is retained when we use soft-sampled k'-mer indexes to complete the first stage of finding $HSLAs$. We measure

the effect of sampling on both the index size and the query time. We only simulate the mapping process because we want our results to be general and independent from the details of the final mapping process of a mapper. We hope our findings encourage more developers to allow the use of a wider range of k' and w values in their mappers.

Figure 4.2: **Illustration of EST mapping process**. The *HSLA*s $(A, A*)$, $(B, B*)$, $(C, C*)$, and $(D, D*)$ are used to report the final mapping.



4.6 Materials and method

We evaluate the effect of soft sampling on using BLAST to (i) find *HSLAs* and (ii) map ESTs to the human genome. For both applications, we describe our database and how we create sampled indexes. We then describe our query sets and how we perform queries. We next describe our evaluation metrics. Finally, we describe how we extend Kent's analytical model to work with our choices of $k' = 12$ and w.

4.6.1 Experimental settings

Database: For both applications of finding *HSLAs* and EST mapping, we use the human genome database provided by Morgulis *et al.* from their MegaBLAST paper [56] as

our database. Morgulis *et al.* note that the human genome database was the most frequently searched database in NCBI in 2007 with 10,000 submitted queries per weekday. They partitioned the human genome database into volumes, each of which is roughly 1 GB in size and available at (`ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast/fasta/human`). We summarize key characteristics of each volume in Table 4.1. We did experiments with both masked and unmasked data but report results only for the unmasked data since the results were similar. As in [56], we treat each volume as a separate database. That is, we create an index for each volume separately and search each volume's index separately. To obtain results for the human database, we then simply union the results found for each volume.

Table 4.1: **Human genome volume characteristics.**

Name	Size(Mbytes)	Size(bp)
Chr. 1-5, unmasked	1039.86	1,025,201,451
Chr. 6-13, unmasked	1093.27	1,077,856,590
Chr. 14-Y, unmasked	778.75	767,769,314
Chr. 1-8, masked	1517.93	1,493,033,824
Chr. 9-Y, masked	1400.78	1,377,793,531

Sampled index construction: For finding $HSLAs$, we use four different minimum alignment lengths l: 50, 100, 200, and 400 and a match threshold $t = 96\%$ or $t = 97\%$. For each of our four choices of l, we use $k^* = l/(1 + \lfloor(1-t)l\rfloor)$. For mapping ESTs, similar to Kent's design of BLAT [36], we use the same choices except we omit $l = 400$. Specifically, Kent used $l = 100$; we also include $l = 50$ and $l = 200$ to study EST mapping under a wider set of possible choices. We use a geometric progression with base $\sqrt{2}$ to choose w values for soft sampling indexes. Specifically, we consider $w = \sqrt{2}^i$. For each l, we ignore w less than $w_0 = k^* - k' + 1$ since w_0 is the largest hard sampling value. Likewise, we ignore $w \geq l$ as

these can completely skip over a potential alignment of length l. This results in a total of eleven choices ranging from $w = 8$ to $w = 256$. Combined with four choices of w_0 for hard sampling and three volumes, we create a total of $15 \times 3 = 45$ sampled indexes. We use $SI(w)$ to denote a sampled index created with sampling parameter w; note $SI(w_0)$ denotes a hard sampling index. These choices are summarized in Table 4.2. Note some sampled indexes are used with multiple l values. For example, the sampled indexes $SI(22)$ and $SI(32)$ are used for each choice of l.

Table 4.2: **A summary of the parameters used in our experiments for (1) finding $HSLAs$ and (2) EST mappings. For $HSLA$, we consider all four choices of l. For EST, we only consider the first three choices of l.**

Sampling parameters					
l	t	k^*	k'	w_0	$w > w_0$
50	96%	16	12	5	8, 11, 16, 22, 32
100	97%	25	12	14	16, 22, 32, 45, 64
200	97%	28	12	17	22, 32, 45, 64, 90, 128
400	97%	30	12	19	22, 32, 45, 64, 90, 128, 181, 256

The k'-mer indexes are built using BLAST with sampling steps w_0 and w. True matches $HSLAs$ are of length $\geq l$ and a match percentage $\geq t$. Only $HSLAs$ that have shared k^*-mers are reported by BLAST.

We build our sampled indexes using the BLAST program `makembindex` for the three volumes of the unmasked human genome database using BLAST's default value of $k' = 12$.

Query Sets and Mappable Queries: For HSLA, we use the same query sets that Morgulis *et al.* used to evaluate Indexed BLAST [56]. Morgulis *et al.* organized the queries into three sets based on the average query length: *qsmall* (average length 500), *qmedium* (average length 10,000), and *qlarge* (average length 100,000). Each set has 100 queries for 300 total queries. We group all the queries into a single set of 300 queries and report all results using

this single query set. The query sets are available at the following url: `ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast/queries/human`. For EST mapping, we form our query set Q by randomly selecting 1000 human ESTs (average length 490) from Expressed Sequence Tags database from NCBI `https://www.ncbi.nlm.nih.gov/dbEST`. For each length l, we define $Q(l)$ to be the subset of Q that has a non-empty $HSLA(DB, q, l, t)$ and refer to these as the *mappable queries* for length l.

Query Processing: For every query q in the query set, we run BLAST using the `blastn` program with the `-task megablast` option using its default settings except we select MEM_{k^*} value using `-word-size` k^*, we use multiple values of w, and we set the matching threshold, also known as identity percentage, using `-perc-identity 96% for` $l = 50$ `and -perc-identity 97% for all other` l. This will return $HSLA(DB, q, k^*, t, k', w)$ where every alignment must have an MEM_{k^*}. That is, the match percentage t will be satisfied, but the lengths are only guaranteed to be at least k^*, not l. We filter out any $HSLAs$ that are too short to produce $HSLA(DB, q, l, t, k', w)$.

False Positives: For any query q and any w, we report the number of false positives $FP(q, w)$ as the number of alignments in $HSLA(DB, q, k^*, t, k', w) - HSLA(DB, q, l, t)$. This should be very close to the number of MEM_{k^*}s that do not extend to alignments in $HSLA(DB, q, l, t)$; the two numbers might differ if multiple MEM_{k^*}s are part of the same alignment in $HSLA(DB, q, k^*, t) - HSLA(DB, q, l, t)$.

Experimental System: We run the experiments on a cluster that runs the Community Enterprise Operating System (CentOS) 6.6. The cluster has 24 nodes where each node has

two 2.5Ghz 10-core Intel Xeon E5-2670v2 processors, 20 cores, and 500 GB.

The *HSLA* Evaluation Metrics: We evaluate the effectiveness of a given k'-mer index $SI(w)$ as a function of w and w_0 using three metrics: (1) index size reduction, (2) retention rate of $HSLAs$, and (3) query time reduction. For retention rate, we consider retention of all $HSLAs$ as denoted by $HSLA(DB, q, l, t)$ and short $HSLAs$ as denoted by $HSLA_{\text{short}}(DB, q, l, t)$. To help explain query time reduction, we also measure false positive reduction. We describe each metric in more detail.

For each $SI(w)$ and each choice of $w > w_0$, we define the sampled index size reduction as

$$SIR(w, w_0) = \frac{|SI(w)|}{|SI(w_0)|} \tag{4.1}$$

where $|I|$ is the size of index I. Index size is the sum of dictionary size, measured by counting the number of k'-mers, and inverted lists' size, measured by counting the number of k'-mer occurrences in all the inverted lists. Since the human genome is split into three volumes and we create a sampled index for each volume, we compute the total index size for all indexes over all volumes. For the total dictionary size, we take the union of all three dictionaries, and then we measure the total dictionary size by counting the number of k'-mers in the union set. For the total inverted lists' size, we take the sum over all three inverted lists' sizes.

For each $SI(w)$ and each choice of $w > w_0$, we report the full retention rate $RR(w, w_0)$ and the short retention rate $RR_{short}(w, w_0)$ which we define as follows. For w, w_0 and q, we define

$$RR(q, w, w_0) = \frac{|\, HSLA(DB, q, l, t, k', w)|}{|\, HSLA(DB, q, l, t, k', w_0)|} \tag{4.2}$$

and

$$RR_{short}(q, w, w_0) = \frac{|\, HSLA_{\text{short}}(DB, q, l, t, k', w)|}{|\, HSLA_{\text{short}}(DB, q, l, t, k', w_0)|} \tag{4.3}$$

We say that $RR(q, w, w_0)$ or $RR_{short}(q, w, w_0)$ is undefined if the denominator is 0. We typically report both ratios as percentages. We use all three volumes to get these percentages. We then set $RR(w, w_0)$ and $RR_{short}(w, w_0)$ to be the average of $RR(q, w, w_0)$ and $RR_{short}(q, w, w_0)$, respectively, where we only include query q in the average if $RR(q, w, w_0)$ or $RR_{short}(q, w, w_0)$, respectively, is defined. We report $RR(w, w_0)$ since this is a typical user query. We specifically define $RR_{short}(w, w_0)$ to fairly compare empirical retention rate with expected retention rate. Intuitively, $RR_{short}(w, w_0)$ focuses on the hardest to retain $HSLAs$.

For each $SI(w)$ and each choice of $w > w_0$, we report the average query time reduction percentage $QTR(w, w_0)$ which we define as follows. We start by defining the query time $QT(q, w)$ for a given query q and sampled index $SI(w)$ (including $SI(w_0)$) as follows. We process each query q on $SI(w)$ five times using BLAST and we set $QT(q, w)$ to be the median of the five values. Since $SI(w)$ is partitioned into three volumes, the query time for a given q is the sum of the query times over the three volumes. The query time reduction $QTR(q, w, w_0)$ is then

$$QTR(q, w, w_0) = \frac{QT(q, w)}{QT(q, w_0)}. \tag{4.4}$$

Finally, the average query time reduction $QTR(w, w_0)$ is the average of $QTR(q, w, w_0)$ over all q.

Finally, to help explain the query time reduction results, for each $SI(w)$ and each choice of $w > w_0$, we report the average false positive reduction rate $FPR(w, w_0)$ which we define as follows. For a given query q and sampled index $SI(w)$ (including $SI(w_0)$), we define

$FP(q, w)$ to be the number of false positive; that is, $HSLAs$ that do not lead to elements of $HSLA(DB, q, l, t)$ when we apply the second, more expensive, extension phase. We believe that $FP(q, w)$ decreases as w increases, and this may help explain any reduction in query time. To test this, we define the false positive reduction rate $FPR(q, w, w_0)$ to be

$$FPR(q, w, w_0) = \frac{FP(q, w)}{FP(q, w_0)} \tag{4.5}$$

Finally, the average false positive reduction rate $FPR(w, w_0)$ is the average of $FPR(q, w)$ over all q.

EST Mapping Evaluation Metrics: For each soft sampled index $SI(w)$ and a given length l, we report its retention rate, $RR_{map}(w, l)$, as the percentage of $Q(l)$ such that *all* of $HSLA(DB, q, l, t)$ is found using $SI(w)$. We use this requirement because this implies that the mapping result for $SI(w)$ for the given query q will be identical to the mapping result for $SI(w_0)$ and q regardless of the mapping procedure used. Otherwise, at least one highly similar local alignment is lost and we pessimistically assume that the mapping result would be lost as well.

More formally, for a given mappable queries set $Q(l)$ and k'-mer index $SI(w)$, we define the set $Q'(l) \subset Q(l)$ as follows

$$Q'(l) = \{\forall q \in Q(l) \mid HSLA(DB, q, l, t, k', w) = HSLA(DB, q, l, t, k', w_0)\} \tag{4.6}$$

Then, we define the index retention rate RR_{map} as follows:

$$RR_{map}(w, w_0) = \frac{|Q'(l)|}{|Q(l)|} \tag{4.7}$$

We also report the effect of w on query time using the same process as with $HSLA$, namely, running each query five times, taking the median time, and then reporting the average reduction in query time over all 1000 queries. Note that we use all queries rather than just the mappable queries when reporting query time.

4.6.2 Analytical modeling

We now describe how we analytically model two of the evaluation metrics, index size reduction and retention rate.

Predicting Index Size: We first show how we compute the expected size of a sampled index $SI(w)$. For the dictionary, we assume the k'-mer dictionary is full and thus the size of a k'-mer dictionary is $4^{k'}$ entries which, in our case, is 4^{12}. This may not be accurate, but since the dictionary size is typically much smaller than the inverted lists size given $k' = 12$, this is accurate enough. The number of k'-mer occurrences stored in the inverted lists is simply $(D - S(k' + 1))/w$ where $D = |DB|$, the number of positions in DB, and S is the number of distinct sequences in DB. Thus, the predicted size of $SI(w)$ is simply

$$size(SI(w)) = 4^{k'} + \frac{D - S(k' + 1)}{w} \tag{4.8}$$

Predicting Retention Rate: We present a new analytical model to compute the expected retention rate of $HSLAs$ in $HSLA_{\text{short}}(DB, q, l, t)$. We start by presenting Kent's

analytical model [36] where he essentially assumed $w = k' = k^*$ in his model. We then propose a new model that we refer to as the BLAST model to account for typical parameters used in BLAST searches. We refer to the expected retention rates as $E[RR_K]$ and $E[RR_B]$. For both retention rates, we make a few simplifying assumptions and refer to the two models generically as $E[RR]$ when describing these common assumptions. First, we restrict our attention to $HSLAs$ that have length exactly l. Second, we assume each $HSLA$ in $HSLA_{\text{short}}(DB, q, l, t, k', w)$ is retained with the same probability, and this probability is independent of other $HSLAs$. This implies

$$E[RR] = E[\frac{|\ HSLA_{\text{short}}(DB, q, l, t, k', w)|}{|\ HSLA_{\text{short}}(DB, q, l, t)|}] \tag{4.9}$$

simplifies to just $p(A)$

$$E[RR] = p(A) \tag{4.10}$$

which represents the probability that a short $HSLA$ A is retained. This allows us to focus on a single short $HSLA$ A in the rest of this analysis. Finally, we assume each position in A is independent of other positions and the probability that any position in A is a match is exactly t.

Kent's original retention rate model ($E[RR_K]$): We start with Kent's original model [36]. where he assumes $w = k^* = k'$. The number of k^*-mers that are guaranteed to be chosen from x within A is

$$T = \lfloor (|x| - k^* + 1)/w \rfloor \tag{4.11}$$

Furthermore, these k^*-mers will be adjacent to each other with no gaps. For A to be retained,

at least one of these chosen k^*-mers from x must exactly match the corresponding k^*-mer in y from q. The probability of such an exact match assuming each position is independent and that the overall match percentage within A is t is then

$$p = t^{k^*} \tag{4.12}$$

Since the sampled k^*-mers do not overlap, the probability that all fail to match is then $(1-p)^T$. Thus, the probability that at least one will match and alignment A will be found is $p(A) = 1 - (1-p)^T$. Since $E[RR_K] = p(A)$, we have

$$E[RR_K] = 1 - (1-p)^T \tag{4.13}$$

BLAST retention rate model ($E[RR_B]$): To extend this analysis to the typical BLAST setting with distinct w, k^* and k', we must modify the formula in two ways. The first key issue is that we sample k'-mers but then extend them to search for k^*-mers. The sampled k'-mer must be an exact match, which again happens with probability $p = t^{k'}$ The key issue after this is whether this can be extended to an MEM_{k^*}. Suppose this can extend exactly $0 \leq l \leq k^* - k' - 1$ characters to the left before we get a mismatch. We then need it to extend at least $k^* - l - k'$ characters to the right. The probability we can extend exactly l characters to the left is $t^l(1-t)$. The probability we can extend at least $k^* - l - k'$ characters to the right is $t^{k^*-k'-l}$. Thus, the probability that we have a k'-mer, it extends exactly $0 \leq l \leq k^* - k' - 1$ characters to the left, and it extends at least $k^* - l - k'$ characters

to the right is then $t^{k'}t^{l}(1-t)t^{k^*-k'-l} = t^{k^*}(1-t)$ There are $k^*-k'-1$ choices for l leading to a final probability of $(k^*-k'+1)t^{k^*}(1-t)$. The other possibility is that it extends at least k^*-k' characters to the left which occurs with probability t^{k^*} giving us a total probability of

$$p' = (k^* - k' - 1)t^{k^*}(1-t) + t^{k^*} \tag{4.14}$$

The second key issue is that in Equation 4.11, we used the floor function as this is the number of k^*-mers from x within A that are guaranteed to be chosen. Using the floor function ignores the possibility that we may have an additional k^*-mer chosen from x. That is, the number of k^*-mers from k that will be sampled might be either

$$T_f = \lfloor (|x| - k^* + 1)/w \rfloor \tag{4.15}$$

$$T_c = \lceil (|x| - k^* + 1)/w \rceil \tag{4.16}$$

where $T_f = T$ from Equation 4.11. If we assume that each possible window for w is equally likely, then

$$p(T_f) = \frac{w \times T_c - (|x| - k^* + 1)}{w} \tag{4.17}$$

$$p(T_c) = 1 - p(T_f) \tag{4.18}$$

For the case where $\lfloor (|x| - k^* + 1)/w \rfloor = (|x| - k^* + 1)/w$, $p(T_f) = 0$ which means $p(T_c) = 1$ so the result is still correct.

In our new BLAST model, we update Equation 4.13 replacing p with p' and replacing T with T_c and T_f as follows.

$$E[RR_B] = 1 - p(T_f)(1-p')^{T_f} - p(T_c)(1-p')^{T_c} \tag{4.19}$$

We will compare both Kent's model and our new BLAST model in our results.

4.7 Results and discussion

We report the impact of sampling on the efficiency of a k'-mer index on the index size and query performance. We report both the expected and the actual impact of sampling.

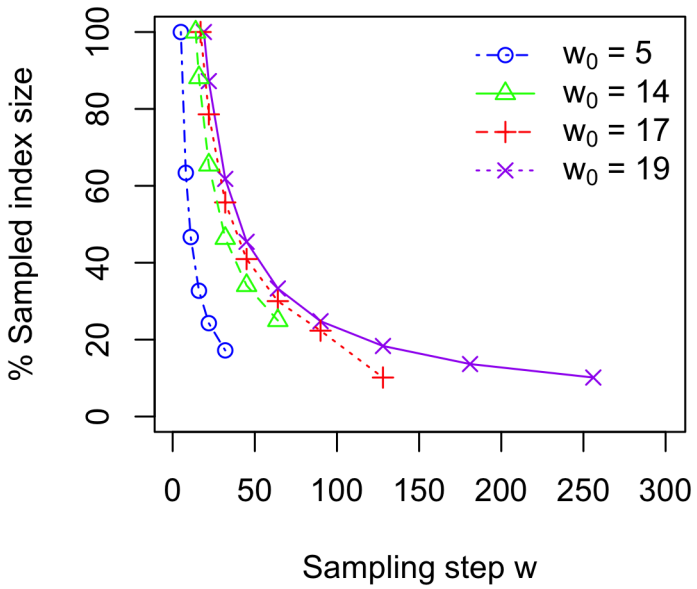
4.7.1 Index size

As expected, the index size is inversely proportional to the sampling step w. This means that soft sampling does lead to a significant reduction in space when compared to hard sampling. For example, when w/w_0 is roughly 1.7 and 4.4, the index size reduces by 38% and 74% for all values of l we considered. The percentage of reduction increases as l increases. For example when $l = 400$ and w/w_0 is almost 10, the index size reduces by 90%.

With hard sampling $w_0 = k^* - k' + 1$, the space reduction is limited by k^*. With soft sampling $w > k^* - k' + 1$, w is limited primarily by l, where typically $l \gg k^*$ (see Table 4.2). We plot results for the percentage reduction in index size in Fig 4.3. Since the expected index size (see Eq 4.8) and the actual index size are almost identical, the expected size is omitted.

Sampling reduces index size because it reduces the number of sampled k'-mers leading to a factor of w reduction in inverted lists size, the dominant component of index size. On the other hand, although sampling does reduce dictionary size, the reduction is relatively small and does not greatly affect the final index size. For example when w/w_0 is roughly 1.7 and

Figure 4.3: **The sampled index $SI(w)$ size (percentage) as a function of sampling step size w of $SI(w)$ versus sampled index $SI(w_0)$.** The k'-mer indexes are built with $k' = 12$ and $w \geq w_0$ where $w_0 = l - k + 1$.



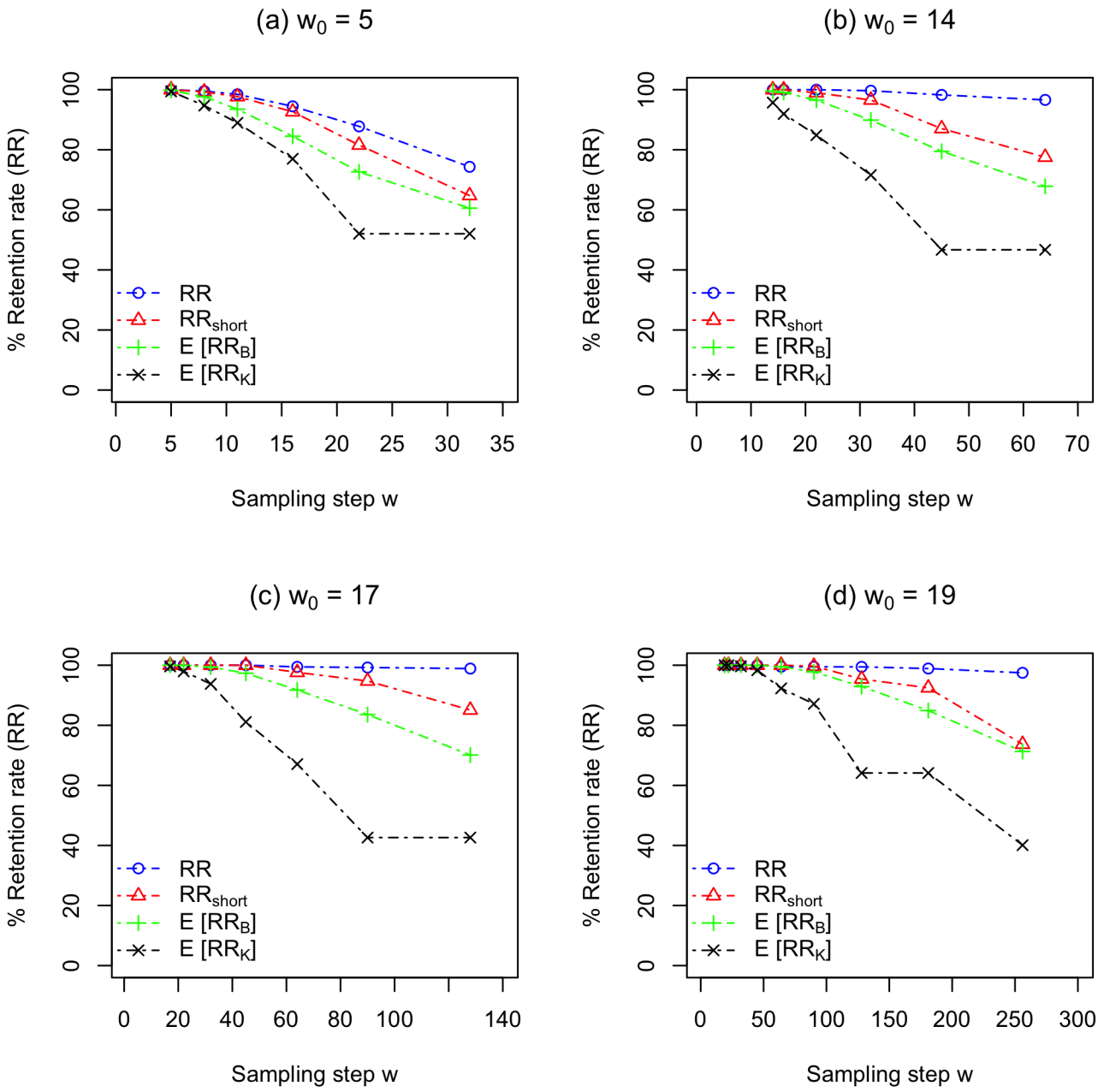
4.4, the dictionary is about 9% and 17% of the the index size and the average reduction in the dictionary size is 9% and 27% respectively.

4.7.2 Retention rate of *HSLA*s

We first examine $RR(w, w_0)$ to study how the overall $HSLA$ retention rate changes as a function of w, w_0, and l. We first observe that $RR(w, w_0)$ improves as l increases. In particular, as can be seen from our $RR(w, w_0)$ results from Figure 4.4, if we look at choices for w and w_0 that have a similar ratio w/w_0, the $RR(w, w_0)$ retention result is higher for larger l. In particular, whereas $RR(32, 5)$ for $l = 50$ falls below 80%, $RR(w, w_0) \geq 96.6\%$ for $l \geq 100$ for all tested values of w, and $RR(256, 30) = 97.5\%$ for $l = 400$. Thus, for

large values of l, we can use soft sampling where w/w_0 approaches even 10 and still achieve retention rates of close to 100%.

Figure 4.4: **The actual $HSLA$ retention rate $RR(w, w_0)$, the actual short HSLA retention rate $RR_{short}(w, w_0)$, and the expected short $HSLA$ retention rate using both Kent's model $E[RR_K(w, w_0)]$ and BLAST model $E[RR_B(w, w_0)]$ for (a) $l = 50$, $w_0 = 5$, (b) $l = 100$, $w_0 = 14$, (c) $l = 200$, $w_0 = 17$, and (d) $l = 400$, $w_0 = 30$.** For other parameters values see Table 4.2.



We now focus on the retention rate of short $HSLAs$. First we compare $RR(w, w_0)$ with $RR_{short}(w, w_0)$ as a function of w, w_0, and l. Figure 4.4 also contains our $RR_{short}(w, w_0)$ results. *We observe that if* $w/w0 < 4$*, then the difference between* $RR(w, w_0)$ *and* $RR_{short}(w, w_0)$ *is small (less than 4%) for almost all choices of* l; the one outlier is $l = 100$ where we see a difference of 11% for $w/w_0 = 45/14 \approx 3.2$. For example, for all values of l except 100, the difference between $RR(w, w_0)$ and $RR_{short}(w, w_0)$ is at most 1.8%. We do see a significant difference between $RR(w, w_0)$ and $RR_{short}(w, w_0)$ for our largest choices of w which means that $RR_{short}(w, w_0)$ does fall off by $w/w_0 = 10$ or so. Thus, in contrast to $RR(w, w_0)$, there is an upper limit to how much we can soft sample before $RR_{short}(w, w_0)$ suffers.

Next we examine how $RR_{short}(w, w_0)$ changes as a function of w, w_0, and l. Similar to $RR(w, w_0)$, we observe $RR_{short}(w, w_0)$ generally improves as l increases given roughly the same ratio of w/w_0. For example, when w/w_0 is roughly 6, $RR_{short}(w, w_0)$ is 65%, and 85%, and 95% for l equal to 50, 200, and 400, respectively. In general, we can retain 90% more short $HSLAs$ for either small w/w_0 ratios (less than 2 or 3) or large l values (200 or 400).

We now want to compare empirical retention rate with predicted retention rate as a function of w, w_0 and l. Comparing $RR(w, w_0)$ to $E[RR_B(w, w_0)]$ is not fair as $RR(w, w_0)$ includes many alignments significantly longer than l whereas $E[RR_B(w, w_0)]$ focuses only on alignments with length exactly l. To more fairly compare empirical retention rate to expected retention rate, we compare $RR_{short}(w, w_0)$, where the length of the alignment is in the range $[l, (2-t)l]$, with $E[RR_B(w, w_0)]$, where an alignment is assumed to have a length l. We consider $HSLAs$ with length up to $(2-t)l$ to ensure there are a reasonable number of $HSLAs$. We also note that if we assume that the $(1-t)l$ errors were all insertions rather than substitutions, this would increase the length of the $HSLA$ to $(2-t)l$. Figure 4.4 also

contains our $E[RR_B(w, w_0)]$ results.

We observe that the BLAST model predicts actual retention rate of short HSLAs with reasonable accuracy, particularly for small w/w_0 and for larger l. For example, the typical difference between $RR_{short}(w, w_0)$ and $E[RR_B(w, w_0)]$ when $w/w_0 \leq 2$ is less than 5% for all choices of l and is less than 1% for large $l = 200$ and $l = 400$. The typical difference stays below 10% for almost all choices of w/w_0 and l with only a few exceptions. The difference between $RR_{short}(w, w_0)$ and $E[RR_B(w, w_0)]$ does grow as w/w_0 increases, but at a relatively slow rate, typically maximized at the largest choice of w/w_0, though this does not hold for $l = 400$. We do note that we have relatively few empirical $HSLAs$ for the large w values for $l = 400$, so perhaps with more samples, the difference between $RR_{short}(w, w_0)$ and $E[RR_B(w, w_0)]$ might increase for these l and w choices.

Finally, we compare the predictions from Kent's model $E[RR_K(w, w_0)]$ and our new BLAST model $E[RR_B(w, w_0)]$. *Our new BLAST model is significantly more accurate than Kent's model, especially as w/w_0 increases.* For example, for w/w_0 equal to 1.7, 3.4 and 4.7, $E[RR_K]$ is on average less than $E[RR_B]$ by 5%, 18%, and 23%, respectively, for all l values we consider. Kent's model has several issues. First, because of the floor function used in Eq. 4.11, it underestimates the number of sampled k^*-mers from a given $HSLA$ resulting in common retention rate predictions for multiple values of w. For example when $l = 100$, $E[RR_K] = 46.70\%$ for both $w = 45$ and $w = 64$. The second flaw is that Kent's model was not designed to handle different values for k', k^*, and w which is what is typically used in BLAST. Because our BLAST model is designed to overcome both issues, it achieves better results, particularly for larger w/w_0 and for larger l.

4.7.3 Possible improvements for the BLAST model

While our new BLAST model is much more accurate than Kent's original model, it still underestimates actual retention rates for large w/w_0. We now explore possible explanations for this underestimate. We believe the fundamental problem with our new BLAST model (as well as Kent's model) is that for any $HSLA_{\text{short}}(DB, q, l, t)$, it only assumes that each position is a match with probability t.

We demonstrate the shortcomings of this assumption in two different ways. We first show that using this assumption, we greatly underestimate the length of the maximum MEM within any $HSLA$; we refer to this maximum MEM length as MAX-MEM. Long MEMs are relevant because long MEMs significantly increase the likelihood of recovering an $HSLA$. For example, if an $HSLA$ includes an MEM of length $w + k - 1$, then it is guaranteed the $HSLA$ will be found since one k-mer is guaranteed to be chosen from within the MEM.

We perform this comparison as follows. We first obtain an empirical distribution of MAX-MEM by recording the length of the longest MEM in every $HSLA$ in $RR_{short}(w, w_0)$. We then use the BLAST model's fundamental assumption that each position in an $HSLA$ is identical with probability t to create a corresponding predicted distribution of MAX-MEM. For this predicted distribution of MAX-MEM, we assume that the length of the $HSLA$ is l, the number of mismatches is exactly $(1 - t)l$, and each position is equally likely to be a mismatch. All told, there are l choose $(1-t)l$ different combinations of errors that are equally likely. We can then compute the predicted distribution by enumerating all possibilities for $l = 50$ and $l = 100$.

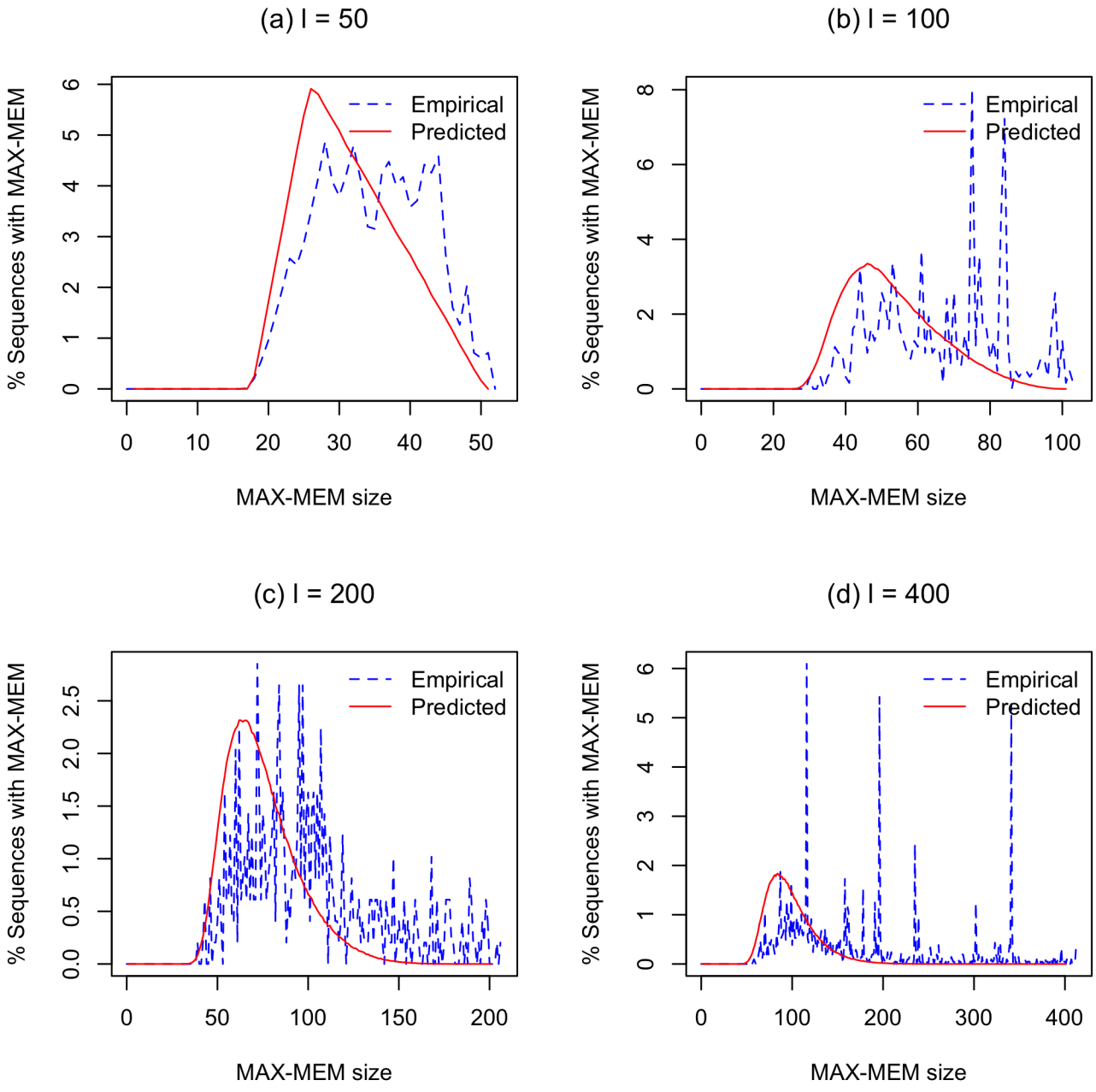
For $l > 100$, it takes too much time to enumerate all possibilities. Thus, we use Monte Carlo simulation to compute a second predicted distribution for MAX-MEM. We create short

$HSLAs$ as follows. We start with an alignment of length l, a set S of mismatch positions which is initialized to empty, and a count C of the number of mismatch positions which is initially 0. Then, we repeatedly choose a position in the range $[0, L-1]$ uniformly at random. If the position is not in S, we add the position to S and increment C by one. Otherwise, we do nothing with the chosen position and choose another one. When C reaches $(1-t)l$, we stop with a complete $HSLA$. We then record its longest MEM. We do this until we have recorded one million such longest MEMs.

For $l = 50$ and $l = 100$, Monte Carlo simulation and complete enumeration produce essentially identical distributions for MAX-MEM. Thus, we only show results from our Monte Carlo simulations since these cover all choices of l. We show the results for our experimental and Monte Carlo distribution of MAX-MEM for all four choices of l in Figure 4.5. We observe that the empirical MAX-MEM distribution is weighted more heavily towards longer MEMs than the predicted distribution. This demonstrates that the assumptions used in the BLAST model do not correctly predict the distribution for MAX-MEM length; in general, they underestimate the probability for finding longer MEMs. While the distribution of MAX-MEM is not identical to retention rate of $HSLAs$, this finding provides evidence that we need stronger assumptions to better predict retention rate of $HSLAs$.

We now show another fundamental flaw with the base assumption of the BLAST model. Consider an $HSLA$ of length l, and suppose we assume that each query position is independent and matches its corresponding database position with probability t (similar to the BLAST model's assumptions). Then, there is a $(1-t)$ probability that each position does not match. In this scenario, the total number of mismatches has the binomial distribution $Bin(l, (1-t))$ which has an expected number of mismatches of exactly $(1-t)l$. If the number of mismatches exceeds this expected value, we would no longer have an $HSLA$, but this is

Figure 4.5: **Distribution of predicted and empirical MAX-MEM lengths in $HSLAs$.** The predicted MAX-MEM lengths are computed from a Monte Carlo simulation. (a) $l = 50$, $t = 96\%$, (b) $l = 100$, $t = 97\%$, (c) $l = 200$, $t = 97\%$, and (d) $l = 400$, $t = 97\%$.



clearly contradicts with the first assumption that we start with an $HSLA$. The probability that the number of mismatches exceeds $(1 - t)l$ is given in Table 4.3. Given this weak as-

sumption, the BLAST model essentially starts with a probability ranging from 32% to 43% that the given $HSLA$ is not an $HSLA$.

Table 4.3: **The probability that the number of mismatches exceeds $(1-t)l$ for various choices of l and t.**

l	t	Probablity mismatches exceeds $(1-t)l$
50	0.96	32.3%
100	0.97	35.3%
200	0.97	39.4%
400	0.97	42.4%

Under the assumption that the number of mismatches in a $HSLA$ follow $Bin(l,(1-t))$, Kents's model underestimates the the existence of the $HSLA$ by 30% - 40%.

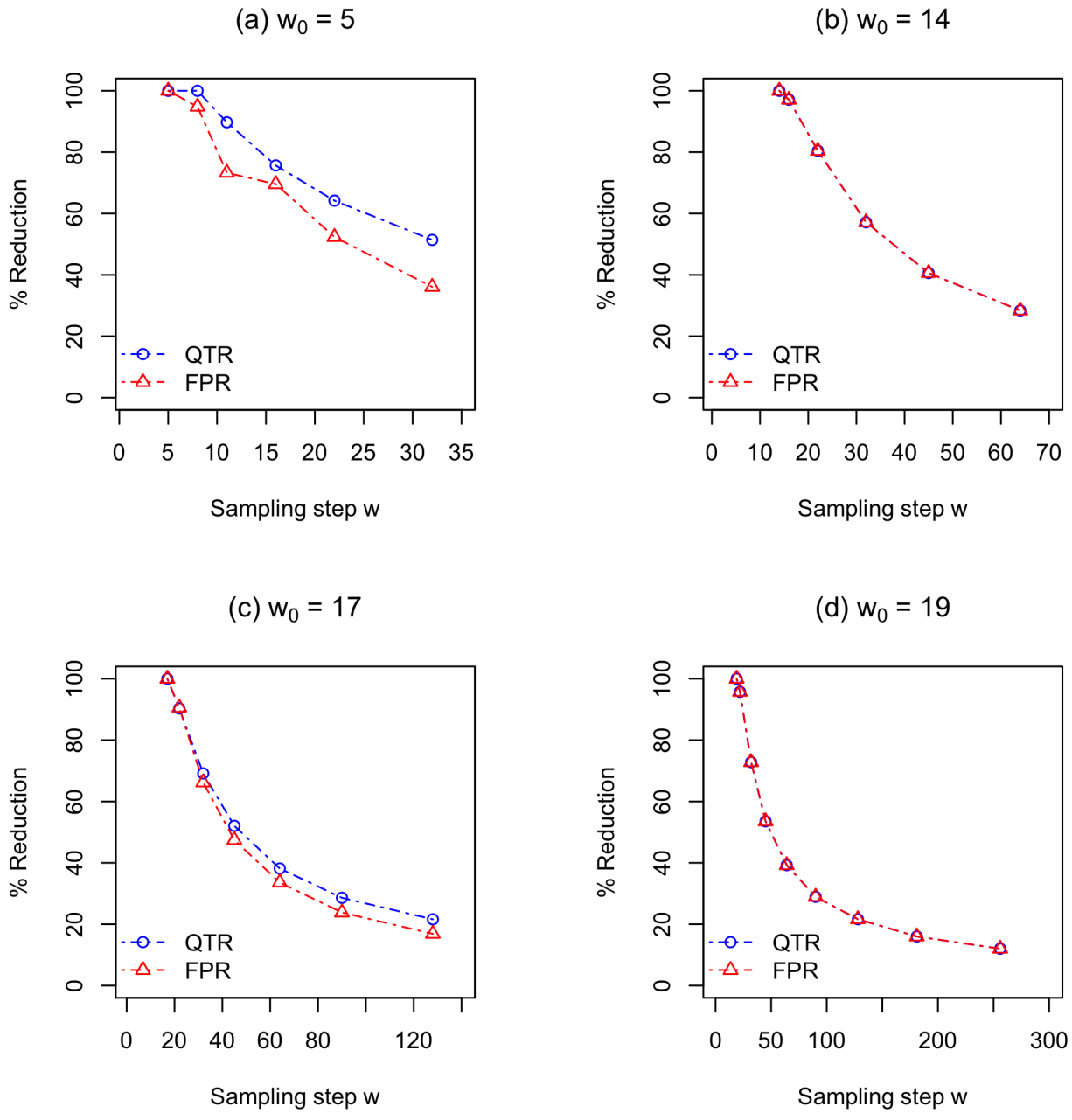
We have shown that the weak assumption used in the BLAST model (1)underestimates the probability of longer MAX-MEMs and (2) gives a significant probability for $HSLAs$ to not be $HSLAs$. Taken together, we believe a new model with stronger assumptions is needed to produce more accurate predictions about retention rate of short $HSLAs$.

4.7.4 Query time

We now examine how soft sampling affects query time. *The query time is approximately inversely proportional to the sampling step w for all l values without significantly reducing retention rate $RR(w, w_0)$.* For example, for $l = 200$, the median query time for hard sampling is 26 hours. When w/w_0 is 3.8 and 5.3, the median query time reduction percentages are 64.51% (median query time 10 hours) and 73.38% (median query time 7.4 hours), respectively, while maintaining $RR > 99\%$. Similarly, for $l = 400$, the median query time for hard sampling is 18.6 hours. When w/w_0 is 6.7 and 9.3, the query time reduction percentages are 78.36% (median query time 4.3 hours) and 83.99% (median query time 3.3 hours), respectively, while maintaining $RR > 99\%$. Fig 4.6 shows our full query time results represented

as query time reduction (QTR) percentages.

Figure 4.6: **The average median query time reduction (QTR) percentages and the actual false positive reduction (FPR) percentages as a function of sampling step w.** (a) $l = 50$, $w_0 = 5$, (b) $l = 100$, $w_0 = 14$, (c) $l = 200$, $w_0 = 17$, and (d) $l = 400$, $w_0 = 30$. For other parameters values see Table 4.2.



The reduction in false positive rate mostly explains the reduction in query time. Re-

call that false positives are alignments in $HSLA(DB, q, k^*, t) - HSLA(DB, q, l, t)$, which is roughly the number of MEM_{k^*}s that do not extend into alignments in $HSLA(DB, q, l, t)$. This implies much of the query time is spent ruling out false positives and that using soft sampling not only has little affect on retention rate but also significantly reduces false positives. Our full false positive reduction rate results are shown in Fig 4.6. As can be seen from this figure, the plots for false positive reduction rate (FPR) and query time reduction (QTR) percentage are very similar.

4.7.5 Mapping results

We report our retention rate and query time results for EST mapping in Tables 4.4, 4.5, and 4.6. *Our results show that the number of mappable queries that retain all $HSLA$s is very high even when we use soft sampling.* Furthermore, we achieve significant reductions in query processing time. Recall that a query q is mappable is there is at least one $HSLA$ between q and the reference (the human genome in our case). When an index $SI(w)$ is used where $w > w_0$, a query is lost if even a single $HSLA$ is not found by $SI(w)$.

Table 4.4: **The retention rate (RR_{map}) and query time reduction (QTR) results for all 1000 queries when $l = 50$ and $w_0 = 5$, where 879 were mappable queries.**

w	# retained queries	RR_{map}	QTR
5	879	100.00%	100.00%
8	876	99.66%	100.00%
11	864	98.29%	89.72%
16	849	96.59%	75.67%
22	811	92.26%	64.19%
32	677	77.02%	51.40%

Using soft sampling, we are able to greatly reduce the index size, significantly reduce query time, and correctly map more than 95% of the mappable queries for $l \geq 100$. In fact,

Table 4.5: **The retention rate (RR_{map})and query time reduction (QTR) results for all 1000 queries when $l = 100$ and $w_0 = 14$, where 794 were mappable queries.**

w	# retained queries	RR_{map}	QTR
14	794	100.00%	100.00%
16	794	100.00%	95.60%
22	794	100.00%	85.67%
32	793	99.87%	83.07%
45	789	99.37%	80.26%
64	784	98.74%	76.68%

Table 4.6: **The retention rate (RR_{map})and query time reduction (QTR) results for all 1000 queries when $l = 200$ and and $w_0 = 17$, where 528 were mappable queries.**

w	# retained queries	RR_{map}	QTR
17	528	100.00%	100.00%
22	528	100.00%	93.76%
32	528	100.00%	91.91%
45	525	99.43%	87.07%
64	528	100.00%	84.08%
90	523	99.05%	80.36%
128	506	95.83%	78.76%

for $l \geq 100$, we correctly map almost 99% of the mappable queries for w/w_0 approaching 5. For $l = 200$, we correctly map at least 95% of the mappable queries for $w/w_0 = 7.5$. For $l = 50$, we still see good retention rates but the drop off is a bit faster. Specifically, for $l = 50$, for w/w_0 approaching 3, we correctly map 96% of the mappable queries. For w/w_0 between 4 and 5, we correctly map 92% of the mappable queries, and for w/w_0 between 6 and 7, we correctly map 77% of the mappable queries. Finally, the actual retention rates may be even better than the ones reported as we required that all $HSLAs$ be retained whereas mapping may proceed properly even if some $HSLAs$ are lost.

In human EST mapping, it is common to use $HSLAs$ of length $l = 100$ to search for

the best mapping [36, 66, 99]. To study a broader range of possibilities, we also consider $l = 50$ (more $HSLAs$ and thus more mappable queries) and $l = 200$ (fewer $HSLAs$ and thus fewer mappable queries). Our results imply that soft sampling can be used with relatively small loss in sensitivity for the commonly used case of $l = 100$. Given that the index sizes are significantly reduced and query times are also reduced, soft sampling may allow for EST mapping using k-mer based methods for larger genomes with only a small loss in sensitivity.

Chapter 5

Conclusions and future work

In many biological applications, k-mer indexes are widely used to find similar sequences. A major problem with using k-merdataset indexes is that the index size is significantly larger than the underlying datasets. To ensure k-mer indexes are feasible, k-mer index size and query time must be mitigated. One of the most effective approaches to mitigate k-mer index size and query time is to use sampling. In this dissertation, I systematically study the effect of sampling on k-mer indexes. Specifically, I study how key parameters such as sampling strategy, sampling step size, and k-mer size affect index size, query time, and query accuracy. The results show that the choice of the right sampling method is not trivial, and in fact has a great impact on both the index size and query time.

We now summarize our main conclusions. When comparing fixed and minimizer sampling, our results show that fixed sampling typically answers queries at least as fast as minimizer sampling and often is faster when both methods to use the same space. In particularly for small k values, such as $k \leq 16$, fixed sampling is 37.43% to 51.11 % faster. As is common in many applications, there is a space versus speed tradeoff. Using a larger k requires more space but results in smaller query times. The key benefit of increasing k is that there are many fewer false positives which leads to much faster query processing.

Based on our experiments with the human genome and NCBI BLAST, fixed soft sampling achieves significant space and time savings while also retaining highly similar local

alignments ($HSLAs$) with much higher probabilities than predicted by analytical modeling. Even better, when applied to EST mapping, fixed soft sampling achieves significant space and time savings while retaining 98% of all mapping results when the length of $HSLAs$ is at least 100 characters, and the retention results may be even better as we pessimistically assume that losing even a single highly similar local alignment will lead to an incorrect mapping result. Further, because we performed all of our local alignments using BLAST, these results can be easily tested and adopted by other researchers.

We can extend this work in many directions. We would like to test the effectiveness of fixed soft sampling using other real biological datasets and in other applications such as clustering or SNP detection. Recently, a complementary approach of space-efficient referentially compressed search indexes has been proposed to support similarity searches on genome datasets [94, 95]. In this method, genomes are compressed against some reference genome(s). Given a query, the index then searches two parts: the reference and all genome-specific individual differences. Both parts are saved in compressed suffix trees. Danek *et al.* [19] extend reference-based compression with the use of a k-mer index. We think employing the complementary approach of reference compression in unison with fixed soft sampled k-mer indexes may be fruitful.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Athena Ahmadi, Alexander Behm, Nagesh Honnalli, Chen Li, Lingjie Weng, and Xiaohui Xie. Hobbes: Optimized gram-based methods for efficient read alignment. *Nucleic Acids Research*, 40(6):e41–e41, 2012.

[3] Can Alkan, Jeffrey M Kidd, Tomas Marques-Bonet, Gozde Aksay, Francesca Antonacci, Fereydoun Hormozdiari, Jacob O Kitzman, Carl Baker, Maika Malig, Onur Mutlu, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41(10):1061–1067, 2009.

[4] Meznah Almutairy and Eric Torng. The effects of sampling on the efficiency and accuracy of k- mer indexes: Theoretical and empirical comparisons using the human genome. *PLOS ONE*, 12(7):e0179046, 2017.

[5] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[6] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[7] Sasha K Ames, David A Hysom, Shea N Gardner, G Scott Lloyd, Maya B Gokhale, and Jonathan E Allen. Scalable metagenomic taxonomy classification using a reference genome database. *Bioinformatics*, 29(18):2253–2260, 2013.

[8] Richard Arratia, Louis Gordon, and Michael Waterman. An extreme value theory for sequence matching. *The annals of statistics*, pages 971–993, 1986.

[9] Ergude Bao, Tao Jiang, Isgouhi Kaloshian, and Thomas Girke. SEED: Efficient clustering of next-generation sequences. *Bioinformatics*, 27(18):2502–2509, 2011.

[10] Alexander Behm, Shengyue Ji, Chen Li, and Jiaheng Lu. Space-constrained gram-based indexing for efficient approximate string search. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 604–615. IEEE, 2009.

[11] Medha Bhagwat, Lynn Young, and Rex R Robison. Using BLAT to find sequence similarity in closely related genomes. *Current Protocols in Bioinformatics*, pages 10–8, 2012.

[12] Jacek Blazewicz, Wojciech Frohmberg, Michal Kierzynka, Erwin Pesch, and Pawel Wojciechowski. Protein alignment algorithms with an efficient backtracking routine on multiple gpus. *BMC bioinformatics*, 12(1):181, 2011.

[13] Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *Journal of Experimental Algorithmics (JEA)*, 16:1–1, 2011.

[14] Michael Brudno, Sanket Malde, Alexander Poliakov, Chuong B Do, Olivier Couronne, Inna Dubchak, and Serafim Batzoglou. Glocal alignment: Finding rearrangements during alignment. *Bioinformatics*, 19(suppl_1):i54–i62, 2003.

[15] Jeremy Buhler, Uri Keich, and Yanni Sun. Designing seeds for similarity search in genomic DNA. In *Proceedings of the Seventh Annual International Conference on Research in Computational Molecular Biology*, pages 67–75. ACM, 2003.

[16] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Digital Equipment Corporation*, 1994.

[17] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden. BLAST+: Architecture and applications. *BMC Bioinformatics*, 10(1):421, 2009.

[18] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *Research in Computational Molecular Biology*, pages 35–55. Springer, 2014.

[19] Agnieszka Danek, Sebastian Deorowicz, and Szymon Grabowski. Indexes of large genome collections on a PC. *PLOS ONE*, 9(10):e109384, 2014.

[20] AP Jason de Koning, Wanjun Gu, Todd A Castoe, Mark A Batzer, and David D Pollock. Repetitive elements may comprise over two-thirds of the human genome. *PLOS Genetic*, 7(12):e1002384, 2011.

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

[22] Todd Z DeSantis, Keith Keller, Ulas Karaoz, Alexander V Alekseyenko, Navjeet NS Singh, Eoin L Brodie, Zhiheng Pei, Gary L Andersen, and Niels Larsen. Simrank: Rapid and sensitive general-purpose k-mer search tool. *BMC ecology*, 11(1):11, 2011.

[23] Naryttza N Diaz, Lutz Krause, Alexander Goesmann, Karsten Niehaus, and Tim W Nattkemper. TACOA–taxonomic classification of environmental genomic fragments using a kernelized nearest neighbor approach. *BMC Bioinformatics*, 10(1):56, 2009.

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

[25] Robert C Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461, 2010.

[26] Peter Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[27] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[28] Mohammadreza Ghodsi, Bo Liu, and Mihai Pop. DNACLUST: Accurate and efficient clustering of phylogenetic marker genes. *BMC bioinformatics*, 12(1):271, 2011.

[29] Louis Gordon, Mark F Schilling, and Michael S Waterman. An extreme value theory for long head runs. *Probability Theory and Related Fields*, 72(2):279–287, 1986.

[30] Faraz Hach, Fereydoun Hormozdiari, Can Alkan, Farhad Hormozdiari, Inanc Birol, Evan E Eichler, and S Cenk Sahinalp. mrsFAST: A cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7(8):576–577, 2010.

[31] Ayat Hatem, Doruk Bozdağ, Amanda E Toland, and Ümit V Çatalyürek. Benchmarking short sequence mapping tools. *BMC Bioinformatics*, 14(1):1, 2013.

[32] Liisa Holm, Chris Sander, et al. Mapping the protein universe. *Science-New York Then Washington*, pages 595–602, 1996.

[33] Farhad Hormozdiari, Faraz Hach, S Cenk Sahinalp, Evan E Eichler, and Can Alkan. Sensitive and fast mapping of di-base encoded reads. *Bioinformatics*, 27(14):1915–1921, 2011.

[34] Kris Irizarry, Vlad Kustanovich, Cheng Li, Nik Brown, Stanley Nelson, Wing Wong, and Christopher J Lee. Genome-wide analysis of single-nucleotide polymorphisms in human expressed sequences. *Nature Genetics*, 26(2):233–236, 2000.

[35] Kazutaka Katoh, Kazuharu Misawa, Kei-ichi Kuma, and Takashi Miyata. MAFFT: A novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic acids research*, 30(14):3059–3066, 2002.

[36] W James Kent. BLAT-the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[37] Zia Khan, Joshua S Bloom, Leonid Kruglyak, and Mona Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.

[38] Nilesh Khiste and Lucian Ilie. E-MEM: Efficient computation of maximal exact matches for very large genomes. *Bioinformatics*, 31(4):509–514, 2015.

[39] You Jung Kim, Andrew Boyd, Brian D Athey, and Jignesh M Patel. miBLAST: Scalable evaluation of a batch of nucleotide sequence queries with blast. *Nucleic Acids Research*, 33(13):4335–4344, 2005.

[40] Ian Korf and Warren Gish. MPBLAST: Improved blast performance with multiplexed queries. *Bioinformatics*, 16(11):1052–1053, 2000.

[41] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[42] Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *Proceedings of the 33rd international conference on Very large data bases*, pages 303–314. VLDB Endowment, 2007.

[43] Heng Li. Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, page btw152, 2016.

[44] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[45] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272, 2010.

[46] Weizhong Li and Adam Godzik. Cd-hit: A fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006.

[47] Yang Li et al. MSPKmerCounter: A fast and memory efficient approach for k-mer counting. *arXiv preprint arXiv:1505.06550*, 2015.

[48] Yang Li, Pegah Kamousi, Fangqiu Han, Shengqi Yang, Xifeng Yan, and Subhash Suri. Memory efficient minimum substring partitioning. In *Proceedings of the VLDB Endowment*, volume 6(3), pages 169–180. VLDB Endowment, 2013.

[49] Hao Lin, Zefeng Zhang, Michael Q Zhang, Bin Ma, and Ming Li. ZOOM! zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.

[50] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[51] Bin Ma, John Tromp, and Ming Li. PatternHunter: Faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[52] M McIlroy. Development of a spelling list. *Communications, IEEE Transactions on*, 30(1):91–99, 1982.

[53] Hamish McWilliam, Weizhong Li, Mahmut Uludag, Silvano Squizzato, Young Mi Park, Nicola Buso, Andrew Peter Cowley, and Rodrigo Lopez. Analysis tool web services from the embl-ebi. *Nucleic acids research*, 41(W1):W597–W600, 2013.

[54] Giles Miclotte, Mahdi Heydari, Piet Demeester, Pieter Audenaert, and Jan Fostier. Jabba: Hybrid error correction for long sequencing reads using maximal exact matches. In *International Workshop on Algorithms in Bioinformatics*, pages 175–188. Springer, 2015.

[55] Luciano Milanesi and Igor B Rogozin. ESTMAP: A system for expressed sequence tags mapping on genomic sequences. *NanoBioscience, IEEE Transactions on*, 2(2):75–78, 2003.

[56] Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L Madden, Richa Agarwala, and Alejandro A Schäffer. Database indexing for production MegaBLAST searches. *Bioinformatics*, 24(16):1757–1764, 2008.

[57] Aleksandr Morgulis, E Michael Gertz, Alejandro A Schäffer, and Richa Agarwala. A fast and symmetric DUST implementation to mask low-complexity DNA sequences. *Journal of Computational Biology*, 13(5):1028–1040, 2006.

[58] Aleksandr Morgulis, E Michael Gertz, Alejandro A Schäffer, and Richa Agarwala. WindowMasker: Window-based masker for sequenced genomes. *Bioinformatics*, 22(2):134–141, 2006.

[59] Narjes S Movahedi, Elmirasadat Forouzmand, and Hamidreza Chitsaz. De novo co-assembly of bacterial genomes from multiple single cells. In *Bioinformatics and Biomedicine (BIBM), 2012 IEEE International Conference on*, pages 1–5. IEEE, 2012.

[60] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[61] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[62] Gonzalo Navarro, Erkki Sutinen, Jani Tanninen, and Jorma Tarhio. Indexing text with approximate q-grams. In *Combinatorial Pattern Matching*, pages 350–363. Springer, 2000.

[63] Gonzalo Navarro, Erkki Sutinen, and Jorma Tarhio. Indexing text with approximate q-grams. *Journal of Discrete Algorithms*, 3(2):157–175, 2005.

[64] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[65] Pauline C Ng and Steven Henikoff. Predicting deleterious amino acid substitutions. *Genome Research*, 11(5):863–874, 2001.

[66] Zemin Ning, Anthony J Cox, and James C Mullikin. SSAHA: A fast search method for large DNA databases. *Genome Research*, 11(10):1725–1729, 2001.

[67] Panagiotis Papapetrou. *Embedding-based subsequence matching in large sequence databases*. PhD thesis, Citeseer, 2010.

[68] Panagiotis Papapetrou, Vassilis Athitsos, George Kollios, and Dimitrios Gunopulos. Reference-based alignment in large sequence databases. *Proceedings of the VLDB Endowment*, 2(1):205–216, 2009.

[69] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[70] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[71] Pierre Peterlongo and Rayan Chikhi. Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. *BMC Bioinformatics*, 13(1):48, 2012.

[72] Nicolas Philippe, Mikael Salson, Thierry Lecroq, Martine Leonard, Therese Commes, and Eric Rivals. Querying large read collections in main memory: A versatile data structure. *BMC bioinformatics*, 12(1):242, 2011.

[73] Peter Rice, Ian Longden, Alan Bleasby, et al. EMBOSS: The european molecular biology open software suite. *Trends in genetics*, 16(6):276–277, 2000.

[74] Todd Richmond. Gene recognition via spliced alignment. *Genome Biology*, 1(1):reports233, 2000.

[75] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[76] Michael Roberts, Brian R Hunt, James A Yorke, Randall A Bolanos, and Arthur L Delcher. A preprocessor for shotgun assembly of large genomes. *Journal of Computational Biology*, 11(4):734–752, 2004.

[77] Stephen M Rumble, Phil Lacroute, Adrian V Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: Accurate mapping of short color-space reads. *PLOS ONE Computational Biology*, 5(5):e1000386, 2009.

[78] Ravi Sachidanandam, David Weissman, Steven C Schmidt, Jerzy M Kakol, Lincoln D Stein, Gabor Marth, Steve Sherry, James C Mullikin, Beverley J Mortimore, David L Willey, et al. A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. *Nature*, 409(6822):928–933, 2001.

[79] Mark F Schilling. The surprising predictability of long runs. *Mathematics Magazine*, 85(2):141–149, 2012.

[80] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[81] Ilya N Shindyalov and Philip E Bourne. Protein structure alignment by incremental combinatorial extension (ce) of the optimal path. *Protein engineering*, 11(9):739–747, 1998.

[82] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[83] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. ABySS: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[84] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[85] Yanni Sun and Jeremy Buhler. Designing multiple simultaneous seeds for DNA similarity search. *Journal of Computational Biology*, 12(6):847–861, 2005.

[86] Erkki Sutinen and Jorma Tarhio. On using q-gram locations in approximate string matching. *Algorithms – ESA'95*, pages 327–340, 1995.

[87] William R Taylor, Tomas P Flores, and Christine A Orengo. Multiple protein structure alignment. *Protein Science*, 3(10):1858–1870, 1994.

[88] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994.

[89] Niko Välimäki and Eric Rivals. Scalable and versatile k-mer indexing for high-throughput sequencing data. In *Bioinformatics Research and Applications*, pages 237–248. Springer, 2013.

[90] Jayendra Venkateswaran, Deepak Lachwani, Tamer Kahveci, and Christopher Jermaine. Reference-based indexing of sequence databases. In *Proceedings of the 32nd international conference on Very large data bases*, pages 906–917. VLDB Endowment, 2006.

[91] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essaMEM: Finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

[92] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. A long fragment aligner called ALFALFA. *BMC Bioinformatics*, 16(1):159, 2015.

[93] Sebastian Wandelt and Ulf Leser. QGramProjector: Q-gram projection for indexing highly-similar strings. In *Advances in Databases and Information Systems*, pages 260–273. Springer, 2013.

[94] Sebastian Wandelt and Ulf Leser. MRCSI: Compressing and searching string collections with multiple references. *Proceedings of the VLDB Endowment*, 8(5):461–472, 2015.

[95] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. RCSI: Scalable similarity search in thousand(s) of genomes. *Proceedings of the VLDB Endowment*, 6(13):1534–1545, August 2013.

[96] David Weese, Anne-Katrin Emde, Tobias Rausch, Andreas Döring, and Knut Reinert. RazerS: Fast read mapping with sensitivity control. *Genome Research*, 19(9):1646–1654, 2009.

[97] Thomas Wiehe, Steffi Gebauer-Jung, Thomas Mitchell-Olds, and Roderic Guigo. SGP-1: Prediction and validation of homologous genes based on sequence alignments. *Genome Research*, 11(9):1574–1583, 2001.

[98] Derrick E Wood and Steven L Salzberg. Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, 2014.

[99] Thomas D Wu and Colin K Watanabe. GMAP: A genomic mapping and alignment program for mRNA and EST sequences. *Bioinformatics*, 21(9):1859–1875, 2005.

[100] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating read mapping with fasthash. *BMC Genomics*, 14(Suppl 1):S13, 2013.

[101] Chengxi Ye, Zhanshan Sam Ma, Charles H Cannon, Mihai Pop, and W Yu Douglas. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(6):1, 2012.

[102] Daniel R Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

[103] Hongyu Zhang. Alignment of BLAST high-scoring segment pairs based on the longest increasing subsequence algorithm. *Bioinformatics*, 19(11):1391–1396, 2003.

[104] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1-2):203–214, 2000.

[105] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.