Methods for Reducing Unnecessary Computation on False Mappings in Read Mapping

Hongyi Xin

CMU-CS-18-102

February 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CCF-0953246, CCF-1147397, and CNS-1423172, the National Institute of Health under grant numbers R01HG006004 and R01HG007104, the Moore Foundation under grant number 4554, Qualcomm, Strecker, and the Intel ISTC-EC. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Read Mapping, Approximate String Matching, Seed-and-Extend, Landau-Vishkin, Error Tolerance, String Algorithms, Suffix Trie, Edit Distance, Affine Gap

To all sentients.

Abstract

Advancements in sequencing technology have brought a large increase in the quantity of raw sequencing data. To cope with the ever growing sequence data sets, efficient read mappers are developed to accurately reconstruct entire genomes from fragments, by mapping fragments of the genome of an organism to a high quality reference genome. For each fragment, a mapper tries to find a distinct corresponding highly similar sequence in the reference, or multiple highly similar sequences if applicable. Despite improvements of the mapping software, modern state-of-the-art mappers struggle between speed and sensitivity: as a mapper increases sensitivity to tolerate more sequencer errors and genetic variations, it inevitably enlarges the search space and spend more time comparing fragments against increasingly dissimilar reference sequences. Comparisons over such *false mappings* are often unnecessary and wasteful, as dissimilar reference sequences are not considered as meaningful mappings. Furthermore, although only a small fraction of all DNA fragments require high mapping sensitivity due to large number of embedded errors, mappers have to raise sensitivity against all fragments, as the error profile of each fragment is unknown to the mapper.

In this dissertation, we provide multiple algorithms and implementations that aim to reduce the amount of unnecessary computation spent on false mappings while achieving high sensitivity by 1) quickly and accurately **filtering** out false mappings and 2) reducing the total number of false mappings without sacrificing sensitivity through **improved seeding mechanisms**. Specifically, we designed SIMD-friendly algorithms that quickly identify false mappings with high accuracy. We also extended a previously proposed approximate string matching algorithm to better suit biological applications. Finally, we developed multiple methods that enhance the popular seed-and-extend mapping strategy by increasing seed selectivity without reducing mapping sensitivity. From our experiments, we showcase inefficiencies of naïve seeding mechanisms in state-of-the-art mappers. We show that our filters can achieve high filtering accuracies while spending only a fraction of the computational cost. We further show that our improved seed selection methods are highly effective in reducing total number of false mappings. With these algorithms and methods, we provide a set of tools available for future read mappers to be more precise when mapping reads to the reference.

Acknowledgments

People say a Ph.D. is like a roller-coaster. Well I believe that is an overstatement, for roller-coasters at least. A PhD does share a lot in common with a roller-coaster: both have ups and downs and both have people screaming all the time. But a Ph.D. is definitely not a funfair. You hop on to a PhD cart and your next stop can either be Hogwarts or Barad-dur. There is an upside, though: people pay you for the ride.

But that is exactly why PhD is so fascinating. Nothing is set for certain and you have got all the freedom to explore in the vast ocean of knowledge (disclaimer: limited applicability due to funding complications). Everyone lives in their own universe. There is no absolute truth; no universal laws. Everything can be and will be changed. It is a city of skyscrapers built in theorems, based on axioms.

It is definitely not easy to navigate through this unknown wonderland. There are pitfalls everywhere. Miss-oriented and you might never get out of the maze. Sometimes after a long walk you are back at the origin; other times promising paths lead to absolute misery. But there is one thing for certain: there will never be boredom. It feels like being the space marine in DOOM (obviously I am much wimpier and my only weapon is my broken English, which tortures the final bosses as they read through my writings). Everyday is a hell of a challenge but in the end it is so satisfying when you beat the game.

I am lucky that I have gotten through to the final stage and I have been always supported along my journey. I am lucky to have Carl Kingsford as my advisor and I am lucky to have so many people supporting me throughout my adventure in the dungeon and bashing "continue" every time I die.

I would like to first give special thanks to Carl. I still remember vividly of his "String Algorithm" class. It seeds all my later researches. I have always admired the clarity of his explanations and the quickness of his mind. I also want to thank Can Alkan. I would not have survived the transformation from a Computer Architect into a Bioinformatician without his guidance. I would like to thank my previous advisor Onur Mutlu for being rigorous on English writing. I want to thank all of my co-authors, Sunny Nahar, John Emmons, Donghyuk Lee, Gennady Pekhimenko, John Greth and Richard Zhu. I appreciate the help from all my lab mates, Mingfu Shao, Dan Deblasio (and thanks Dan for *adopting* me when I was homeless), Heewook Lee, Guillaume Marçais, Hao Wang, Cong Ma, Brad Soloman and Natalie Sauerwald, as well as my previous lab mates, Kevin Chang, Yoongu Kim, Nandita Vijaykumar, Kevin Hsieh, Rachata Ausavarungnirun, Saugata Ghose, Yixin Luo, Samira Khan, Justin Mesa, Chris Fallin (and many others!). I also want to thank my mentor Bill Bolosky at Microsoft Research for his guidance and many of his wise advices. Additionallly, I want to thank Iman Hajirasouliha for introducing new research opportunities to me.

Finally, I would like to specially thank Lingfei Zhou. Without her support, my PhD would take at least 1.499x longer. Every time that I am deflated, it is she who pumped confidence back into my chest. I appreciate all the hardship she endured as I persuded my Ph.D. I also want to attribute this dissertation to my lovely parents, for their infinite understanding and unconditional support.

Many thanks to NIH for supporting my research (grant HG007104 to professor Carl Kingsford and grant HG006004 to professor Onur Mutlu and professor Can Alkan).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Massively parallel sequencing, or so-called next-generation sequencing (NGS), technologies have substantially changed the way biological research is performed since 2000 [14]. With these new DNA sequencing platforms, we can now investigate human genome diversity between populations [1], find genomic variants that are likely to cause diseases [5, 6, 7, 8, 9, 10], and investigate the genomes of the great ape species [11, 12, 72, 82, 85, 98] and even ancient hominids [33, 77] to understand our own evolution. Despite all the revolutionary power these new sequencing platforms offer, they also present difficult computational challenges due to 1) the massive amount of data produced, 2) shorter read lengths, resulting in more mapping locations and 3) handling sequencing errors as well as genetic variations between individuals.

With NGS platforms, such as the popular Illumina platform [62], the currently dominating technology in the market [31], billions of raw short reads are generated at a fast speed. Each short read represents a contiguous DNA fragment of length 100 base-pairs to 300 base-pairs (bp) from the subject genome. The sequenced read might contain errors. The most common errors are substitutions with an error rate of roughly 0.005-0.010 [86, 87], while there are also occasional insertion and deletions. The error rate increases towards the end of the reads. Multiple methods have been proposed to correct sequencer errors [36, 37, 52, 60].

For many bioinformatics analysis, after the short reads are sequenced, the first step is to *map* the reads to the reference genome. The mapping process is computationally very expensive since the reference genome is large (e.g., the human genome has roughly 3.2 gigabase-pairs). The software that performs read mapping, called the mapper, has to search the entire reference for millions to billions of short reads. To further complicate the matter, the mapper also needs to tolerate sequencer errors, as well as genetic variations. Due to the fact that a mapper by itself cannot distinguish genetic variations from sequencer errors (finding genetic variations is a different problem often called "variation calling"), in this dissertation, we call both sequencer errors and genetic variations together as *errors*. Because reads may contain errors, instead of searching for perfect mappings, mappers perform expensive approximate searching for each read. In addition, the ubiquitous common repeats and segmental duplications within the reference genome [95] further increases the complexity of the task, since a short read from genomic repeats may map to multiple locations in the reference.

Despite numerous improvements over time, modern state-of-the-art read mappers still fall short from attaining both high speed and high sensitivity (defined as the ability to find all valid mappings of a given read, despite embedded errors).

To speed up processing large quantity of reads, modern state-of-the-art mappers employ a strategy called *seed-and-extend*. Seed-and-extend mappers exploit the fact that individual genomes within a species are highly similar due to genomic conservation. Together with the low error rate of modern NGS sequencing technologies, mappers can assume that most reads deviate very little from the reference sequence, if at all. This enables the mapper to focus only on searching mappings where the read and the reference share high similarities (usually set to less then 5% errors between the read and the reference).

Given a read, if there exists any mapping of high similarity (less than 5% errors), then between the read and the corresponding reference sequence, there must be a shared common subsequence of at least $L/(5\% \cdot L)$ base pairs, where L is the length of the read.

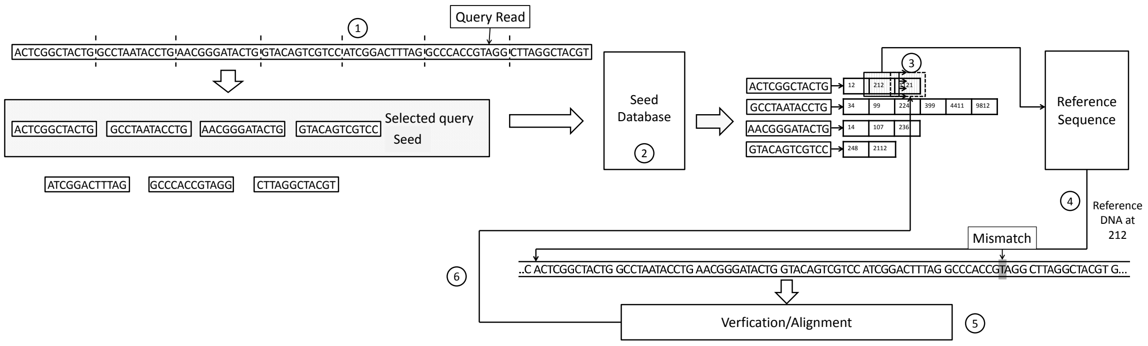
With above observation, from a read, seed-and-extend based mappers extract multiple subsequences called *seeds*, which are subsequently used as keys to access the indexed reference genome. Depending on the design principles, the mapper either iterates through all seed locations or only select a subset of all seed locations. For each selected seed location, the mapper compares the read to the surrounding reference.

The major reason of using the seed-and-extend mapping strategy, compared to the naïve method of scanning through the entire reference, is that seeds drastically reduce the search space. However, to successfully map a read, it requires at least one seed to be free of errors. Since a mapper only fails to map a read when all seeds have errors, the maximum number of errors that a mapper can guarantee to tolerate equals the number of non-overlapping seeds extracted from the read minus one. While a mapper can also draft overlapping seeds (seeds that overlaps in the read), it complicates the calculation of error tolerance. With overlapping seeds, the error guarantee is often calculated using the q-gram theorem [97]. Attaining error tolerance guarantee through the q-gram theorem with overlapping seeds typically involves drafting many more seeds than using non-overlapping seeds, which further slows down the mapping process. Therefore, in this dissertation, we focus on mapping strategies using non-overlapping seeds.

A major dilemma faced by modern mappers, which is also a recurring theme of this thesis, is that mappers need to balance between having more seeds over achieving faster mapping speed. On the one hand, having more seeds increases the error tolerance of a mapper (hence increases the sensitivity of the mapper). On the other hand, using more seeds decreases the average seed length which in turn decreases the selectivity of seeds. This is because as seeds get shorter, they appear more frequently in the reference hence generate more comparisons while mapping a read, albeit most comparisons are false which only yield vastly dissimilar mappings.

1.1 Overview of Seed-and-Extend Based Mappers

Given a reference genome, modern mappers first index it into a reference index database, which can quickly find all occurrences in the reference of any query sequences, or returns NULL when no occurrence is found. Depending on the demand, a reference can be indexed into a suffix array, a suffix tree, a Burrows-Wheeler transformed suffix array, a simple hash table, or more sophisticated hash tables. In general, suffix-array or suffix tree can handle queries of varying sequence length while hash tables can only handle queries of fixed-length sequences.



Query Read
ACTCGGCTACTG GCCTAATACCTG AACGGGATACTG GTACAGTCGTCC ATCGGACTTTAG GCCCACCGTAGG CTTAGGCTACGT
Selected query Seed
Seed Database
Reference Sequence
Reference DNA at 212
Mismatch
..C ACTCGGCTACTG GCCTAATACCTG AACGGGATACTG GTACAGTCGTCC ATCGGACTTTAG GCCCACCGTAGG CTTAGGCTACGT G...
Verfication/Alignment

Figure 1.1: The flow chart of seed-and-extend based mappers.

Figure 1.1 shows the flow chart of a typical seed-and-extend based mapper during the mapping stage. The mapper follows six steps to map a read to the reference. In step 1, the mapper extracts smaller subsequences from the read. Each subsequence has enough length to be used as a key to access the reference index database. In step 2, several of these subsequences are selected as seeds. Seeds are then fed to the reference index database as inputs. The database returns the location lists for each seed. The location list stores all the occurrence locations of each seed in the reference genome. In step 3, the mapper probes the location lists of each seed. For each seed location, as step 4, the mapper retrieves the reference sequence flanking the seed location. In step 5, the mapper coarsely compares the read and the reference around the seed location and rejects the location if the read and the reference are obviously dissimilar. Otherwise, the mapper proceeds to step 6, which aligns the read against the reference sequence using dynamic programming algorithms [75, 92] and measures the edit distance [50], or by more sophisticated metrics [40], between the read and the reference. If the similarity score between the read and the reference exceeds a user-set threshold, then in step 7, the mapper designates the location as a valid mapping. Valid mappings are stored in the output in SAM format, where information of positions and types of differences are described with the so-called Compact Idiosyncratic Gapped Alignment Report string, or the CIGAR string (see example in [57]).

Among the above seven steps, we call step 2 and 3 together as the *seeding stage* while step 6 and 7 as the *filter and verification stage*. Both stages are critical in determining the speed and sensitivity of a read mapper. In general, a mapper is faster when seeds provide fewer seed locations, has a fast and accurate filter and a proficient dynamic programming implementation; and is more sensitive if more non-overlapping seeds are drawn.

1.1.1 Case Studies

A detailed list of mappers, their categorizations and comparisons between them can be found in the literature [18, 55, 78, 79]. Below we provide a few case studies of popular state-of-the-art mappers, focusing on their seeding stage as well as filter and verification stage.

BWA-MEM

BWA-MEM [51] extracts supermaximal exact matches (SMEMs) from reads as seeds. Formally, maximal exact matches (MEMs) are substrings in the read that matches exactly to at least one location in the reference, and cannot be extended in either direction, as any extension will result in zero matching to the reference. A SMEM **X** not only carries all properties of a MEM, but also there does not exist any superstring in the reference, **Y**, of **X** such that **Y** always flanks **X** in the reference: there is at least at one occurrence of **X** where **Y** does not appear.

Since SMEM at times can be as long as more than half of the read itself, using SMEM often leads to a reduced number of seeds hence reduced sensitivity. Therefore, as a remedy, BWA-MEM sub-samples shorter seeds within a long SMEM to increase the seed count. When BWA-MEM subsamples a long seed, it requires that the sub-sampled shorter seeds to appear more frequently than the original SMEM itself. BWA-MEM iteratively sub-samples long seeds until the lengths of the seeds are all below a user-defined threshold, which is 28 base-pairs by default.

After gathering all seeds, BWA-MEM checks if there are locations shared by multiple seeds. BWA-MEM prioritizes verifying locations that have more seeds supporting them. BWA-MEM calls this method “seed chaining”. Finally BWA-MEM uses dynamic programming with an affine-gap penalty score, which penalizes consecutive insertions following a linear equation: $score = p_{open} + (l - 1) \cdot p_{extend}$, where l is the length of the insertion, p_{open} and p_{extend} are two penalty scores.

A major drawback of BWA-MEM is that there is no guarantee of how many seeds will it draw from a read. Hence, it does not provide any guarantee on error tolerance or searching depth on secondary mappings. To make matters even worse, when it sub-samples shorter seeds, it picks a randomized substring in the SMEM. Therefore, BWA-MEM sometimes generates inconsistent mapping results between multiple executions of the same read.

Bowtie2

Bowtie2 [47] extracts a set of fixed number (3 by default), fixed length seeds at fixed positions in each read. Its sort all seeds based on their frequency in the reference, which equals to the lengths of their location lists. Afterwards, bowtie2 randomly extracts a seed location from all seeds, with the probability of drafting from each seed being reversely proportional to the number of remaining locations of the seed. It stops further searching when the mapping cannot be further improved after repeated effort.

Similar to BWA-MEM, Bowtie2 provides no guarantee on its sensitivity. Even though it extracts a fixed number of seeds, which is a small number by default, due to its terminate-if-no-improvement methodology, as well as its probabilistic nature, it could terminate early on a read without exhaustively searching through all seed locations. As a result, not only can bowtie2 generate inconsistent mapping results between multiple runs of the same input, from time to time, it misses high-quality secondary mappings and sometimes even the best mapping, if the best mapping includes many fewer errors than the total number of seeds.

SNAP

SNAP [73] extracts all possible fixed-length seeds from a read, allowing overlapping between seeds. SNAP discards seeds with excessive frequency, since highly likely such seeds belong to genomic repeats. Afterwards it prioritizes seed locations based on the number of seeds supporting each seed location. While similar to the seed chaining mechanism of BWA-MEM, SNAP's seed location prioritization mechanism differs from BWA-MEM in the sense that SNAP also prioritizes locations shared by overlapping seeds. When multiple overlapping seeds supporting a single mapping location, it essentially equals to merging multiple overlapping seeds into a virtual longer seed. Therefore, prioritizing seed locations shared by multiple overlapping seeds as drawing a longer seed. At the same time, SNAP also prioritizes locations that are shared by seeds that are distant in the read. Therefore SNAP balances between using virtual long seeds and seed chaining.

Similar to BWA-MEM and bowtie, by default SNAP prioritizes searching for best mapping over finding all mappings within a user-defined error threshold. As a result, SNAP is very fast but is not as sensitive as fully sensitive mappers (described in the following subsection).

SNAP can also be set to map in fully sensitive mode, which turns SNAP into a fully sensitive mapper. In fully sensitive mapping mode, the speed of SNAP is drastically reduced, as SNAP wastes a lot of execution time on evaluating false mappings (formally defined in the next subsection).

Fully Sensitive Mappers

While none of the above mappers provide sensitivity guarantees, there also exist a number of mappers that do provide sensitivity guarantees, including: [4, 34, 48, 56, 58, 59, 61, 71, 84, 90, 91, 99, 100]. However, due to their sluggish speed [35, 83], in practice, they are less preferable and mostly used as a last resort when faster, less sensitive mappers fail to produce satisfiable results.

A key feature of fully sensitive mappers is to find all mappings within a user defined error threshold, often quantified in edit distances. Even though most downstream analysis only features the best mapping of each read, defined as the most similar mapping in the reference, finding all mappings is crucial in estimating the confidence of the best mapping. If a read has multiple high quality secondary mappings that rivals the best mapping, then the read is highly likely to be part of a genomic repeat. As a result, the mapper should assign a low confidence score to the best mapping as the read might originate else where. Alternatively, even if a read only has a few high quality mappings, it is crucial to find them all as they provide extra information for downstream analysis such as SNP (single nucleotide polymorphism) calling (which finds single base pair genomic variations in a target genome) [89].

To enable finding all mappings within an edit-distance threshold e, fully sensitive mappers rely on comprehensive seeding mechanisms as well as high accuracy verification implementations. Specifically, to tolerate e, errors, fully sensitive mappers first draw $e + 1$ non-overlapping seeds, such that with at most e errors, it is guaranteed at least one seed will be error free. Then they thoroughly go through **all** seed locations and accurately calculate the edit distance between the read and the reference at each seed location.

As we will further elaborate in Chapter 4, comprehensively extracting seeds generates a large number of false mappings, which are seed locations where the reference drastically differs from the read. False mappings lead to high edit distances and are discarded from the mapping results. Hence, computing edit distances for false mappings is a waste of computational resources and is a major reason why fully sensitive mappers are slow.

In this thesis, we focus on computational methods to improve the speed of fully sensitive mappers while keeping their sensitivity guarantees. Specifically, **we aim at reducing the impact of false mappings on the performance of fully sensitive mappers**.

1.2 Existing Seeding Mechanisms

To better interpret why fully sensitive mappers generate a high number of false mappings, it is crucial to understand how fully sensitive mappers select seeds from a read. Below we provide an overview of existing seed selection mechanism, as well as four case studies of existing seed selection mechanisms.

Existing seed selection optimizations can be classified into three categories: (1) extending seed length, (2) avoiding frequent seeds, and (3) rebalancing frequencies among seeds. Optimizations in the first category extend frequent seeds in order to reduce their frequencies. Optimizations in the second category sample seed positions in the read and reject positions that generate frequent seeds. Optimizations in the third category rebalance frequencies among seeds such that the average seed frequency at runtime is more consistent with the static average seed frequency of the seed table.

We selectively pick four representative state-of-the-art seed selection mechanisms from the above three categories for case studies. They are: *Cheap K-mer Selection (CKS)* in FastHASH [102], *Optimal Prefix Selection (OPS)* in Hobbes [3], *Adaptive Seed Filter (ASF)* in the GEM mapper [71] and *spaced seeds* in PatternHunter [67]. Among the four prior works, ASF represents works from the first category; CKS and OPS represent works from the second category and spaced seeds represents works from the third category.

The **Adaptive Seed Filter (ASF)** [71] seeks to reduce the frequency of seeds by extending the lengths of the seeds. For a read, ASF starts the first seed at the very beginning of the read and keeps extending the seed until the seed frequency is below a pre-determined threshold, t. For each subsequent seed, ASF starts it from where the previous seed left off in the read, and repeats the extension process until the last seed is found. In this way, ASF aims to guarantee that all seeds have a frequency below t.

ASF has two major drawbacks. First, ASF assumes the least frequent **set of seeds** in a read has similar frequencies; hence, it shares a common frequency threshold t. We observe that this is not always true. The optimal set of seeds often have very different frequencies. This is because some seeds do not provide much frequency reduction despite long extensions while other seeds yield significant frequency reductions only at certain extension lengths (the frequency reduction due to extension looks like a step function). By regulating all seeds with the same frequency threshold, ASF inefficiently distributes base-pairs among seeds. Second, the fact that ASF sets a fixed frequency threshold t for **all reads** often leads to under-utilization of base-pairs in reads. Different reads usually have different optimal thresholds (the threshold that provides the least frequent set of seeds under ASF for the read). For reads that contain frequent seeds, optimal thresholds are usually large (e.g., $t > 1000$), while for reads without frequent seeds, optimal thresholds are usually small (e.g., $t < 100$). Unfortunately, ASF can apply only a single threshold to *all* reads. If t is set to a large value to accommodate reads with frequent seeds, then for other reads, ASF extracts only short seeds even if there are many unused base-pairs. Otherwise, if t is set to a small value, then frequent seeds consume many base-pairs and reads with frequent seeds have insufficient base-pairs to construct enough seeds to tolerate all errors.

Note that the method of selecting seeds consecutively starting at the beginning of a read does not always produce infrequent seeds. Although most seeds that are longer than 20-bp are either unique or non-existent in the reference, there are a few seeds that are still more frequent than 100 occurrences even at 40-bp (e.g., all "A"s). With a small t ($t \leqslant 50$), ASF cannot guarantee that there will be enough seeds from the read to meet the error tolerance guarantee. This is because some seeds still have greater-than-t frequencies even with large seed lengths (such as poly-A sequences). In such cases, forcing ASF to always draw seeds with frequencies below t produces long seeds that occupy too many base pairs from the read, leaving insufficient base pairs to construct other seeds.

Setting a static t for all reads further worsens the problem. Reads are drastically different. Some reads do not include any frequent short patterns (e.g., 10-bp patterns) while other reads have one to many highly frequent short patterns. Reads without frequent short patterns do not produce frequent seeds in ASF, unless t is set to be very large (e.g., $\geqslant 10,000$) and as a result the selected seeds are very short (e.g., $\leqslant$ 8-bp). Reads with many frequent short patterns have a high possibility of producing longer seeds under medium-sized or small t's (e.g., $\leqslant 100$). For a batch of reads, if the global t is set to a small number, reads with many frequent short patterns will have a high chance of producing many long seeds that the read does not have enough length to support. If t is set to a large number, reads without any frequent short patterns will produce many short but still frequent seeds as ASF will stop extending a seed as soon as it is less frequent than t, even though the read could have had longer and less frequent seeds.

Cheap K-mer Selection (CKS) [102] aims to reduce seed frequencies by selecting seeds from a wider potential seed pool. For a fixed seed length k, CKS samples $\lfloor \frac{L}{k} \rfloor$ seed positions consecutively in a read, with each position set apart from another by k-bp. Among the $\lfloor \frac{L}{k} \rfloor$ positions, it selects x seed positions that yield the least frequent seeds (assuming the mapper needs x seeds). In this way, it avoids using positions that generate frequent seeds.

CKS has low overhead. In total, CKS needs only $\lfloor \frac{L}{k} \rfloor$ lookups for seed frequencies followed by a sorting of $\lfloor \frac{L}{k} \rfloor$ seed frequencies. Although fast, CKS can provide only limited seed frequency reduction as it has a very limited pool to select seeds from. For instance, in a common mapping setting where the read length L is 100-bp and seed length k is 12, the read can be divided into at most $\lfloor \frac{100}{12} \rfloor = 8$ positions. With only 8 potential positions to select from, CKS is forced to gradually select more frequent seeds under greater seed demands. To tolerate 5 errors in this read, CKS has to select 6 seeds out of 8 potential seed positions. This implies that CKS will select the 3rd most frequent seed out of 8 potential seeds. For human genome, 12-bp seeds on average have a frequency over 172, and selecting the 3rd frequent position out of 8 potential seeds renders a high possibility of selecting a frequent seed which has a higher frequency than average.

Similar to CKS, **Optimal Prefix Selection (OPS)** [3] also uses fixed length seeds. However, it allows a greater freedom of choosing seed positions. Unlike CKS, which only select seeds at positions that are multiples of the seed length k, OPS allows seeds to be selected from any position in the read, as long as seeds do not overlap.

The basis of OPS is also a dynamic programming algorithm that implements a recurrence function. OPS iteratively finds the optimal set of fixed-length seeds (seeds with minimum total frequency) of all profixes of a read, starting from a single seed to the total number of required seeds.

Compared to CKS, OPS is more complex and requires more seed frequency lookups. In return, OPS finds less frequent seeds, especially under large seed numbers. However, with a fixed seed length, OPS cannot find the optimal non-overlapping variable-length seeds.

Spaced seeds [67] aims to rebalance frequencies among patterns in the seed database [17]. Rebalancing seeds reduces the frequent seed phenomenon which, in turn, reduces the average seed frequency in read mapping (in other words, it improves the sensitivity/selectivity ratio of seeds in read mapping [27]). Spaced seeds rebalance seeds by using a hash function that is guided by a user-defined bit-mask, which combines frequent and infrequent long seeds together and merges them into a single spaced short seed. Different patterns that are hashed into the same hash value are considered as a single "spaced seed". By carefully designing the hashing function, which extracts base-pairs only at selected positions from a longer (e.g., 18-bp) pattern, spaced seeds can group frequent long patterns with infrequent long patterns and merge them into the new and more balanced "spaced seeds", which have smaller frequency variations. At runtime, long raw seeds are selected consecutively in the reads, which are processed by the rebalancing hash function later.

Many extensions of spaced seeds were proposed, including: vector seeds [13], indel seeds [68], subset seeds [81], neighbor seeds [21, 22] or adaptive seeds [43]. A detailed list and description can be found in this literature [76].

Spaced seeds has two disadvantages. First, the hash function cannot perfectly balance frequencies among all "rebalanced seeds". After rebalancing, there is still a large disparity in seed frequency amongst seeds. Second, seed placement in spaced seeds is static, and does not accommodate for high-frequency seeds. Therefore, positions that generate frequent seeds are not avoided, and they still give rise to the frequent seeds phenomenon.

Table 1.1 summarizes the complexity and memory traffic of each seed selection optimization.

	ASF	CKS	OPS	Spaced seeds	naïve
Empirical average-case complexity	$\mathcal{O}(x)$	$\mathcal{O}(x \times log(\frac{L}{k}))$	$\mathcal{O}(x \times L)$	$\mathcal{O}(x)$	$\mathcal{O}(x)$
Number of lookups	$\mathcal{O}(x)$	$\mathcal{O}(\frac{L}{k})$	$\mathcal{O}(L)$	$\mathcal{O}(x)$	$\mathcal{O}(x)$

Table 1.1: An average-case complexity and memory traffic comparison (measured by the number of seed-frequency lookups) of seed selection optimizations, including Adaptive Seed Filter (ASF), Cheap K-mer Selection (CKS), Optimal Prefix Selection (OPS), spaced seeds and naïve (selecting fixed-length seeds consecutively).

1.3 Existing Verification Implementations

The original method for verification fully relied on dynamic programming. These methods find the optimal alignment between two sequences that maximizes the similarity score with regard to specific sequence alignment penalty scoring schemes. The Needleman-Wuncsch algorithm [75] solves the problem of global alignment while the Smith-Waterman algorithm [92] finds the optimal local alignment and the Gotoh algorithm [32] finds the optimal alignment with the affine gap penalties. All three algorithms are often referred as Smith-Waterman-Gotoh algorithms. In general, the time and space complexity of Smith-Waterman-Gotoh algorithms is $O(m \cdot n)$, where m and n are length of the two input sequences.

A later variant of the Smith-Waterman-Gotoh algorithms took advantage of the fact that read mappers only care about mappings with few differences and uses a restriction of the dynamic programming to a strip of width k around its diagonal in the dynamic programming matrix. Then the space and memory complexity reduces to $O(m \cdot k)$. The resulting algorithms, are called Striped-Smith-Waterman-Gotoh algorithms [28].

Many mechanisms have been proposed to further speed up the verification process. These mechanisms can be divided into five main classes: (i) SIMD implementations of dynamic programming (or simply DP) algorithms, (ii) bit-vector implementations of DP algorithms, and (iii) locality-based filtering mechanisms. Notice that although mechanisms in both (i) and (ii) are often just different implementations of the same basic dynamic programming algorithm, we separate them into two categories because they employ different optimization strategies: while mechanisms in (i) faithfully implement the DP algorithm in a SIMD fashion, mechanisms in (ii) use a modified bit-parallel algorithm to calculate a bit representation of the DP matrix [74].

In the classical Smith-Waterman-Gotoh approach, to map a read of length l to a reference of length l, a $(l+1) \times (l+1)$ matrix is generated. Given the read and the reference strings S and T, element (x, y) is defined as the dissimilarity score (assuming we are using Needleman-Wuncsch algorithm) between string $S[1..x]$ and $T[1..y]$. In cases where x or y equals to 0, the value simply measures the score of comparing against an empty string.

For popular penalty schemes such as edit-distance or affine-gapping panelties, the DP matrix can be iteratively filled following a top-to-bottom, left-to-right manner. For example, for edit-distance, the first row (0th row) and the first column (0th column) of the matrix are initialized to the sequences $0, 1, 2, \ldots, l$ and $0, 1, 2, \ldots, l$ respectively. The value of each element is dependent on its top neighbor, its left neighbor, its top-left neighbor, and the comparison of basepairs from the read and the reference. This relationship is formally described by Equation 1.1:

$$m_{i,j} = \min \begin{cases} m_{i-1,j} + 1, \\ m_{i,j-1} + 1, \\ \begin{cases} m_{i-1,j-1} & \text{read[i] = reference[j]} \\ m_{i-1,j-1} + 1 & \text{otherwise} \end{cases} \end{cases} \tag{1.1}$$

The complexity of the Smith-Waterman-Gotoh algorithm is $O(l^2)$. However, in read mapping, computing the precise edit-distance between the read and the reference sequences would be wasteful, as it is only necessary to determine if the two sequences differ by more than e errors. Ukkonen's algorithm [96], a Striped-Smith-Waterman-Gotoh algorithm, takes advantage of this fact and only determines if two sequences differ by more than the error threshold by calculating a strip of $2e+1$ diagonal lanes of the matrix, rather than the entire matrix as in the Smith-Waterman approach. The time complexity of Ukkonen's improved DP algorithm is $O((2e+1) \times l)$.

Several **bit-vector algorithms** [39, 74] that exploit bit-parallelism in DP algorithms have been proposed. By making the observation that elements in the edit-distance matrix of DP algorithms differ from their top and left neighbors by at most 1 (± 1), the edit-distance matrix can be transformed into two series of bit-vectors. We choose Gene Myers' bit-vector implementation [74] as a representative algorithm. In Gene Myers' algorithm, bit-vectors record differences between consecutive columns of the edit-distance matrix. Because each element can either be $+1$, -1, or 0 away from its left neighbor, two separate bit-vectors are used for each column: one indicates rows where the elements are $+1$ away from their left neighbors while the other indicates rows where the elements are -1 from their left neighbors. Gene Myers further proves that the differences between each pair of cells in two consecutive columns can be computed in parallel. A *minimum edit-distance score* for each column is computed as part of this process.

Although the complexity of the algorithm is $O(l^2)$, the runtime of Gene Myers' algorithm is much faster than basic DP algorithms. Each bit-vector is mapped to a few computer registers; therefore, applying an operation to the register is equivalent to applying the same operation to many elements in the edit-distance matrix. If each register has w bits, theoretically Gene Myers' algorithm can provide a $(w/|S|)\times$ speedup over the basic Smith-Waterman DP algorithm, where S is the cardinality of some set of symbols ($S := |\{\text{A, C, T, G}\}| = 4$ in DNA). Nonetheless, even if the registers are wide enough to store an entire bit-vector ($w \geqslant l+1$), Gene Myers' algorithm still requires $O(l)$ bit-wise operations to cascade the computation for l total bit-vectors.

Another approach to speed up the basic DP algorithm is to efficiently map the DP algorithm to **SIMD units** [23, 26, 28, 38, 88, 93, 94, 105]. Many modern computers have SIMD units, such as GPUs and vector units in CPUs. These vector units pack multiple data elements into a single, wide register and apply the same instruction to all of the packed data elements simultaneously. SIMD implementations of DP algorithms, such as swps3, exploit the data parallelism between elements in the edit-distance matrix. In swps3, elements in the edit-distance matrix are mapped to SIMD registers in a striped manner. Data is placed such that within a single register there are no dependencies bewteen elements. Also, elements within a register share dependencies on other registers. Therefore, elements within a SIMD register are synchronized for SIMD operations (either all of them are ready or none of them are ready). There exist also a number of implementations for special computational infrastructure, namely for GPUs and Xeon Phis [24, 44, 63, 64, 65, 66, 69].

Similar to bit-vector implementations of DP algorithms, SIMD implementations do not reduce the complexity of the algorithm, but speed up the process by exploiting parallelism within the algorithm. Theoretically, a SIMD platform that packs p elements into a single register, can provide up to $p\times$ speedup over basic Smith-Waterman implementation. In practice, however, due to extra computation spent on data mapping and other auxiliary processing, the speedup of SIMD implementations is generally smaller than p. For example, swps3 uses Intel SSE, which packs 16 elements into a single register, while providing a maximum speedup of only $8\times$.

It is not always necessary to calculate the edit-distance between two strings to verify a potential mapping. Incorrect mappings can also be filtered out with simple **locality based searches**. Several works [3, 102] have shown that the potential mappings of a read can be verified by checking (searching) relative distances between its seeds (small subsequences of the read that are used to generate potential mappings). We select Adjacency Filtering (AF) from FastHASH [102] as a representative example. In AF, if a read can be divided into m non-overlapping seeds, then with e allowed errors, at least $m - e$ seeds should be located near the mapping site in the reference for correct mapping. For example, a potential mapping location, loc, is valid for the ith seed of the read if there exists a location within the expected range of $[loc + i \cdot k - e, loc + i \cdot k + e]$ in the location list of the seed (there is a range of $\pm e$ because of possible insertions/deletions from other seeds). A potential mapping location passes AF only if more than $m - e$ seeds pass the locality check.

Filters that are similar to AF work well only when e is very small (e.g., below $1/3$rd of m). When e gets larger, their accuracy decreases drastically. There are two reasons for this. First, with a larger e, the range of expected locations ($[loc + i \cdot k - e, loc + i \cdot k + e]$) expands, which causes more potential locations to pass locality checking. Second, the required minimum number of seeds that exhibit locality around loc, $m - e$, is largely reduced, since m is typically a smaller number of 10 or less. As a result, filtering mechanisms exploiting locality among seeds are not favorable for large e (e.g., $e \geqslant 3$).

It is worth mentioning that although BWA-MEM's seed chaining and SNAP's prioritization of seed locations shared with multiple seeds are not designed for mappers with error tolerance guarantees, they employ the same key insight with seed-locality based filters.

1.4 Overview

1.4.1 Thesis Statement

Within this thesis, I investigate the following question: can we drastically reduce the unnecessary computation spent on verifying false mappings, without giving up the error tolerance guarantee? Is it possible to maintain the same error tolerance guarantee while encountering significantly fewer false mappings? In sight of the above questions, my thesis statement can be summarized as follows:

With novel computational techniques, we can significantly mitigate the amount of unnecessary computation spent on false mappings, without surrendering the error tolerance guarantee or reducing the sensitivity of fully sensitive read mappers. Mitigating unnecessary computation on false mappings is critical in improving the mapping speed of fully sensitive read mappers, as false mappings consume a large amount of execution time while providing no contribution to the mapping result.

1.4.2 Major Contributions

We propose four major computational techniques that are split into two categories: **false mapping filters** and **improved seeding mechanisms**. Among the two, false mapping filters focuses on rejecting false mappings early without spending much computation while improved seeding mechanisms aim at reducing overall seed locations without undermining mapping sensitivity.

- **Shifted Hamming Distance**. Shifted Hamming Distance (SHD) is a SIMD-friendly filter that accelerates the alignment verification procedure in read mapping, by quickly filtering out error-abundant sequence pairs using bit-parallel and SIMD-parallel operations. SHD only filters string pairs that contain more errors than a user-defined threshold, making it fully comprehensive. It also maintains high accuracy with moderate error threshold (up to 5% of the string length) while achieving a 3-fold speedup over the best previous algorithm (Gene Myers's bit-vector algorithm). SHD is published in Bioinformatics [101].
- **LEAP**. LEAP is a verification algorithm that extends the well known Landau-Viskin algorithm to support any monotonic-penalty scoring scheme. Most importantly, LEAP supports affine gap penalties, which is a widely used penalty scheme for modern read mappers. We show that LEAP is up to 7.4x faster than the state-of-the-art bit-vector edit-distance implementation. LEAP is published on bioRxiv [104].
- **Optimal Seeds Solver**. Optimal Seeds Solver (OSS) is a dynamic programming algorithm that discovers the least frequently-occurring set of x seeds in an L-base-pair read in $\mathcal{O}(x \times L)$ operations on average and in $\mathcal{O}(x \times L^2)$ operations in the worst case, while generating a maximum of $\mathcal{O}(L^2)$ seed frequency database lookups. We compare OSS against four state-of-the-art seed selection schemes and observe that OSS provides a 3-fold reduction in average seed frequency over the best previous seed selection optimizations. OSS is published in Bioinformatics [103]
- **Error Resilient Seeds**. Error Resilient Seeds (ERS) is a new seeding concept that allows seed-and-extend based mappers to achieve a higher error tolerance that often exceeds what traditional seeds can offer, without incurring any extra computation. In other words, Error resilient seeds can provide the same error tolerance with fewer seeds, hence reducing the over all seed locations. We implemented Error Resilient Seeds and showed that it provided 1.17-fold reduction in average compared to OSS. The work of ERS is on-going and it is yet to be published.

We show that each of our proposed computational techniques successfully reduces the impact of false mappings on the performance of fully sensitive mappers. We show that SHD and LEAP improve the speed of the verification process, specially for false mappings; and OSS and ERS reduce the overall seed locations without compromising the error tolerance guarantee of a fully sensitive mapper. Overall, this dissertation provides a foundation in developing future fully sensitive mappers.

Part I

False Mapping Filters

Chapter 2

Shifted Hamming Distance: SIMD-Friendly Filter to Accelerate Read Mapping

2.1 Background

As a mapper increases its mapping sensitivity in hope of recovering more reads from errors, it inevitably decreases the seed length and reduce the selectivity of seeds. As a result, fully sensitive mappers often include many seed locations that lead to drastically dissimilar false mappings. Evaluating false mappings is wasteful as they are not reported as valid mappings of the read. In fact, from our profiling, we found as much as 98% of all seed locations provide false mappings.

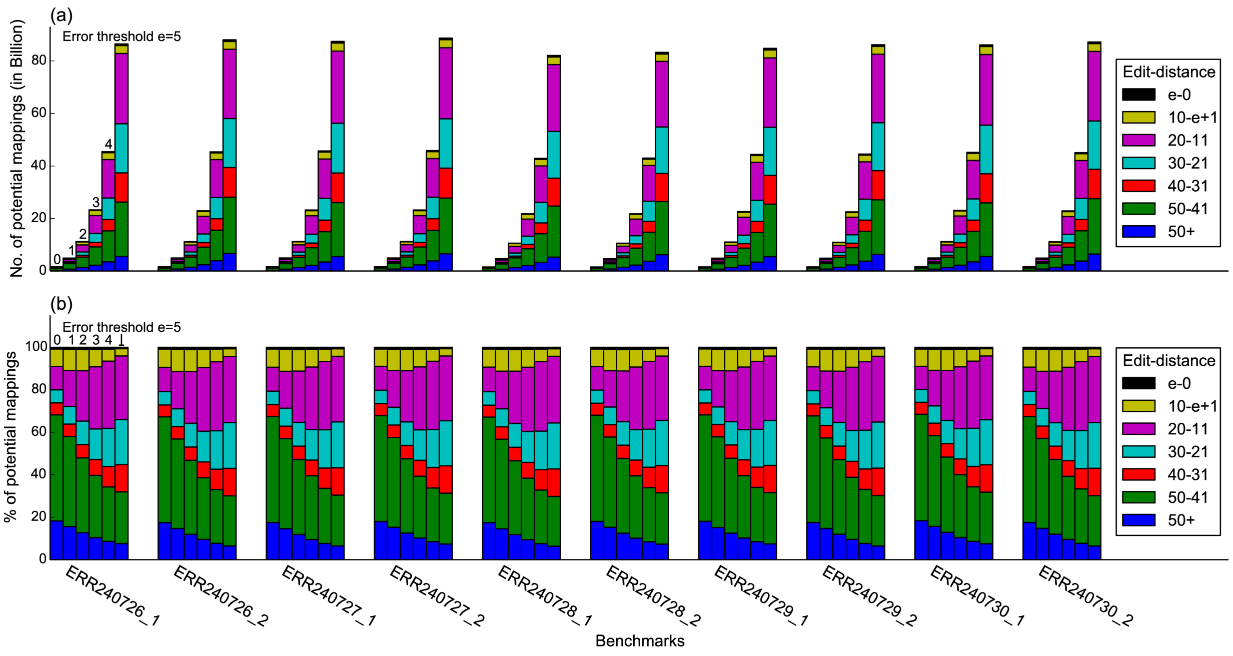


Figure 2.1: Edit-distance edit-distance distribution of all seed locations generated by mrFAST.

In Figure 2.1, we computed the exact edit-distance between 100 basepair long reads (reads are real data from the 1000 Genomes project) and their corresponding potential locations in the Human reference genome as determined by a seed-and-extend based mapper with an error tolerance up to 5% read length, mrFAST [4], measured by edit-distance e between the read and the reference. For $e = 0, 1, \ldots, 5$, more than 98% of the potential locations did *not* meet the error threshold. Moreover, over half of all potential locations contained more than 20 errors. Seed-and-extend based mappers that do not filter these clearly invalid mappings waste much of their time performing computationally expensive edit-distances calculations.

One popular measure to speed up the mapping process, is to deploy fast and accurate filters, which detect and reject incorrect mappings using cheap heuristics. This can increase the speed of seed-and-extend mappers (by speeding up the verification procedure) [102] while maintaining their high accuracy and comprehensiveness. An ideal filter should be able to quickly verify the correctness of a mapping, yet require much less computation than rigorous local alignment, which precisely calculates the number of errors between the read and reference using dynamic programming methods. More importantly, a filter should never falsely remove a correct mapping from consideration, as this would reduce the comprehensiveness of the mapper.

Recent work has shown the potential of using single instruction multiple data (SIMD) vector execution units including general-purpose GPUs and Intel SSE [41] to accelerate local alignment techniques [28, 69, 94]. However, these publications only apply SIMD units to existing scalar algorithms, which do not exploit the massive bit-parallelism provided by SIMD platforms.

In this chapter, we present Shifted Hamming Distance (SHD), a fast and accurate SIMD-friendly bit-vector filter to accelerate the local alignment (verification) procedure in read mapping. The key idea of SHD is to avoid wasting computational resources on incorrect mappings by verifying them with a cheap, SIMD-friendly filter before invoking canonical complex local alignment methods. Our studies show that SHD quickly identifies the majority of the incorrect mappings, especially ones that contain far more errors than allowed, while permitting only a small fraction of incorrect mappings to pass SHD which are later filtered out by more sophisticated and accurate filters or by local alignment techniques.

To evaluate SHD, we choose three representative implementations: swps3 [94], SeqAn [26], and FastHASH [102], representing SIMD DP implementations, bit-parallel DP implementations as well as seed locality based filters as we mentioned in Section 1.3. These mechanisms were not designed as both SIMD and bit-parallel filters and are either fast or accurate (can filter out most incorrect mappings) but not both. On the other hand, SHD leverages both bit-parallelism and SIMD instructions to achieve high performance while preserving high accuracy.

2.2 Contributions

SHD provides the following contributions:

- We show that for seed-and-extend based mappers, most potential mappings contain far more errors than what is typically allowed.
- We introduce a fast and accurate SIMD-friendly bit-vector filter, SHD, which approximately verifies a potential mapping with a small set of SIMD-friendly operations (Section 2.3).

- We prove that SHD never removes correct mappings from consideration; hence, SHD never reduces the accuracy or the comprehensiveness of a mapper (Section 2.3).
- We provide an implementation of SHD with Intel SSE and compare it against three previously proposed filtering and local alignment implementations (Section 2.4), including an SSE implementation of the Smith-Waterman algorithm, swps3 [94]; an implementation of Gene Myers's bit-vector algorithm, SeqAn [26]; and an implementation of our Adjacency Filtering algorithm, FastHASH [102]. Our results on a wide variety of real read sets show that SHD SSE is both fast and accurate. SHD SSE provides up to 3x speedup against the best previous state-of-the-art edit-distance implementation [26] with a maximum false positive rate of 7% (the rate of incorrect mappings passing SHD).

2.3 Methods

Overview: Our filtering algorithm, Shifted Hamming Distance (SHD), uses simple bit-parallel operations (e.g. AND, XOR, etc.) which can be performed quickly and efficiently using the SIMD architectures of modern CPUs. SHD relies on two key observations:

1. If two strings differ by e errors, then all non-erroneous characters of the strings can be aligned in at most e shifts.
2. If two strings differ by e errors, then they share at most $e+1$ identical sections (Pigeonhole Principle [102]).

Based on the above observations, SHD filters potential mappings in two steps:

1. Identify basepairs (bps) in the read and the reference that can be aligned by incrementally shifting the read against the reference.
2. Remove short stretches of matches identified in step 1 (likely noise).

We call these two steps *shifted Hamming mask-set* (SHM) and *speculative removal of short-matches* (SRS) respectively. In the remainder of this section, we describe these two steps, then analyze SHD in terms of false negatives[1] and false positives[2].

2.3.1 Shifted Hamming Mask-Set (SHM)

Shifted Hamming mask-set (SHM) aligns basepairs in the read and the reference by horizontally shifting the read against the reference. SHM is based on the key observation that *if there are no more than e errors between the read and the reference, then each non-erroneous basepair (bp) in the reference can be matched to a basepair in the read within $[-e, +e]$ shifts from its position.* Thus, if there are more than e basepairs in the read that failed to find a match in the reference, then there must be more than e errors between the read and the reference, hence the potential mapping should be rejected.

[1]Defined as correct mappings that are falsely rejected by SHD. Later we show that the false negative rate of SHD is zero.

[2]Defined as incorrect mappings that pass SHD.

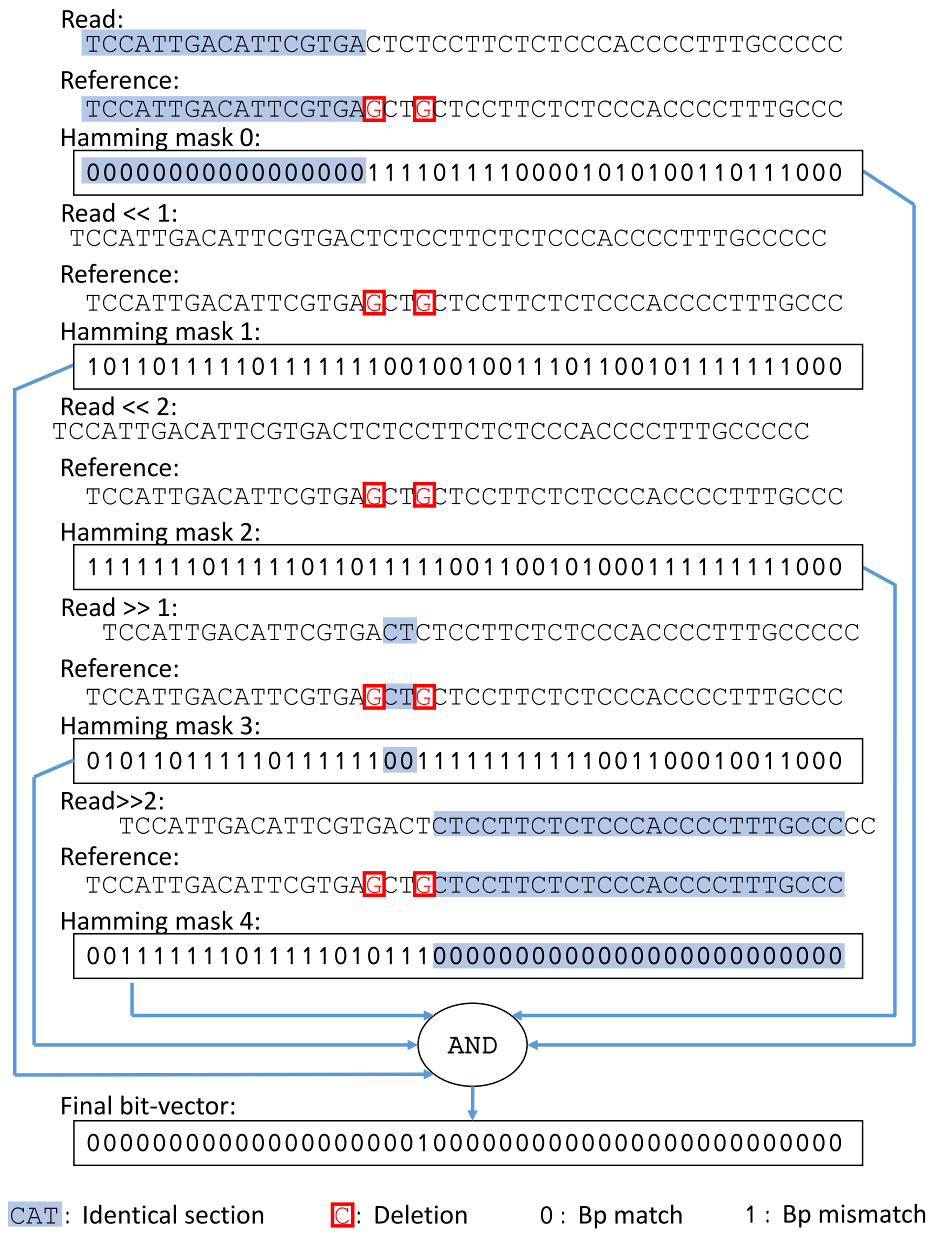


Figure 2.2: An example of applying SHM to a correct mapping with two deletions.

Based on this observation, SHM verifies a potential mapping in two steps. First, SHM separately identifies all basepair matches by **calculating a set of *2e + 1* Hamming masks while incrementally shifting the read against the reference** (one Hamming mask per shift). Each Hamming mask is a bit-vector of ‘0’s and ‘1’s representing the comparison of the read and the reference, where a ‘0’ represents a bp match and a ‘1’ represents a bp mismatch (implementation details of computing Hamming masks using bit-parallel operations are provided in later sections). Figure 2.2 illustrates the production of these Hamming masks for a correct mapping. Once found, SHM **merges all basepair matches together** through multiple bit-wise AND operations.

In SHM, to tolerate e errors, $2e + 1$ Hamming masks must be produced where: e Hamming masks are calculated after incrementally shifting the read to the left by 1 to e bps; e Hamming masks are calculated by incrementally shifting the read to the right by 1 to e bps; one additional Hamming mask is calculated without any shifting. By incrementally shifting the read in SHM, all basepairs between the read and the reference of a correct mapping (except the errors) are brought into alignment with at least one matching bp of the read and identified in one or more of the $2e + 1$ masks, as shown in Figure 2.2.

The Hamming masks are merged together in $2e$ bit-wise AND operations. When ANDing Hamming masks, a '0' at any position will lead to a '0' in the resulting bit-vector at the same position. When aligned with a match, a bp produces a '0' in the Hamming mask, which masks out all '1's in any other Hamming masks at the same position. Therefore, the final bit-vector produced after all bit-wise AND operations are complete is guaranteed to contain '0's for all non-error basepairs; as a result, the number of '1's that remain in the final bit-vector provides a lower bound on the edit-distance between the read and the reference. Since correct potential mappings must have e or fewer errors, SHM can safely filter mappings whose final bit-vector contains more than e '1's, without any risk of removing correct read mappings.

It is worth noting that we assumed the bit-wise hamming mask can be obtain by XORing a byte-wise ASCII string pair. Such operation can be implemented by first converting the byte-wise ASCII string into $log_2\Sigma$ (Σ is the total size of the alphabet) bit-wise vectors, where the ith vector stores the ith bits of the original ASCII byte string. Assume we have two strings A and B, while A_i is the ith bit-wise vector of string A and B_i is the ith bit-wise vector of string B, then the hamming mask of A and B by ORing all hamming masks of A_i and B_i, for all i from 0 to $log_2\Sigma - 1$.

2.3.2 Speculative Removal of Short-Matches (SRS)

SHM ensures all correct read mappings are preserved; however, many incorrect mappings may also pass the filter as false positives. For example, the read in Figure 2.3 is compared against a drastically different reference using SHM with an error threshold of two ($e = 2$). Despite the presence of substantially more than two errors, the final bit-vector produced by SHM does not contain any '1's, as if there were no errors at all. In SHM, '0's in the final bit-vector are considered to be matches and '1's are considered to be errors. In this example, most basepairs in the reference find a match within two shifts of the read, so the read and the reference are considered similar enough to pass the filter.

The false positive rate of SHM increases superlinearly as e increases. Consider a random read and the reference pair, where each basepair in the read and reference are generated completely randomly (having $1/4$ probability of being either A, C, G or T). The probability that a bp in the reference does not match any neighboring bp in the read during any of the $2e + 1$ Hamming masks of SHM (hence rendering a '0' at its position in the final bit-vector) is $(3/4)^{2e+1}$, which decreases exponentially as e increases. Therefore, when e is large, most basepairs in the reference find matches in the read during SHM, even if the read and the reference differ by more than e errors.

Some of the incorrect mappings that pass SHM can still be identified by checking if the read and the reference share large sections of identical substrings. According to our second

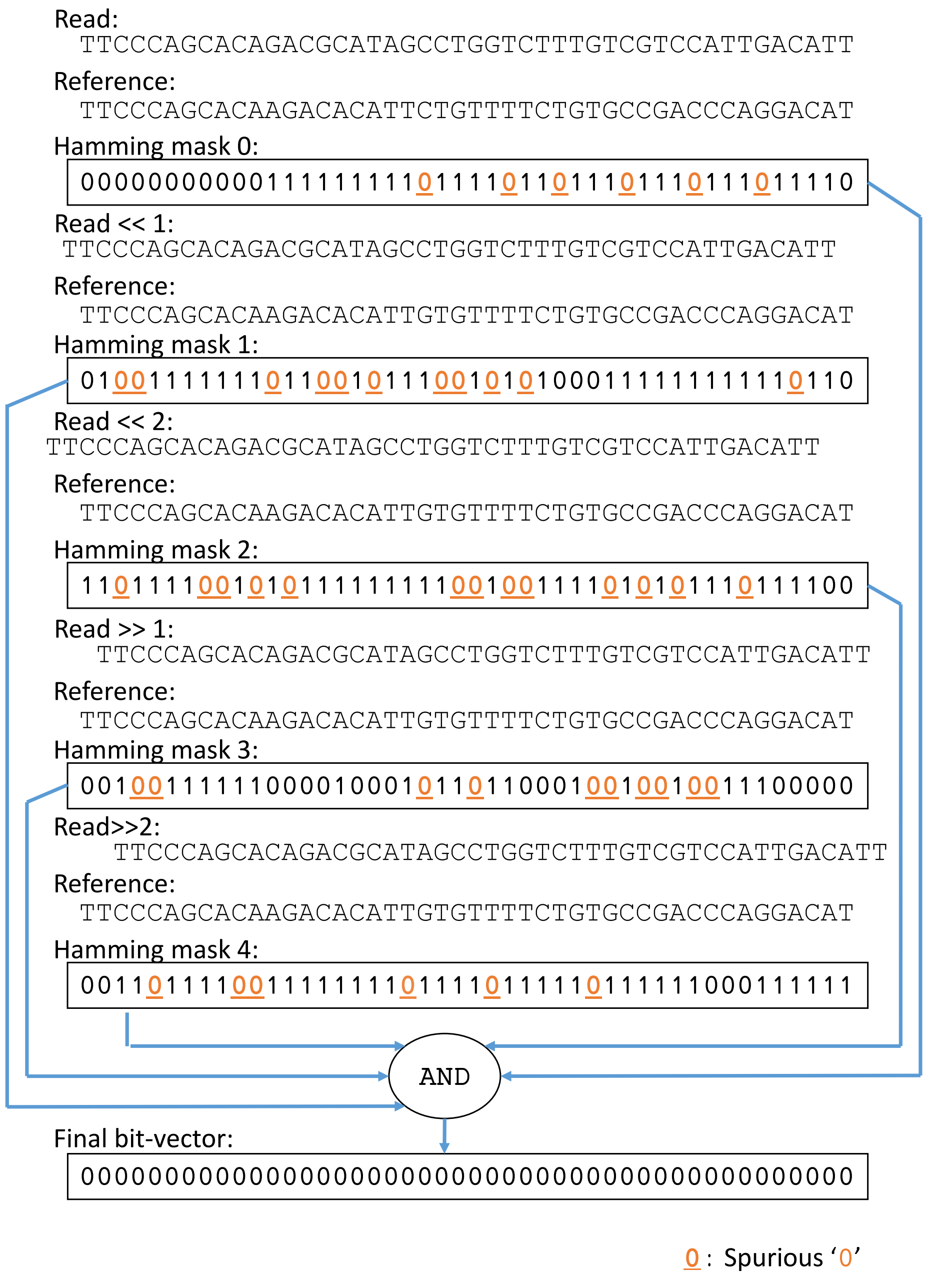


Figure 2.3: An example of an incorrect mapping that passes SHM.

observation, two strings that differ by e errors will share no more than $e + 1$ identical sections. These *identical sections* are simply the bp segments between errors. In fact, the goal of the entire local alignment (edit-distance) computation is to identify these identical sections and the errors between them. When basepairs of an identical section are aligned in SHM, all basepairs of this identical section in the read *simultaneously* match all basepairs in the reference, which **produces a *contiguous streak* of '0's in the Hamming mask** (blue-highlighted region in Figure 2.2). Other '0's in the Hamming masks (unhighlighted '0's in the Hamming masks) that are not produced by an identical section represent only individual bp matches, which are not part of the correct

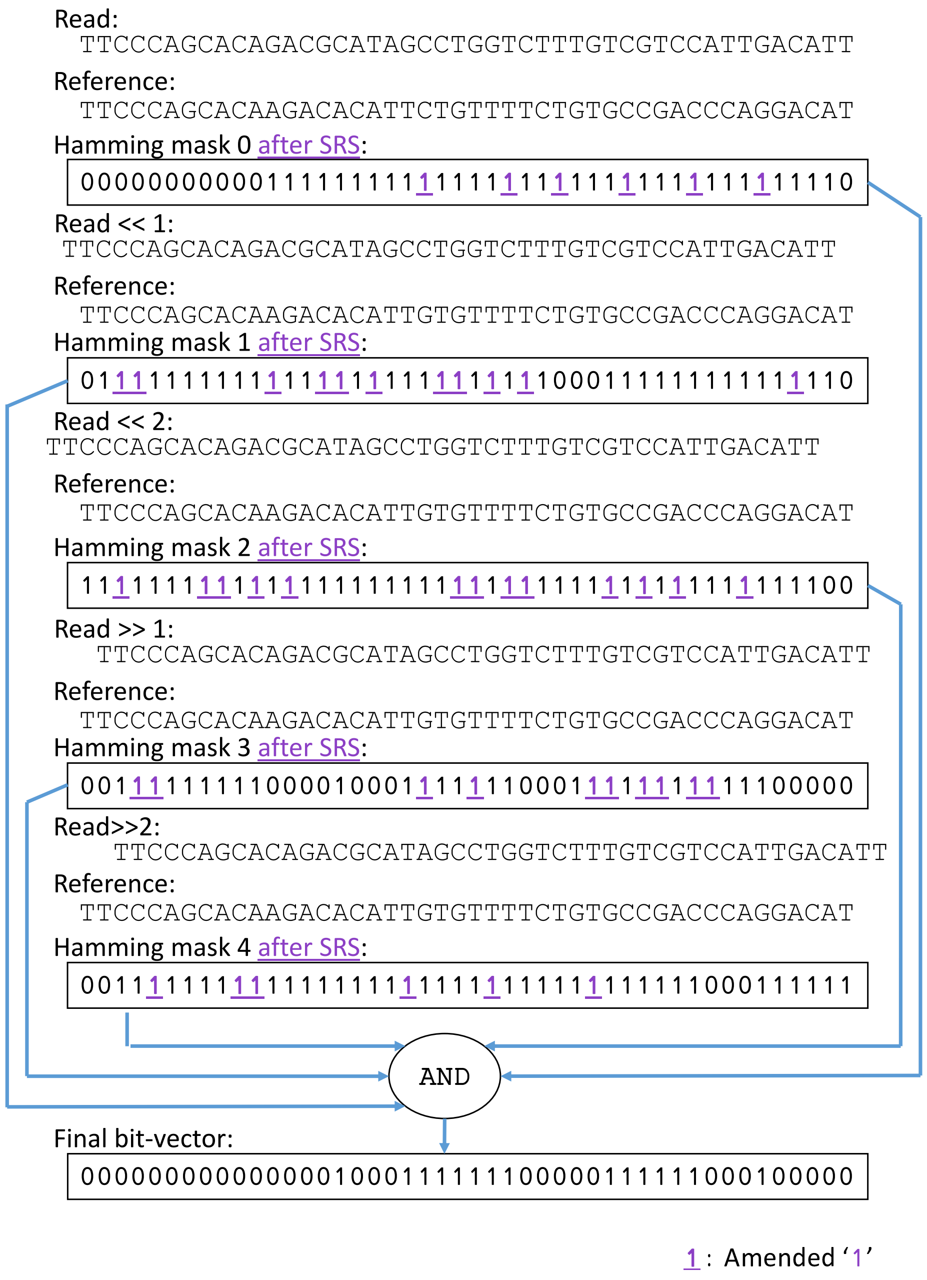
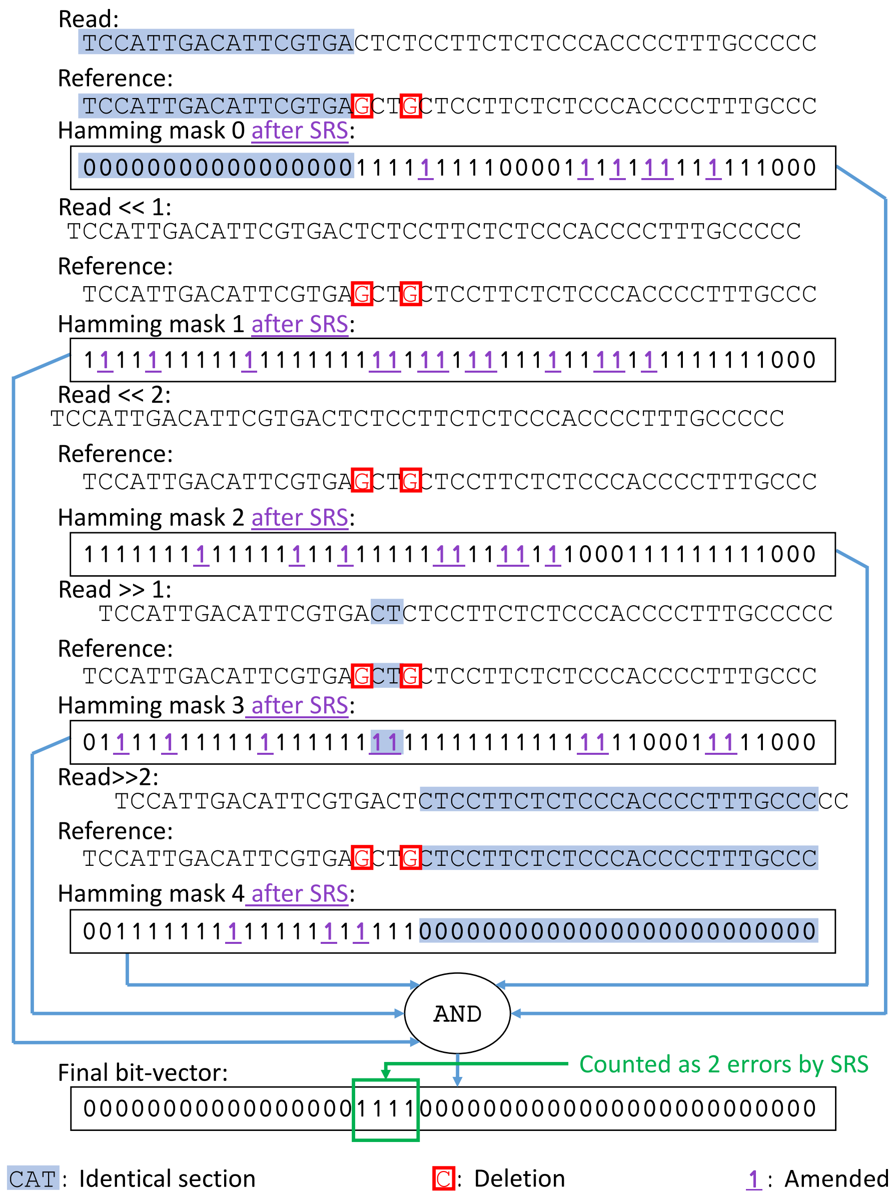

Read:
TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATT
Reference:
TTCCCAGCACAAGACACATTCTGTTTTCTGTGCCGACCCAGGACAT
Hamming mask 0 after SRS:
0000000000011111111111111111111111111111111111110
Read << 1:
TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATT
Reference:
TTCCCAGCACAAGACACATTGTGTTTTCTGTGCCGACCCAGGACAT
Hamming mask 1 after SRS:
0111111111111111111111111110001111111111111110
Read << 2:
TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATT
Reference:
TTCCCAGCACAAGACACATTGTGTTTTCTGTGCCGACCCAGGACAT
Hamming mask 2 after SRS:
1111111111111111111111111111111111111111111100
Read >> 1:
TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATT
Reference:
TTCCCAGCACAAGACACATTGTGTTTTCTGTGCCGACCCAGGACAT
Hamming mask 3 after SRS:
0011111111100001000111111100011111111111100000
Read>>2:
TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATT
Reference:
TTCCCAGCACAAGACACATTGTGTTTTCTGTGCCGACCCAGGACAT
Hamming mask 4 after SRS:
0011111111111111111111111111111111111000111111
AND
Final bit-vector:
0000000000000001000111111100000111111000100000
1 : Amended '1'

Figure 2.4: The incorrect mapping from Figure 2.3 is filtered correctly by SRS.

alignment (the alignment produced by the local alignment computation) of the mapping. We call these '0's **spurious**, as they conceal mismatch errors and give the false impression that the read and the reference have a small edit distance, even when they differ significantly.

We propose a heuristic, *speculative removal of short-matches (SRS)*, which aims to remove spurious '0's. SRS uses one important observation: *identical sections are typically long* ($\geqslant 10$ *basepairs) while streaks of spurious '0's are typically short* (<3 *basepairs).* This insight is confirmed empirically through experiments, but is also supported by theory. Given that for most mappers e is in general less than 5% of the read length L, the average length of an identical



Read:
TCCATTGACATTCGTGACTCTCCTTCTCTCCCACCCCTTTGCCCCC
Reference:
TCCATTGACATTCGTGAGCTGCTCCTTCTCTCCCACCCCTTTGCCC
Hamming mask 0 after SRS:
0000000000000000011111111110000111111111111000
Read << 1:
TCCATTGACATTCGTGACTCTCCTTCTCTCCCACCCCTTTGCCCCC
Reference:
TCCATTGACATTCGTGAGCTGCTCCTTCTCTCCCACCCCTTTGCCC
Hamming mask 1 after SRS:
1111111111111111111111111111111111111111111000
Read << 2:
TCCATTGACATTCGTGACTCTCCTTCTCTCCCACCCCTTTGCCCCC
Reference:
TCCATTGACATTCGTGAGCTGCTCCTTCTCTCCCACCCCTTTGCCC
Hamming mask 2 after SRS:
1111111111111111111111111111111000111111111000
Read >> 1:
TCCATTGACATTCGTGACTCTCCTTCTCTCCCACCCCTTTGCCCCC
Reference:
TCCATTGACATTCGTGAGCTGCTCCTTCTCTCCCACCCCTTTGCCC
Hamming mask 3 after SRS:
0111111111111111111111111111111111110001111000
Read>>2:
TCCATTGACATTCGTGACTCTCCTTCTCTCCCACCCCTTTGCCCCC
Reference:
TCCATTGACATTCGTGAGCTGCTCCTTCTCTCCCACCCCTTTGCCC
Hamming mask 4 after SRS:
0011111111111111111110000000000000000000000000
AND
Final bit-vector:
0000000000000000011110000000000000000000000000
Counted as 2 errors by SRS
CAT: Identical section
C: Deletion
1: Amended '1'

Figure 2.5: Correct short streaks of '0's might also get overwritten by SRS.

section is greater than 16 basepairs for, say, $L = 80$. ($l_{sec} \geqslant \frac{L}{0.05L+1} \approx 16$). The probability that a streak of n '0's will be spurious (i.e., part of a random alignment between basepairs) is $(1/4)^n$. For streaks where n is greater than 3 basepairs, the probability of being spurious is below 1%.

Using this insight, we replace all streaks of '0's in the Hamming masks that are shorter than three digits with '1's. We call the '1's that replace the '0's (i.e., amended from '0's) as *amended '1's*. Amended '1's do not affect the final bit-vector of the SHM as they are "transparent" during AND operations. The potential trade-offs and reasoning for choosing three as our threshold for SRS is discussed in Section 2.1. Note, the incorrect mapping which passed SHM in Figure 2.3 is identified and correctly rejected using SRS in Figure 2.4.

Since SRS amends all short streaks of '0's, even the ones produced by correct alignments of

Algorithm 1: SHD

```
Inputs: Read[0]...Read[s − 1], Ref[0]...Ref[s − 1] (bit-vectors of the Read and
  Reference), $e$ (error threshold)
Outputs: Pass (returns True if the string pair passes the SHD)
Functions: see Supplementary Materials
ComputeHammingMask: computes the Hamming mask
SRS_amend: amends short streaks of ‘0’s into ‘1’s
SRS_count: counts the number of errors in the final bit-vector
Pseudocode: HMask = $ComputeHammingMask$(Read, Ref);
Final_BV = $SRS\_amend$(HMask);
for i = 1 to $e$ do
    // Left shift Read
    for j = 0 to $s - 1$ do
        ShiftedRead[j] = A[j] << i;
    HMask = $ComputeHammingMask$(ShiftedRead, Ref);
    SRS_HMask = $SRS\_amend$(HMask);
    Final_BV = Final_BV&SRS_HMask;
    // Right shift Read
    for j = 0 to $s - 1$ do
        ShiftedRead[j] = A[j] >> i;
    HMask = $ComputeHammingMask$(ShiftedRead, Ref);
    SRS_HMask = $SRS\_amend$(HMask);
    Final_BV = Final_BV&SRS_HMask;
errorNum = $SRS\_count$(Final_BV);
if errorNum $\leqslant e$ then
    Pass = True;
else
    Pass = False;
return Pass;
```

basepairs, it could cause correct read mappings to be mistakenly filtered, as shown in Figure 2.5. To avoid this possibility, SRS counts the number of errors in the final bit-vector more conservatively than SHM. Each streak of ‘1’s in the final bit-vector could be the outcome of multiple streaks of amended ‘1’s. However, ‘0’s are changed only if they are two-or-fewer-bit ‘0’ streaks and are surrounded by ‘1’s. In the worst case, multiple back-to-back short identical sections that are separated by single errors can be mistakenly overwritten into a long streak of ‘1’s (e.g., 1001001 → 11111111). As a result, the number of errors covered by a streak of ‘1’s (e_1) of length l_1 after SRS is $e_1 = 1 + [(l_1 + 1)/3]$. The streak of four ‘1’s in the final bit-vector of Figure 2.5 is now counted as only two errors rather than four and the correct mapping passes the filter. Using this counting scheme, we ensure all correct mappings will pass through the filter, while still identifying and removing read and reference pairs with errors up to 5% of the read length (results are provided in Section 2.4).

To achieve high efficiency, we implemented both SRS_amend and SRS_count using Intel SSE instructions. Specifically, we used the *packed shuffle* operation, a SIMD parallel-table-lookup instruction provided by the Intel SSE instruction set.

The SSE instruction packed shuffle ($pshuff$) takes two vectors of integers $A = [a_0\ a_1\ a_2\ ...]$ and $B = [b_0\ b_1\ b_2\ ...]$ and replaces integers in vector A with integers in vector B with the result $A = [B[a_0]\ B[a_1]\ B[a_2]...]$. Figure 2.6 gives an example of applying $pshuff$ on a single bit-vector. Notice that the values of the integers in vector A must not exceed the maximum size of vector B, otherwise an incorrect result is returned. On Intel platforms, the maximum size of an SSE vector is 16 (with single-byte integers). Therefore, on Intel platforms, the value of the elements in either vectors must not exceed 16 (1111 in binary, four bits).

Implementation of SRS_amend

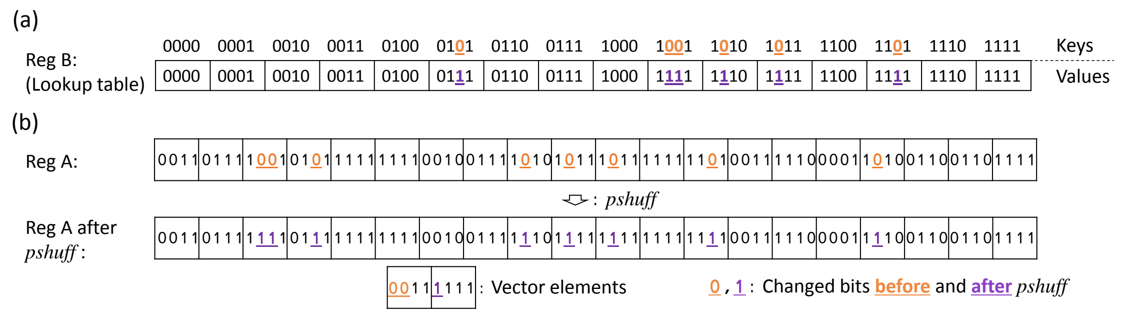


Figure 2.6: An example of quickly correcting short streaks of '0's using $pshuff$ operations.

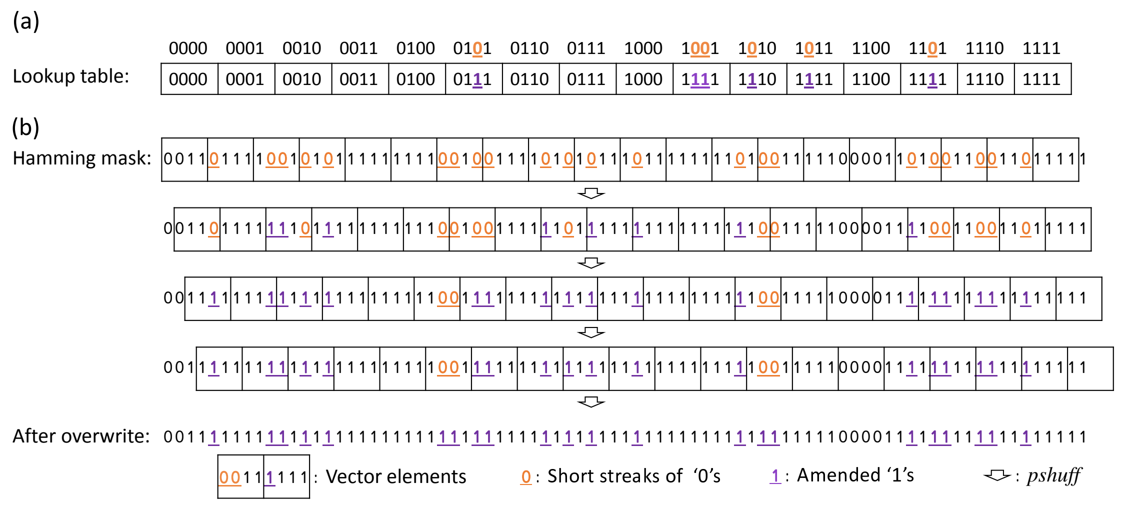


Figure 2.7: An example of the overall procedure of SRS, including four $right\ shift$s and $pshuff$s.

Algorithm 2 shows the pseudocode of SRS_amend. We use packed shuffle to efficiently replace all short streaks of '0's with '1's in parallel. According to SRS (Section 2.3.2), all streaks of '0's shorter than three (and are bounded by '1's) are turned into streaks of '1's. As a result,

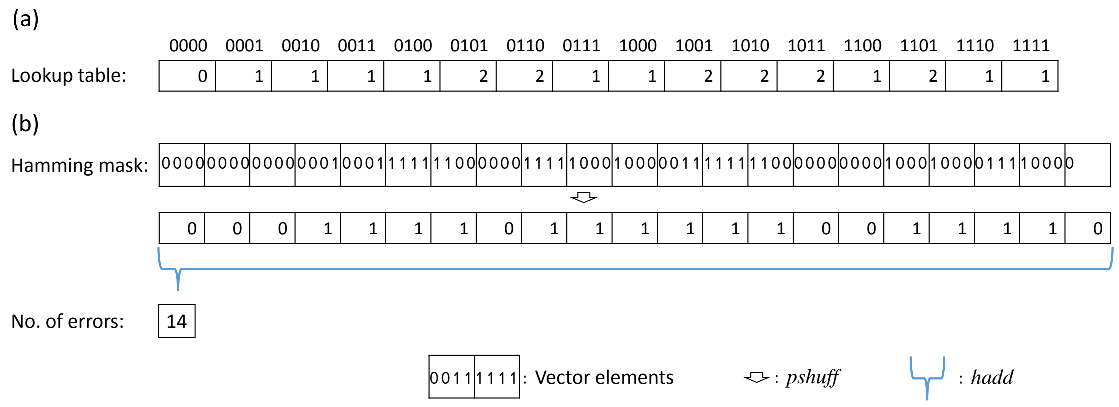


Figure 2.8: Conservatively counting errors with SHM.

bit streams such as 101, 1001 are replaced with 111 and 1111, respectively. Packed shuffle can drastically speed up this replacement process by storing the pre-processed replacement patterns in a lookup table and replacing target bit streams in parallel at runtime. Specifically, the lookup table stores replacement bit streams for binary keys ranging from 0000 to 1111 (16 keys in total), as shown in Figure 2.6 (a). Among all keys, for those that have '0's in the middle of '1's, such as 0101 or 1010, their values replace the middle '0's with '1's (0101 → 0111 and 1010 → 1110); for all other keys, their values are simply the keys themselves (hence packed shuffle does not change them). Figure 2.6 (a) presents the entire lookup table for SRS_amend. Figure 2.6 (b) presents an example of amending short streaks of '0's using a single packed shuffle operation.

To successfully amend all short streaks of '0's into '1's, however, a single packed shuffle operation is not enough. Short streaks of '0's that span two neighboring keys in the lookup table stay unchanged after the packed shuffle operation (e.g., 0010 0100 → 0010 0100). To solve this problem, we incrementally shift the bits in the vector to the right (as shown in Figure 2.7 (b)) which eventually brings the short streak of '0's into one key (0010 0100 >> 001 0010 0 >> 00 1001 00 → 00 1111 00). Figure 2.7 (b) illustrates SRS_amend (Algorithm 2) amending all short streaks of '0's in a Hamming mask through four *pshuff* operations and shifts.

Note that to amend a short streak of '0's using packed shuffle, the entire short streak of '0's and the bounding '1's must fit into the space of a single key at once (e.g., $\underbrace{1001}$ but not $\underbrace{1000}1$). Since the packed shuffle operation on Intel platforms only supports a key length of four bits at maximum, it cannot amend any short streak of '0's longer than two. As a result, here we choose an SRS threshold of *three bps*. A study of the effect of other SRS thresholds on the false positive rate is provided in Section 2.1.

Implementation of SRS_count

Similar to SRS_amend, SRS_count is implemented using packed shuffle as well. According to SRS (Section 2.3.2), a streak of '1's in the final bit-vector of SHM is always assumed to be amended from back-to-back short streaks of '0's. Therefore, the number of errors of a short streak of '1's must be counted as the minimum errors that this streak can cover.

Using packed shuffle, we can quickly provide a lower bound of the number of errors that the final bit-vector contains. In this case, the lookup table of packed shuffle stores the minimum

```
Algorithm 2: SRS_amend
    Inputs: HMask (Hamming mask), LTable (lookup table)
    SIMD Registers: r1, r2 and r3
    Outputs: SRS_HMask (SRS amended Hamming mask)
    r1 = HMask;
    r2 = LTable;
    r3 = $pshuff$(r2);
    r1 = r1 |r3;
    for $i = 1$ to $3$ do
        r3 = r1 >> $i$;
        r3 = $pshuff$(r2);
        r3 = r1 << $i$;
        r1 = r1 | r3;
    SRS_HMask = $R1$;
    return SRS_HMask;
```

number of errors that each key covers. For example, key 0100 clearly covers only a single error hence, stores a “1” in the table while key 0110 clearly covers two errors hence, stores a “2” in the table. However, for keys such as 1100 or 1111, it is unclear what bits are next to them hence the minimum number of errors that they cover is hard to determine (e.g., in bit-stream [0000 1100] 1100 covers two errors, but in bit-stream [0001 1100] 1100 covers only one error). In order to preserve correctness (such that we do not not over-estimate errors for any input) while maintaining speed, a lower bound of errors is (always) assumed for such keys. A complete lookup table for SRS_count is provided in Figure 2.8 (a).

```
Algorithm 3: SRS_count
    Inputs: Final_BV (the final bit-vector of the SHM), LTable (lookup Table)
    SIMD Registers: r1 and r2
    Outputs: errorNum (minimum number of errors in the bit-vector)
    r1 = Final_BV;
    r2 = LTable;
    r1 = $pshuff$(r2);
    errorNum = $hadd$(r1);
    return errorNum;
```

Algorithm 3 provides the pseudocode of SRS_count. As the pseudocode shows, we first load the pre-processed lookup table into a SIMD register. Then, we count the minimum number of errors of each key in the final bit-vector using packed shuffle. Finally, we sum up all minimum numbers of errors of all keys using a horizontal add (*hadd*). The final sum is a lower bound of the minimum number of errors of the final bit-vector (and the potential mapping). Figure 2.8 (b) visualizes the entire workflow of Algorithm 3. In this figure, a minimum of 14 errors is counted from the final bit-vector, which indicates that the potential mapping must be erroneous hence must be rejected.

SRS can be implemented using SIMD-friendly operations. As we explain in Supplementary Materials, the ability to implement SRS with SIMD instructions is crucial for the high performance of SHD, as it enables computing SRS in constant time with few instructions: both overwriting of short streaks of ‘0’s and counting the number of errors of streaks of ‘1’s can be computed in constant time using SIMD *packed shuffle* operations. See Section 1.3 in Supplementary Materials for details.

Combined with SHM, SRS and SHM form the two-step filtering algorithm Shifted Hamming Distance (SHD), which guarantees that correct read mappings are preserved, while quickly removing incorrect mappings with simple bit-parallel operations.

Pseudocode

The pseudocode of SHD is shown in Algorithm 1. Overall, SHD computes $2e + 1$ Hamming masks (ComputeHammingMask), with e of them computed with the read incrementally shifted to the left; e of them computed with the read incrementally shifted to the right, and one computed without any shifts. Each Hamming mask is then processed by SRS to amend short streaks of ‘0’s into ‘1’s (SRS_amend). Finally, all Hamming masks are merged together into a final bit-vector through bit-wise AND operations and a lower bound of errors is computed from the final bit-vector (SRS_count). Details of implementations of ComputeHammingMask, SRS_amend and SRS_count are discussed in Supplementary Materials.

False Negatives

SHD never filters out correct mappings; hence, it has a zero false negative rate. As we discussed in Section 2.3.2, identical sections longer than three bps are recognized and preserved in the final bit-vector by SHD. Identical sections shorter than three bps are amended into ‘1’s; however, SHD counts ‘1’s in the final bit-vector conservatively, ensuring correct mappings are not filtered.

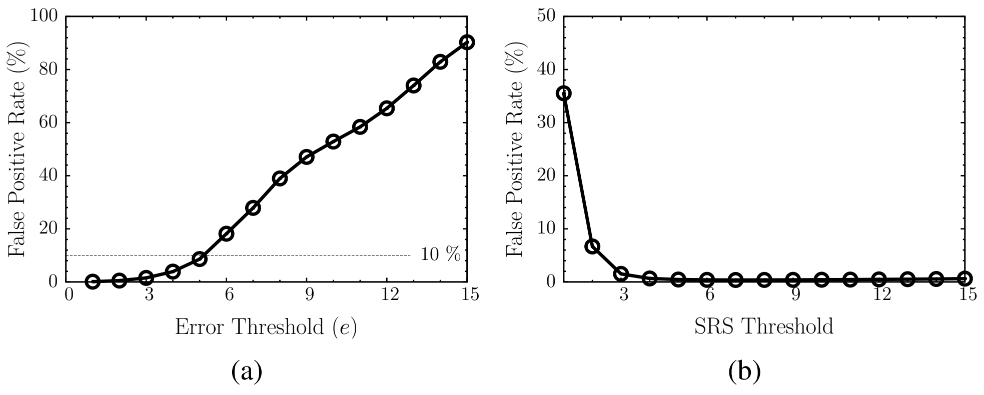


Figure 2.9: A sweep of the false positive rate of SHD, against variant allowed error rate.

False Positives

SHD does allow a small portion of incorrect mappings to pass the filter as false positives. This is acceptable since SHD is only a filter. Incorrect mappings that pass SHD are discarded later by more rigorous edit-distance calculations. Below, we describe two major sources of false positives, both of which are related to the threshold of the SRS (the minimal length of a streak of '0's that will not be amended by SRS).

First, long streaks of spurious '0's are not identified by SRS. Although less likely, long streaks of '0's can still be spurious (i.e., identical substrings between the read and the reference that do not belong to the correct alignment between the read and the reference). Long spurious streaks of '0's in an incorrect mapping can mask out real errors ('1's in other Hamming masks) and produce a mostly '0' final bit-vector even though the read and the reference differ by more errors. We can increase the SRS threshold beyond three bps, which amends longer streaks of '0's, to reduce such false positives.

Second, SRS may underestimate the number of errors while examining the final bit-vector. SRS always assumes the worst case where any streak of '1's in the final bit-vector is the result of amending short streaks of spurious '0's, despite the possibility it could be a sequence full of real errors. When counting streaks of '1's, SHD only assigns the minimal number of errors required to produce the pattern (e.g., **1**00**1111** → 11111111 → **1**00**1**00**1** 3 errors counted when 5 errors are present). By always assuming the worst case, SHD may underestimate the number of errors in the final bit-vector and let incorrect mappings pass the filter. Although using a smaller SRS threshold would help filter out such false positives, it would also let long streaks of spurious '0's pass the filter as we described in the previous paragraph. As a result, a carefully chosen SRS threshold should consider both factors: it should neither be too small to omit long spurious '0's nor should it be too large to underestimate the number of errors. Figure 2.9a shows this dilemma, as the false positive rate first drops and then slowly increases as SRS threshold increases. We chose three as our SRS threshold because: 1, the false positive rate of SHD drops below 2% (with the configuration of $e = 3$) at three and remains steady afterwards and 2, with Intel SSE platform we are only able to provide an efficient implementation of SHD with an SRS threshold of no-more-than three (further elaborated in Supplementary Materials).

A sweep of the false positive rate of SHD against the variant allowed error rate (error threshold divide by read length) is shown in Figure 2.9b. While the false positive rate of SHD increases with larger allowed error rate, at 5% error rate (which is the upper limit of most available mappers [3, 4, 25, 47, 54, 59, 84, 100]), the false positive rate of SHD is only 7%, indicating a high accuracy ($> 93\%$) of the filter.

2.4 Results

We implemented SHD in C, using Intel SSE. We compared SHD against three edit-distance calculation/filtering implementations introduced in Section 1.3, they are: SeqAn [26], an implementation of Gene Myers's bit-vector algorithm [74]; swps3 [94], a Smith-Waterman algorithm [92] implementation; and FastHASH [102], an Adjacency-Filtering (AF) implementation. Both SeqAn and swps3 are also implemented with SSE and all implementations were configured to be

	ERR240726_1	ERR240726_2	ERR240727_1	ERR240727_2	ERR240728_1
No. of Reads	4,031,354	4,031,354	4,082,203	4,082,203	3,894,290
Read Length	101	101	101	101	101

	ERR240728_2	ERR240729_1	ERR240729_2	ERR240730_1	ERR240730_2
No. of Reads	4,389,429	4,013,341	4,013,341	4,082,472	4,082,472
Read Length	101	101	101	101	101

Table 2.1: Benchmark data, obtained from the 1000 Genomes Project Phase I [2]

single threaded.

We used a popular seed-and-extend mapper, mrFAST [4], to retrieve all potential mappings (read-reference pairs) from ten real data sets from the 1000 Genome Project Phase I [2]. Table 2.1 lists the read length and read size of each set. Each read set is processed using multiple error thresholds (i.e., e from 0 to 5 errors).

We benchmarked all four implementations using the same potential mappings (i.e., seed hits) produced by mrFAST for a fair comparison of the four techniques. Figure 2.10 shows the execution time of the four techniques with different error thresholds across multiple read sets. Notice that when the indel threshold is zero, SHD reduces to bit-parallel Hamming distance. A detailed comparison against bit-parallel Hamming distance implementation is provided in Supplementary Materials, Section 1.3.

Among the four implementations, SHD is on average $3\times$ faster than SeqAn and $24\times$ faster than swps3. Although SHD is slightly slower than FastHASH (AF) when e is greater than two (e.g., $2.5\times$ slower when $e = 5$), SHD produces far fewer (on average, $0.25\times$) false positives than FastHASH (seen in Figure 2.11). Note, the speedup gained by SHD diminishes with greater e. This is expected since the number of bit-parallel/SIMD operations of SHD increases for larger e.

Figure 2.11 illustrates the false positive rates of SHD and FastHASH (AF). SeqAn and swps3 both have a 0% false positive rate, compared to SHD which has a 3% false positive rate on average. That being the case, SHD is only a heuristic to filter potential mappings while both SeqAn and swps3 must compute the exact edit distances of the potential mappings.

As we discussed in Section 2.1, the false positive rate of SHD increases with larger e. Nonetheless, the false positive rate of SHD at $e = 5$ is only 7%, much smaller than the false positive rate (50%) of FastHASH (AF) as Figure 2.11 shows.

With these results, a mapper can selectively combine multiple implementations together to construct an efficient multi-layer filter/edit-distance calculator. For instance, a mapper can attach SHD with FastHASH, in order to obtain both the fast-speed of FastHASH and the high accuracy of SHD. A mapper can also combine SHD with SeqAn to obtain 100% accuracy without significantly sacrificing the speed of SHD. Of course, there are many possibilities to integrate SHD into a other mappers, but a comprehensive study of this topic is beyond the scope of this work and is part of our future work.

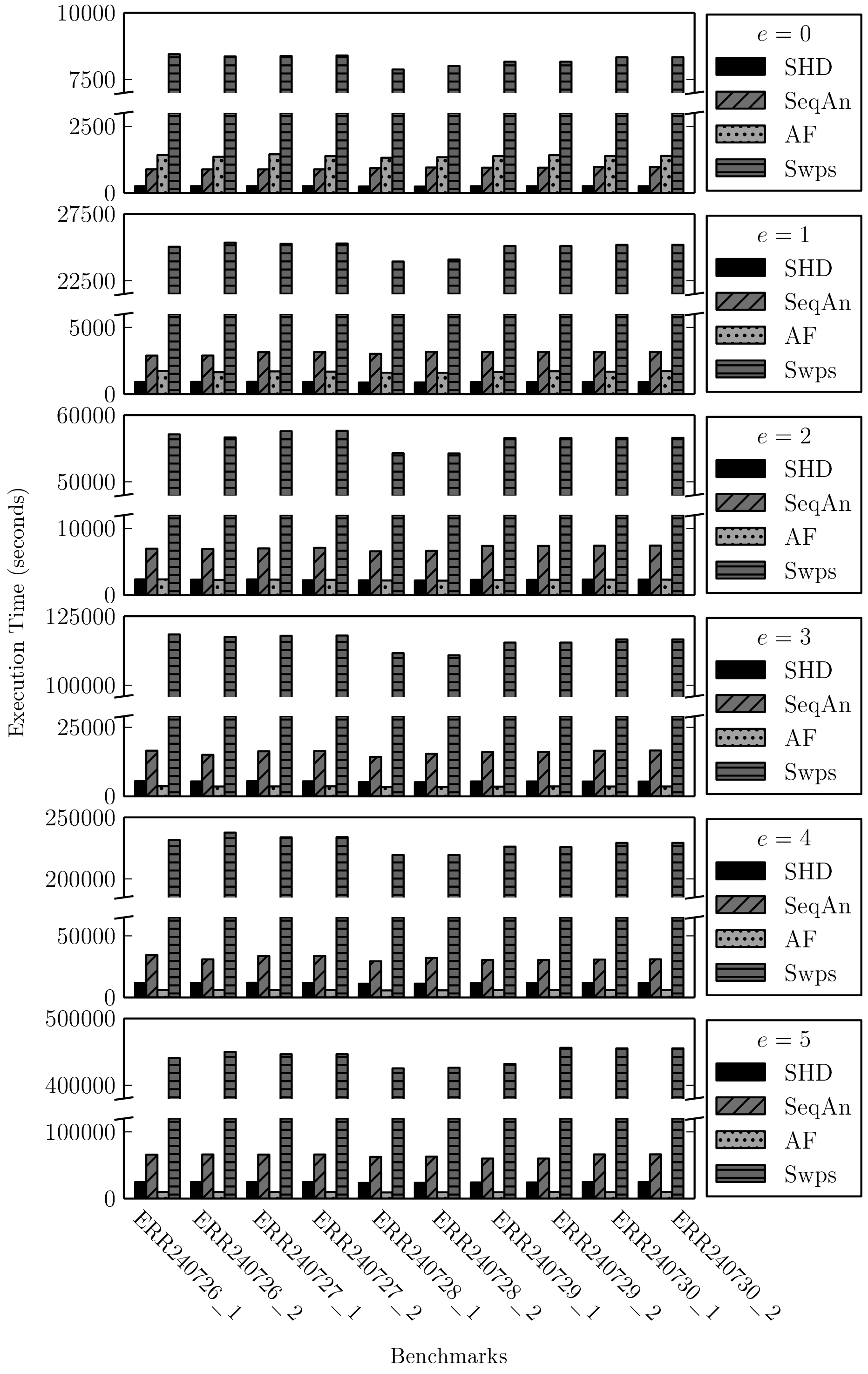


Figure 2.10: The execution time of SHD, SeqAn, swps3 and FastHASH (AF) with different error thresholds (e) across multiple read sets.

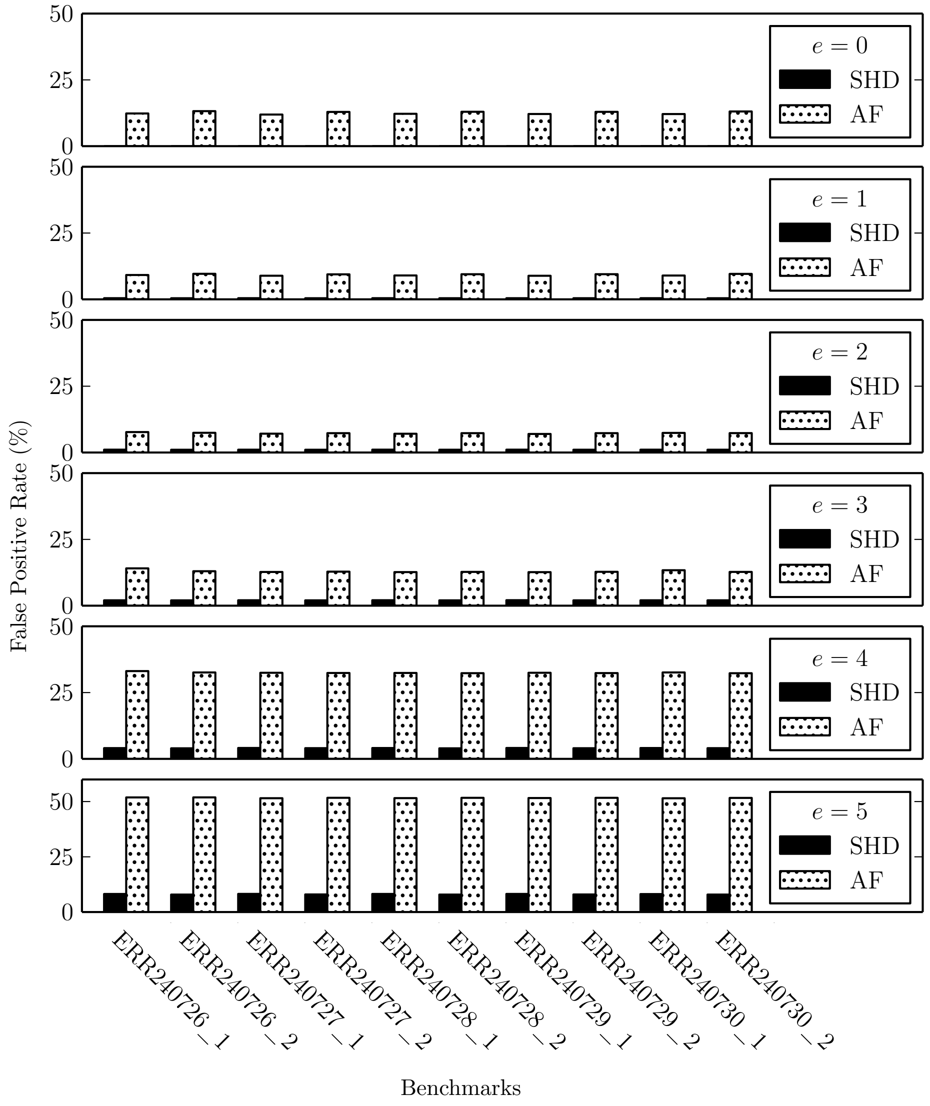


Figure 2.11: The false positive rates of SHD and FastHASH (AF) with different error thresholds (e) across multiple read sets.

2.5 Conclusion

Most potential mappings that must be verified by seed-and-extend based mappers are incorrect, containing far more errors than what is typically allowed. Our proposed filtering algorithm, SHD, can quickly identify most incorrect mappings (through our experiment, SHD can filter 86 billion potential mappings within 40 minutes on a single thread while obtaining a false positive rate of 7% at maximum), while preserving all correct ones. Comparison against three other state-of-the-art edit-distance calculation/filtering implementations revealed that our Intel SSE implementation of SHD is $3\times$ faster than SeqAn [26], the previous best edit-distance calculation technique.

Chapter 3

LEAP: A Generalization of the Landau-Vishkin Algorithm with Custom Gap Penalties

3.1 Background

While edit-distance provides a coarse measurement of similarities between the read and the reference, it is often considered too unrefined to be directly applied for biological analysis. Most notably, it gives equal penalties to insertion, deletion and substitution of base-pair, no matter where such errors occurs between the read and the reference.

Assigning equal penalties to insertions, deletions as well as substitutions alike is often considered inaccurate since insertions and deletions happen much less frequently in nature than substitutions [19]. Furthermore, measuring similarity by edit distances suggests that consecutive insertions or deletions (or simply *indels*), or indel of a single long continuous sequence, share an equal penalty with multiple discrete short indels, as long as their total length equals the continuous long indel. This practice is also imprecise, as in nature having multiple indels in a short span is much less likely than having a single but longer indel. In fact, a single mutational event can create indels of different sizes; hence the likelihood between two individual indels does not share much difference, especially when their lengths do not differ in magnitudes. Overall, a good penalty score should differentiate penalties between indels and substitutions, as well as applying less penalty to single larger indel than multiple smaller indels.

One mostly common used penalty scheme that satisfies above requirement is the *affine-gap* penalty score. Given an indel of length w, affine-gap assigns a total penalty of $g(w) = p_{open} + w \cdot p_{extend}$ to the indel, where p_{open} is dubbed gap open penalty while p_{extend} is the gap extension penalty. The key idea of affine-gap penalty is that p_{open} designates the penalty of having a mutational event, while p_{extend} guarantees that short indels are penalized less compared to long indels, as long indels occur less frequently.

A major benefit of using edit distance, as opposed to affine-gap penalties, is that edit distance algorithms have been thoroughly studied with many improvements. Most of the past improvements focus on speeding up the canonical dynamic programming algorithm which fills an $L \times L$

matrix (assuming the two strings are of equal length L), as covered in the previous section.

An alternative strategy to the canonical dynamic programming method, is to iteratively find the longest matching substrings with an increasing number of edits. This concept was first proposed by Landau and Vishkin ([46]) and is often called the Landau-Vishkin algorithm For simplicity, we refer to the Landau-Vishkin algorithm as LV for the reminder of this chapter. One limitation with LV is that it was only proposed for edit distance scoring schemes and it is not proven to work for more general scoring schemes, including affine-gap penalties.

LV uses the fact that edit-distance is conserved along the diagonal for a sequence of matches, so it can simply traverse along the diagonal to the position of the next error. The length of the traversal is $\texttt{LCE}(i, j)$ (longest common extension), which is the length of the longest prefix which $s_{i..m}$ and $r_{j..n}$ share.

LV uses a variant matrix for edit-distance computation: $\texttt{LV}_{d,e}$ stores the maximal row along diagonal d with edit distance e, where d is calculated as $j - i$, where i is the row and j is column in the edit-distance matrix. By conditioning on the last error, the recurrence for $\texttt{LV}$ follows:

$$\texttt{LV}_{d,e} =$$

$$\max \begin{cases} \underbrace{\texttt{LV}_{d,e-1} + 1 + \texttt{LCE}(\texttt{LV}_{d,e-1} + 2, \texttt{LV}_{d,e-1} + d + 2)}_{\texttt{substitution}} \\ \underbrace{\texttt{LV}_{d-1,e-1} + \texttt{LCE}(\texttt{LV}_{d-1,e-1} + 1, \texttt{LV}_{d-1,e-1} + d + 1)}_{\texttt{insertion}} \\ \underbrace{\texttt{LV}_{d+1,e-1} + 1 + \texttt{LCE}(\texttt{LV}_{d+1,e-1} + 2, \texttt{LV}_{d+1,e-1} + d + 2)}_{\texttt{deletion}} \end{cases}$$

3.2 Contributions

In this chapter, we present an extension to the LV algorithm. We show that the same principle of Landau-Viskhin can be applied not only to global approximate string matching with edit distance penalty scores, but also to any banded global or semi-global approximate string matching problems with non-negative scoring schemes. To achieve this, we first propose a generalization of the approximate string matching problem called the *Leaping Toad* problem and show that all banded global and semi-global approximate string matching problems with positive penalty scores can be transformed into the Leaping Toad problem. Then we propose *LEAP*, a general dynamic programming solution for the Leaping Toad problem based on the Landau-Vishkin algorithm. Finally we provide a bit-vectorized de Bruijn sequence-based optimization over LEAP. We show that LEAP is 7.4x faster than the state-of-the-art bit-vector edit-distance implementations and 32x faster than the state-of-the-art parallel affine gap penalty Needleman-Wunsch implementations.

LEAP provides the following contributions:

- It proposes the Leaping Toad problem, a generalization of all banded global or semi-global approximate string matching problems with positive penalty scores. It then shows the detailed procedure of transforming approximate string matching problems with edit distance penalty scores and affine-gap penalty scores into the Leaping Toad problem.

- It provides a new algorithm, LEAP, an extension of the Landau-Vishkin's algorithm, that solves the general Leaping Toad problem.
- It provides a detailed proof of the optimality of LEAP. The proof confirms that LEAP captures the minimum-score edit sequence between the two strings, under any positive penalty scoring scheme.
- It provides a bit-vectorized, de Bruijn sequence-based optimization over LEAP, which uses a perfect hash function that exploits properties of de Bruijn sequences to find the position of the most significant '1' in a bit-vector with simple bit-vector operations.
- It shows that bit-vectorized LEAP is 7.4x faster than the state-of-the-art edit-distance approximate string matching implementations and up to 32x faster than the state-of-the-art affine-gap approximate string matching implementations.

3.3 Methods

Both the global and semi-global Banded Edit Distance Problem (BEDP) and Banded Affine Gap Distance Problem (BAGDP) can be generalized as a restricted optimal path finding problem in a directed acyclic graph. We call this the *Leaping Toad* problem (LTP). In this section, we first propose the Leaping Toad problem, and we show how a general edit-distance problem can be converted to an instance of LTP. Subsequently we propose an improved dynamic programming algorithm *LEAP* as a solution, followed by a proof of its optimality. We discuss the backtracking process of LEAP. In addition, we provide a de Bruijn sequence-based bit-vector optimization over LEAP. Finally, we discuss specific optimizations to the algorithm for affine-gap penalties.

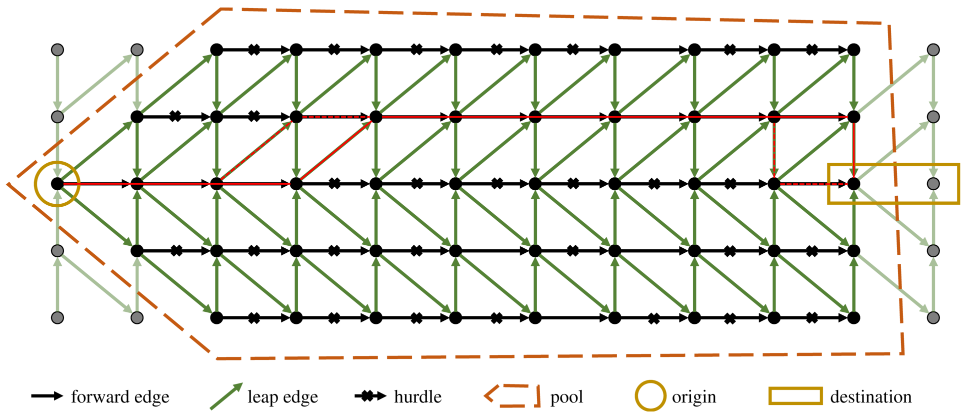


Figure 3.1: A swimming pool setup with LEAP of the banded edit distance problem.

3.3.1 The Leaping Toad Problem

The Leaping Toad Problem (or simply *LTP*) can be summarized as a traversal problem in a special directed acyclic graph, where a toad travels in the weighted graph and the goal is to find a path that connects the origin and the destination vertices while minimizing the sum of edge costs along the path.

The directed acyclic graph of LTP is described as follows:

- There is a **convex** *swimming pool* that encircles vertices taken from a 2-dimensional vertex grid, where vertices are aligned in rows and columns. Vertices in the swimming pool are then organized into disjoint *lanes* which are rows in the vertex grid. Inside a lane, each vertex is connected to the next vertex on its right by a directional edge, with itself being the source and the vertex on the right being the destination. We call these edges as *forward edges*. Forward edges only exist among vertices inside the swimming pool and do not exist for vertices outside of the swimming pool.
- A vertex may also have edges pointing to vertices in other lanes. We call these edges *leap edges*. In LTP, for a vertex and a separate lane, there can be at most one leap edge pointing to at most one vertex in that lane. In other words, there can not be multiple edges pointing to the same lane from the same vertex. We also require all the vertices in the same lane share the same types of leap edges: the same directions and lengths. When visualized, leap edges between two lanes are an array of parallel arrows. Notice that some leap edges might have their source and/or destination vertices staying out of the swimming pool and we call these edges *out edges*. Outside of the swimming pool enclosure, out edges continue to exist, connecting vertices between different lanes.
- Over any edge, there is a **non-negative integer** weight. Leap edges sharing the same origin and destination lanes have the same, **positive** weight. Forward edges have zero or positive weights. We call forward edges with positive weights as *hurdles*. Traveling across a hurdle is called *hurdle crossing*. Hurdles may have different costs.

In the swimming pool, we appoint a number of lanes as *origin lanes* and a number of lanes as *destination lanes*. The set of origin lanes and destination lanes may overlap. The general goal of the LTP is to find a path in the directed graph, with minimum sum of edge weights, that starts at the first vertex (the leftmost vertex in the swimming pool of the lane) of an origin lane and **either travels to the last vertex of a destination lane or travels out of the swimming pool while exiting onto a destination lane**.

For simplicity, we call edge weights as *energy costs*; we call traveling along the leaping edge as *leaps*. Also for the simplicity of developing a solution, we require all the leaping edges to never point backwards to vertices in previous columns on the left. A relaxation of this restriction is discussed in the Discussion section.

Figure 3.1 shows an example setup of the swimming pool as well as the optimal path to cross the pool (in red). In this setup, as the figure shows, the toad starts at the first vertex in the middle lane on the left side of the pool and the goal is to travel to the last vertex of the middle lane on the right side of the pool. In a lane, black crosses are placed on the hurdle edges.

In this particular setup, the toad can only leap to neighboring lanes, as the arrow shows. The leaping edges are set differently depending on whether the lane is 1) the middle lane, 2) above the middle lane or 3) below the middle lane. If it is in the middle lane, leaping edges are tilted by 45 degrees pointing to the vertex in the next column as they point to neighboring lanes. For other lanes, leap edges are vertical when they point towards the center lane and are tilted by 45 degrees when they point away from the center lane. Here we also set the energy cost of hurdles as well as leaps as 1. The red line depicts an optimal path for the toad to travel across the pool. Notice that there can be multiple optimal paths with the same total energy cost (shown as dashed red lines).

There are many alternative setups to LTP. For an alternative setup, a number of settings could be changed:

1. The energy cost of overcoming different hurdles can be different.
2. There can be leap edges pointing to more lanes.
3. The energy cost of leaps can be random and lane specific.

Figure 3.2 shows an alternative setup.

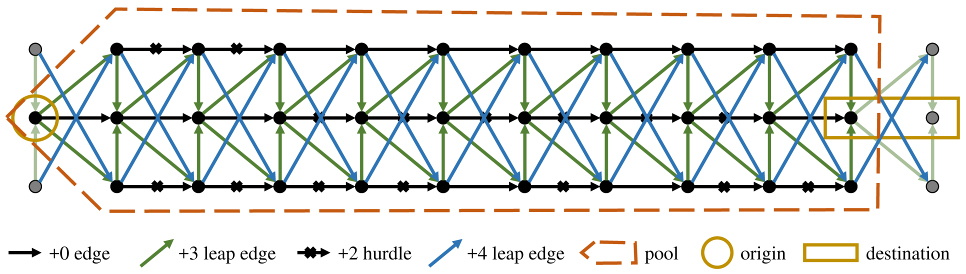


Figure 3.2: An alternative swimming pool setup with LEAP using custom penalties.

3.3.2 Conversion of Approximate String Matching to the Leaping Toad problem

Both the banded edit distance and the banded affine gap string matching problems can be converted to an instance of LTP. To convert both problems into LTP, we first convert both string matching problems into an optimal path finding problem in a directed graph. Then we show that the optimal path problem in the converted directed graph is indeed an instance of LTP.

The Banded Edit Distance Problem (BEDP) can be easily converted into an optimal path finding problem in a directed graph. For simplicity, we assume BLDP takes a pair of equal-length strings, such as strings r, s of length L. For the $(L+1)\times(L+1)$ edit-distance matrix D, we assign each element $D_{i,j}$ of the matrix a unique vertex $v_{i,j}$. Using the edit-distance recurrence function, a directional edge is drawn from vertex $v_{i,j}$ to $v_{i',j'}$ if and only if $v_{i,j} \neq v_{i',j'}$ and $i' - i \leqslant 1$ and $j' - j \leqslant 1$ (an edge to the right, bottom and bottom-right element). On each edge $(v_{i,j}, v_{i',j'})$, we place an integer weight w, where $w = 0$ if $i' = i + 1$, $j' = j + 1$, and $s_i = r_j$, or $w = 1$ otherwise. An example of the directed graph representation of the edit distance problem is shown in Figure 3.3. The objective function of BLDP becomes an optimal path finding problem where we want to find a path with minimum total edge weight within the edit-distance threshold e from $v_{0,0}$ to $v_{L,L}$.

For BLDP, the equivalent swimming pool directed graph setup is shown in the example in Figure 3.1. We call this the Edit Distance Leaping Toad setup. In general, given a $(N + 1) \times (N + 1)$ LDP matrix and a maximum edit-distance threshold e, we formulate the equivalent Edit Distance Leaping Toad setup as the following:

		A	C	T	T	A	G	C	A	C	T
	0	1	2								
A	1	0	1	2							
C	2	1	0	1	2						
T		2	1	0	1	2					
A			2	1	1	1	2				
G				2	2	2	1	2			
A					3	2	2	2	2		
A						3	3	3	2	3	
C							4	3	3	2	3
T								4	4	3	2
T									5	4	3

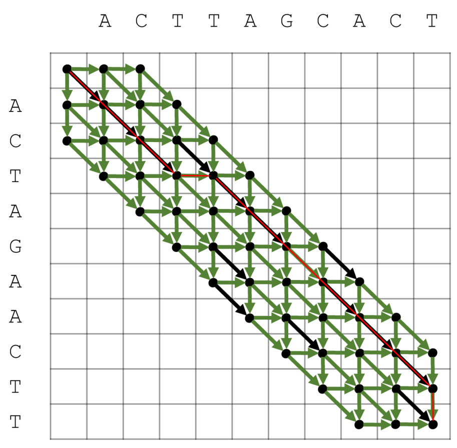


Figure 3.3: The directed graph representation of the dynamic programming method of the banded edit distance problem.

The swimming pool has $2e + 1$ lanes. The lane l contains $L - |l|$ vertices, specifically the set of vertices $v_{i,j}$ such that $i - j = l$. As Figure 3.1 shows, the right-end of all lanes are aligned while the left-end of the lanes forms a wedge shape. Each lane in the swimming pool corresponds to a diagonal in the edit-distance matrix by construction: the center lane represents the central diagonal and the kth lane above or below the center lane represents the kth diagonal above or below the central diagonal.

After mapping vertices and edges accordingly, we can observe that a hurdle is placed in the lane l between the kth and the $k + 1$st vertex if the corresponding edge in the BLDP graph has nonzero weight. Also for each lane in the pool, there are leap edges pointing to neighboring lanes. For the center lane ($l = 0$), leap edges are always tilted by 45 degrees. When they are on any other lane, leap edges are vertical when they are pointing towards the center lane and are tilted by 45 degrees when they are pointing away from the center lane. All hurdles and leaps cost 1 unit of energy.

After converting BLDP to the Edit Distance Leaping Toad setup, the goal becomes:

1. Determine if the toad can swim from the first vertex of the center lane to the last of the center lane while spending at most E energy.
2. If it can, then find the path that costs the minimum amount of energy. While there is slightly different from the goal of LTP, which allows traveling out of the swimming pool and allows terminating on the center lane but outside of the swimming pool, we will show later that for BLDP with small edit distance budget E, the result path of LTP is either the same with BLDP, or can be easily transformed into the path of BLDP. For now, we the goal of LTP is equivalent to the goal of BLDP.

The equivalence between the two directional graphs of LTP and BLDP can be visualized in Figure 3.4. For the vertex at the kth column of the lth lane above (or below) the center lane in the swimming pool, we assign its equivalent vertex in the edit distance direction as the vertex of element $E_{x,y}$ in the edit-distance matrix E, where $x = k - l$ and $y = k$ (or $x = k$ and $y = k - l$).

For example, the red, yellow, and blue vertices highlighted in both figures are equivalent between the two graphs, as well as their inbound and outbound edges. It's worth noting that vertices in the same column in the swimming pool forms a mirrored "L" shape in the BLDP graph (highlighted as purple lines in both graphs in the figure), with the vertex of the center lane on the corner. Also, no leap edges point backwards. All leap edges point either to vertices in the same column or in the next column.

For semi-global alignment, the objective function of BLDP changes from finding an optimal path **from the top-left element of the matrix to the bottom-right element of the matrix** (global alignment), to finding an optimal path **from any element of the first column of the matrix to any element of the last column of the matrix** (semi-global alignment), we simply need to reflect the same changes in LTP. Therefore for semi-global alignment, the objective function in the Leaping Toad setup changes to finding an optimal path from **the first vertex of bottom half lanes to the destination of top half lanes** with minimum energy cost.

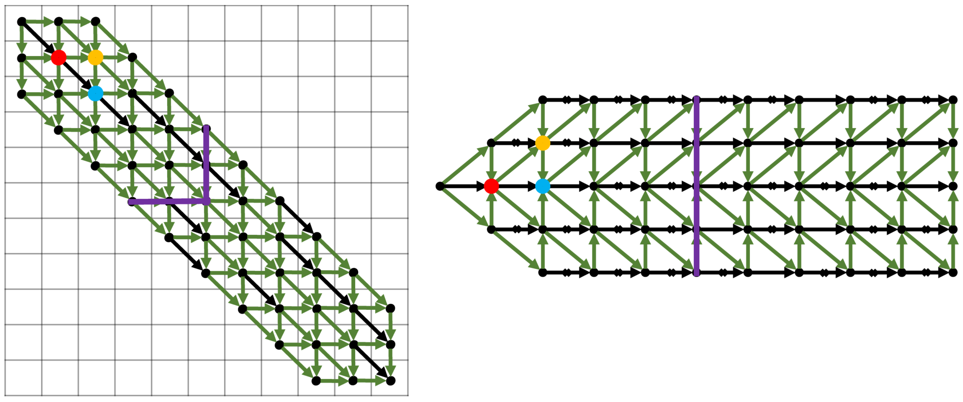


Figure 3.4: Illustration of the equivalence between the directed graph in Figure 3.3 and the Leaping Toad setup in Figure 3.1.

For Banded Affine Gap Distance Problem (BAGDP), or in general for any banded custom-gap-penalty string-matching problem (with a maximum insertion or deletion threshold k, a maximum energy budget E and positive gap penalties for different gap lengths), as now the toad can take an arbitrary length insertion or deletion (within the insertion limit k) and each insertion length has its own penalty score, we modify the swimming pool setup as each lane can have leap edges pointing to up to k lanes above and below the lane. Leap edges having the same origin but different destinations will have different energy costs. Figure 3.2 shows the leaping toad setup for a global BAGDP with a specific affine gap penalty scheme, where mismatches are penalized with +2, gap openings are penalized with +3 and gap extensions are penalized with +1.

3.3.3 LEAP: The General Solution of the Leaping Toad Problem

Similar to approximate string matching problems, the Leaping Toad Problem can be solved through dynamic programming. Since we restrict the toad from ever going backward, the toad can only reach a vertex from another vertex that is from the same or previous column. Therefore, for each new column, we can find the optimal paths leading to its vertices, as well as the minimum traveling energy, by reusing the optimal path results from previous columns: for each new vertex, we find all prior-vertices that can reach to the target vertex in one step, then pick the prior-vertex that requires the least amount of combined energy of both reaching to the prior-vertex and the intermediate move. We repeat this process until either we have reached a destination vertex, or no vertices in a column is still within the energy budget.

A major drawback of the above naïve dynamic programming solution of the Leaping Toad Problem, as with the naïve dynamic programming solution of the edit-distance problem, is that for each new vertex, we have to compare all of its previous-step vertices, then pick the vertex with the minimum overall energy cost. Similarly in backtrack, as we move one step backward at a time, we have to once again resolve the previous-step vertex for each and every vertex along the optimal path.

Inspired by the Landau-Vishkin algorithm for the edit-distance problem, we propose an improvement over the naïve solution of the Leaping Toad problem. We only consider switching lanes at vertices that are right before a hurdle. When there is no hurdle, we always let the toad swim forward; therefore avoid frequently checking possibilities of leaping from other lanes. We name this algorithm *LEAP*.

LEAP is developed upon a key observation that among all possible optimal paths with minimum energy costs, there must exist at least one optimal path that the toad **either never leaps or only leaps right before a hurdle or only leaps through out edges.**

Theorem 1. *Among all optimal paths of the Leaping Toad problem, there must exist one path in which the toad either never switches lanes or only switches right before hurdles.*

Before proving the theorem, we first define some terminology: we refer to a path from the origin vertex to the destination vertex simply as a *path*. The path may or may not have any lane switches. Whenever there is a lane switch, we call it a *leap*. Between two leaps, the toad only goes forward and we call such straight segments of the path as *segments*. We further categorize segments into two groups: segments that end with the destination vertex or a hurdle as *complete segments* and segments that do not end with such conditions as *incomplete segments*. We call the operation that extends the incomplete segment until it either reaches a destination vertex or a hurdle as *completing the segment*. Equipped with this terminology, we are now ready to prove the stepping-stone lemma of Theorem 1:

Lemma 1. *For a path with an incomplete segment S, there must exist an alternative path that shares the same moving sequence before S, while completing S into S_c and have at most the same cost.*

Proof. To prove the lemma, we need to find an alternative path that supports the claim. Assume in the original path, after S, the path continues with a series of leaps and segments, denoted as a *moving sequence*, [L_1, S_1, L_2, L_3, S_2, $\dots$], where S_i is the ith segment after S while L_j is the jth leap after S. Note that between two leaps there can be either zero or one segment, while between two segments there has to be at least a single leap.

Assuming that S_c is d vertices longer than S ($|S_c| = |S| + d$), we propose an alternative path that shares the same segments and leaps before S, followed by S_c, and then continues with the same sequence of leaps [L_1, $\dots$, L_t], while skipping all the segments from S_1 to S_{k-1}, where $\sum_{i=1}^{k-1} |S_i| < d \leqslant \sum_{i=1}^{k} |S_i|$, and L_t is right before S_k in the original moving sequence.

If $d < \sum_{i=1}^{k} |S_i|$, which suggests that after taking $[L_1, \ldots, L_t]$, the alternative path merges with the original path somewhere in S_k, then we also add the latter half of S_k after the merge point. The alternative path is then completed with the same moving sequence after S_k.

If original path does not have enough segments after S to match the length of d, then the alternative path simply takes the remaining leaps while skipping all the remaining segments. In this special case, the toad will take the out edges and leap out of the pool to finish the leaping sequence.

Compared to the original path, the alternative path is guaranteed to have at most the original energy cost. This is because:

1. The energy cost before S in the original path and before S_c in the alternative path are identical as they take identical moving sequences.
2. The energy costs of the two paths after the merge point (if they do merge) are also identical, as the two paths also take identical moving sequences.
3. The energy cost of the leaping sequence after S_c and before the merge point of the alternative path is at most the energy cost of the moving sequence after S and before the merge point in the original path. This is because the original path takes the same leaping sequence $[L_1, \ldots, L_t]$ (according to the settings of the LEAP problem, the energy cost of a leap only depends on the source and destination lanes but is irrelevant to the horizontal position of the toad. Hence, same leaping sequences have the same leaping energy costs, even if the horizontal positions of the leaps are different) but the other segments skipped by the alternative path may contain hurdles and hence cost extra.
4. The energy costs of S and S_c are identical since S_c is only a completion of S, and by construction the extension is free of hurdles so it costs zero energy.

□

Figure 3.5 depicts an example of converting a segment S in the original path (red) into S_c with an alternative path (blue) using the above procedure. Compared to the red path, the blue path consumes less energy as the red path contains segments with hurdles which are skipped by the blue path.

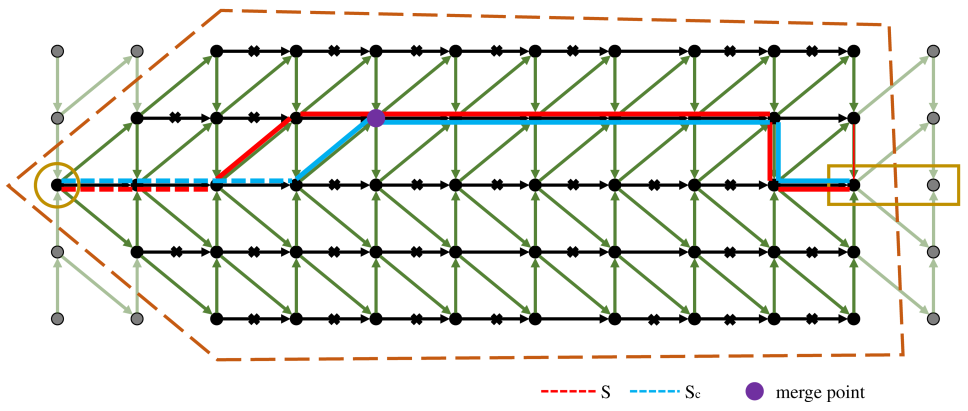


Figure 3.5: An example of Lemma 1.

With Lemma 1, we are now ready to proof Theorem 1. We prove through contradiction:

Proof. Assume there exist no optimal paths that either never leap or only have complete segments. Then for any optimal path, it must take at least one leap and have at least one incomplete segment.

We arbitrarily pick an optimal path ρ_1 and find the first incomplete segment S_1 in the path. Following the procedure in Lemma 1, we can find an alternative path ρ_2 without the incomplete segment S_1 with the same energy cost (ρ_2 cannot have a smaller energy cost; otherwise ρ_1 is not optimal). ρ_2 is also an optimal path, so by assumption, it must contain another incomplete segment S_2. We subsequently repeat the procedure.

The above process can only iterate a finite number of times since the procedure in Lemma 1 does not introduce any new segments, as completing S into S_c maintains that segment, and the procedure skips the segment sequence $[S_1,\ldots,S_{k-1}]$ and may either shorten or skip S_k, depending on the position of the merge point. Hence each iteration removes a incomplete segment. Given that there are finite number of incomplete segments, there can only be finite number of iterations. The final product after all iterations is a path with no incomplete segments with the same energy cost as ρ_1. However, according to our assumption, such path does not exist. Hence, this leads to a contradiction, which proves Theorem 1.

□

With Theorem 1, we now transform the general Leaping Toad problem of *finding an optimal path with minimum cost* to a sub-problem that *finds an optimal path which only contains complete segments*. As we have proven in Theorem 1, the resulting optimal path of the sub-problem must also be an optimal path of the general Leaping Toad problem.

LEAP solves the above sub-problem through an optimized dynamic programming method that can be viewed as an extension of the Landau-Vishkin algorithm. LEAP can be summarized into four steps:

1. LEAP iterates through all intermediate energy costs from 0 to E and for each energy cost, LEAP iterates through all lanes.
2. For an intermediate energy cost e and a lane l, LEAP finds the furthest vertex v in l that is reachable at precisely the energy cost e from either a leap or a hurdle-crossing.
3. LEAP extends the segment at v (if permitted) until the segment hits a hurdle.
4. LEAP repeats step 2) and 3) until either a lane has reached to the destination vertex or all intermediate energy levels have been exhausted. The path that leads to the destination vertex is reported as the final path.

To summarize, LEAP uses a core recurrence function shown below:

$$\texttt{start}[l][e] = \min_{\forall l' \in \texttt{lanes}} \texttt{end}[l'][e - P(l', l)] + F(l', l)$$

$$\texttt{end}[l][e] = \texttt{start}[l][e] + \texttt{VtH}(l, \texttt{start}[l][e])$$

In above equations, $P(l', l)$ returns the penalty of leaping from lane l' to l; $F(l', l)$ returns the number of columns the toad moves forward when it leaps from lane l' to l, and $\texttt{VtH}(l, \texttt{start\_column})$ (abbreviated for Vertices to Hurdle) returns the number of vertices until next hurdle from `start_column` in lane l. When $l = l'$, $P(l, l')$ is simply the energy cost of the next hurdle and $F(l, l') = 1$.

The detailed pseudo-code of LEAP is shown in Algorithm 4.

Theorem 2. *The final path returned by LEAP is indeed an optimal path of the sub-problem.*

We validate the correctness of Theorem 2 using two arguments. First, all segments in the final path of LEAP are guaranteed to be complete, because in step 3, LEAP always extends a segment until it reaches a hurdle. Second, for any energy cost e and any lane l, the last vertex extended in step 3 (if any) marks the furthest vertex that the toad can reach to in l using precisely e energy. Combining both arguments, we can conclude that if LEAP can find a path that reaches to the destination vertex with energy cost $e < E$ while the toad cannot reach the destination with energy cost $e' < e$, then the energy cost of the optimal path of the Leaping Toad sub-problem must be e (otherwise, according to the second argument, for a smaller energy, the toad would have already reached the destination) and the final path returned by LEAP must be an optimal path.

Proof. The first argument is obvious. The second argument can be proven through induction:

Base case: When e = 0, since any leap or hurdle-crossing would consume a non-zero amount of energy, the furthest vertex the toad can reach in a lane with zero energy cost would be the last vertex in the lane before hitting a hurdle. Therefore, the second argument holds true for the base case.

Induction step: Assume for all intermediate energy costs $e' < e$, and for all lanes, the second argument holds true. That is, for any lane l', the last vertex $\texttt{end}[l'][e']$ reached by step 3 in LEAP marks the furthest vertex the toad can reach in l' while consuming precisely e' amount of energy.

Now, because both hurdle crossings and leaps cost positive amount of energy, to get to a vertex with $e \neq 0$ energy cost in lane l, the toad has to either leap from a vertex in another lane or cross a hurdle in the same lane from a vertex in which the the total energy cost to get to that vertex is less than e . Since LEAP has already calculated the furthest vertices of all lanes for all energy levels $e' < e$ (based on our assumption), we can conclude that step 2 of LEAP will find the furthest vertex in l such that it is reachable from either a leap or a hurdle-crossing at precisely e energy cost.

Finally the only remaining method for the toad to get to a vertex while costing e energy, is to swim straight, without running into a hurdle, from a previous e-energy vertex. This vertex is also captured by LEAP in step 3. Therefore, the argument is correct for the induction step.

Conclusion: After step 3, LEAP always reflects the furthest vertex the toad can reach in the target lane l under the target energy cost e. □

3.3.4 Backtracking in LEAP

The pseudo-code of the backtracking method of LEAP is shown in Algorithm 5.

A remaining issue with LEAP, or LTP in general, is that it allows the toad to leap out of the swimming pool (recall the LTP and BLDP equivalent-goal assumption we made in Section 3.2). According to the LTP problem definition, as long as the toad reaches to a destination lane, even if the destination vertex is outside of the pool, the path is still acceptable. Translated back to approximate string matching problems, this sometimes leads to awkward results, since inserting or deleting letters (counterparts of leaps in the approximate string matching notion) beyond

Algorithm 4: LEAP

```
Input: $E$, destination_lanes, origin_lanes
Output: pass; final_lane; final_energy;
Initialization: end$[l][e]$ = start$[l][e]$ = $-$MAX_INT, $\forall(l, e)$;
final_energy = MAX_INT
Functions:
VtH$(l, pos)$: computes the number of vertices until the next hurdle from column from $pos$
 in lane $l$.
Pseudocode:
// Initialization
for $l$ in $[-k ... + k]$ do
    if $l$ in origin_lanes then
        start$[l][0]$ = origin_lanes$[l]$;
        length = VtH($l$, start[l][0]);
        end$[l][0]$ = start$[l][0]$ + length;
// Iterate through all energy levels
for $e = 1$ to $E$ do
    // Finds the furthest starting position
    // after a leap or a hurdle-crossing
    for $l$ in $[-k ... + k]$ do
        for $l'$ in $[-k ... + k]$ do
            $e' = e - P(l', l)$;
            if $e' \geqslant 0 \wedge$ end$[l'][e']$ then
                candidate_start = end$[l'][e'] + F(l', l)$;
                if candidate_start ¿ start$[l][e]$ then
                    start$[l][e]$ = candidate_start;
        // Find how long the toad can travel
        // without running into a hurdle
        length = VtH($l$, start[l][e]);
        end$[l][e]$ = start$[l][e]$ + length;
        if end$[l][e] \geqslant$ destination_lanes$[l]$ then
            if $e <$ final_energy then
                final_lane = $l$;
                final_energy = $e$;
pass = final_energy $< E$;
```

the end of the string is undefined. For instance, assume we are computing the global alignment between strings "AAAAAC" and "AAAAAG" with a simple scoring scheme: mismatches are penalized with +5, single letter gap is penalized with +4, double letter gap is penalized with +2 (this gap penalty might not make sense for DNA alignment, as here single letter gap is more costly than double letter gap). For this particular string pair, the correct edit sequence would be, `M-3I-2D-2`, which translates to, "3 matches, inserting 2 letters, then deleting 2 letters". The minimum edit cost would be +4. When subjected to LEAP, this string pair will instead generate an edit sequence of `M-4I-2D-2`. We have `M-4` simply because LEAP does not consider taking a leap before running into a hurdle, which in approximate string matching notion, is the `C-G` mismatch. Although the energy cost of the LEAP path is still +4, the edit sequence clearly does not make sense since one cannot delete two letters after 4 matches when there is only a single letter `C` left.

Algorithm 5: Backtrack

```
Input: final_lane; final_energy; origin_lanes
Output: path; path_count
Pseudocode:
// Initialization
$l$ = final_lane;
$e$ = final_energy;
path[0].start=start[$l$][$e$];
path[0].end=end[$l$][$e$];
path_count= 1;
// Stop when reached origin
while start[$l$][$e$] $\neq$ origin_lanes[$l$] do
    path[path_count].start=start[$l$][$e$];
    path[path_count].end=end[$l$][$e$];
    for $l'$ in $[-k, \ldots, +k]$ do
        $e' = e - P(l', l)$;
        if end[$l'$][$e'$] $+ F(l', l) ==$ start[$l$][$e$] then
            $l = l'$;
            $e = e'$;
            break;
    path_count++;
```

We can easily correct backtracking sequences from the out-of-bound LEAP backtrack sequences. Our work assumes that forward edges without hurdles always cost 0 energy. This corresponds to matches in approximate string matching and alignment. Hence, we can remove matches from the out-of-bound LEAP backtrack sequence until the length matches the intended length. This transformation maintains the total energy cost. For the above example, the intended sequence length is 7. So we have to simply remove one match, and we can remove the last `M` and transform `M-4I-2D-2` to `M-3I-2D-2`, which is an optimal edit sequence.

3.3.5 De Bruijn Sequence Optimization

While LEAP can drastically reduce the number of comparisons in the dynamic programming solution of the Leaping Toad problem, step 3 of LEAP still involves a costly loop that searches for the next hurdle.

By encoding the hurdle information as bit-vectors, we can significantly improve the performance of step 3 using de Bruijn sequences and bit-vector operations. The detailed technique is described in [49]. Here we provide a brief summary of the technique.

First, we encode the sequence of all forward edges (edges that go straight) of a lane as a bit-vector, where '0' denotes an edge that does not have a hurdle in between the two connected vertices while '1' denotes an edge that does. For example, the middle lane in Figure 3.1 can be represented as '0001111110'.

Counting the number of edges before the next hurdle after the ith vertex is equivalent to counting the number of 0's from the ith bit until we hit a 1. After shifting the bit-vector i bits to the left, the problem then becomes finding the position of the most significant 1 in the resulting bit-vector, which is equivalent to counting the number of trailing 0's of the reverse bit-vector.

First proposed in the paper of [49], counting the number of trailing 0's in a bit-vector can be carried out through a hash-table lookup with a perfect hash function. Assume the machine word has a length of 2^n bits. The least significant 1 of a vector b can be singled out by b ANDed with its two's complement number $\bar{b}$ (computed through $\texttt{NOT}(b)+1$). For example for a machine of word size of $2^3 = 8$ bits, the least significant 1 of a vector $b = 01001000$ can be singled out by $b \texttt{ AND } \bar{b}$ which is $b_{\texttt{LSB}} = 01001000 \texttt{ AND } (10110111 + 1) = 01001000 \texttt{ AND } 10111000 = 00001000$ (LSB stands for least significant bit). Then the number of trailing 0's can be computed by multiplying $b_{\texttt{LSB}}$ with a pre-computed de Bruijn sequence $\texttt{dB}_{\texttt{seq}}$ of 2^n bits (in our example, $n = 3$ and subsequently $\texttt{dB}_{\texttt{seq}} = 00011101$). Because $b_{\texttt{LSB}}$ must be power of two, $b_{\texttt{LSB}} \times \texttt{dB}_{\texttt{seq}}$ essentially translates to shifting $b_{\texttt{LSB}}$ to the left m times, where m is the number of trailing zeros. By taking the most significant n bits of the product (carried out through shifting the product to the right $2^n - n$ bits), we have then produced a unique number, a $\texttt{key}$, between [0, … ,$2^n - 1$]. Finally, we can use the $\texttt{key}$ to query a pre-computed lookup table of 2^n entries, which returns the pre-computed number of trailing 0's in $b_{\texttt{LSB}}$. The example lookup table for $\texttt{dB}_{\texttt{seq}} = 00011101$ is provided below in Table 3.1.

000	0
001	1
010	6
011	2
100	7
101	5
110	4
011	3

Table 3.1: The de Bruijn sequence LSB lookup table for 8 bit words.

The pseudo code of finding the next hurdle is shown in Algorithm 6.

3.3.6 LEAP Variant for Affine Gap Penalty

Similar to the Needleman-Wunsch, LEAP can also be modified to more efficiently support affine gap penalties using separate insertion I and deletion D matrices. Instead of tracking leaps from

Algorithm 6: Vertices To Hurdle

```
Input: $l$; start_pos
Output: V_num
Internal Variable: bit_vec: hurdle encoded binary bit-vectors; $dB_{seq}$: de Bruijn sequence
  of $2^n$ bits
Functions: reverse_bits(bitvec): reversing the bit-vector sequence
lookup(key): lookup the precomputed de Bruijn sequence table
Pseudocode:
// Initialization
shift_bit_vec = bit_vec[$l$] $\ll$ start_pos;
rev_bit_vec = reverse_bits(shift_bit_vec);
b_LSB = rev_bit_vec $\wedge(\neg$(rev_bit_vec) $+ 1)$;
key = (b_LSB $\times$dB$_{seq}$) $\gg 2^n - n$;
V_num = lookup(key)
```

all possible lanes, with affine gap penalty we maintain an I and a D matrix separately for each lane to track the furthest column the toad can reach to from a leap, under different energy costs. Specifically, $I[l][e]$ and $D[l][e]$ stores the furthest column that the toad can arrive to, from an upward or downward leap, respectively, while consuming precisely e energy.

With I and D arrays, we modify the core recurrence function as follows:

$$\mathtt{I}[l][e] = \max \begin{cases} \mathtt{I}[l][e - \mathtt{gap\_ext\_cost}] + F(l-1, l) \\ \mathtt{end}[l][e - \mathtt{gap\_open\_cost}] + F(l-1, l) \end{cases}$$

$$\mathtt{D}[l][e] = \max \begin{cases} \mathtt{D}[l][e - \mathtt{gap\_ext\_cost}] + F(l+1, l) \\ \mathtt{end}[l][e - \mathtt{gap\_open\_cost}] + F(l+1, l) \end{cases}$$

$$\mathtt{start}[l][e] = \max \begin{cases} \mathtt{I}[l][e] \\ \mathtt{D}[l][e] \\ \mathtt{end}[l][e - \mathtt{mismatch\_cost}] + 1 \end{cases}$$

$$\mathtt{end}[l][e] = \mathtt{start}[l][e] + \mathtt{VtH}(l, \mathtt{start}[l][e])$$

where $F(l-1, l) = 0$ if lane l is above the center lane and $F(l-1, l) = 1$ otherwise, and $F(l+1, l) = 0$ if lane l is below the center lane and $F(l-1, l) = 1$ otherwise.

3.4 Results

We implemented LEAP for both banded edit distance and banded affine gap penalties. For each scoring scheme, we compare LEAP against three state-of-the-art approximate string matching implementations, including: an in-house vanilla Landau-Vishkin implementation (LV); an implementation of Gene Myer's bit-vector algorithm from SeqAn (SeqAn) ([26]) and finally a SIMD implementation of banded global Needleman-Wunsch algorithm (NW-SIMD) ([23]). Additionally, in order to benchmark the benefit of the de Brujin sequence-based bit-vector optimization, we implemented two versions of LEAP: one with (LEAP-BV) and one without (LEAP), the bit-vector optimization.

To benchmark the performance of the above implementations, we augmented a popular aligner, bowtie2, to dump all read and reference pairs into a separate file as ASCII string pairs, during the mapping procedure. For comprehensiveness, we gathered reads from six read files from the 1000 Genomes Project ([1]), ERR240726_1, ERR240726_2, ERR240727_1, ERR240727_2, ERR240728_1, ERR240728_1. Each read file is mapped against the human reference genome version 37 with bowtie2 under default settings. All reads in the above read files are 100-bp long. For banded edit distance, we benchmarked all six implementations with different edit-distance thresholds E ranging from 1 to 5. For banded affine gap penalties, we set the matching score as 0; the mismatch penalty as +2; gap open penalty as +3 and gap extend penalty as +1. We set the total affine gap penalty threshold to be $3 \times E$.

Finally, we conducted two separate tests. In the first test, shown in Table 3.2, we benchmarked all read and reference pairs from bowtie2 on all 6 implementations. In the second test, shown in Table 3.3, we only selected read-reference pairs that have at most five edits. While the first test evaluates the performance of different implementations under a realistic mapper environment, the second test evaluates how fast can each implementation find the optimal alignment in a highly similar string pair.

From both tables, we can observe that LEAP-BV is the fastest in both edit distance setup and affine gap setup. For edit distance, compared to SeqAn, LEAP-BV achieves up to **7.4x** speedup under $E = 1$ and **1.6x** speedup under $E = 5$. For affine gap, compared to NW-SIMD, LEAP-BV achieves even greater performance, with up to **32x** speedup under $E = 1$ and **2.3x** speedup under $E = 5$. Notice that even though both vanilla LV and SeqAn are reasonably fast under edit distance settings, neither support affine gap penalties due to their tight coupling with edit distance scores.

Furthermore, we observe that the performance of LV and LEAP are very similar. This is expected since under edit distance scores, LEAP reduces to LV. We also observe that the performance of both LEAP and LEAP-BV decreases with increasing E. This is also expected, since under a greater E, LEAP checks more lanes and iterates through more energy levels. We also notice that both LV and LEAP run slightly slower in Table 3.3 than Table 3.2. This is because Table 3.3 benchmarks highly similar string pairs, many of which contain long segments of identical substrings. As a result, for both LV and LEAP, the loop searching for the next hurdle runs longer. However, since LEAP-BV computes the length of the identical subustrings in bit-vectorized fashion, the speed of LEAP-BV between the two tables remains consistent.

Last but not the least, compared to LEAP, LEAP-BV provides on average 39% improvement.

Overall, LEAP performs best under small edit-distance thresholds, while its performance quickly decreases as the edit-distance threshold increases. Nonetheless, from our experiments, LEAP-BV is still faster than other implementations even under moderate edit-distances.

3.5 Discussion

While we required the toad to never move backwards while it leaps in the original definition of the Leaping Toad Problem, this requirement is not a necessity for LEAP. Both Theorem 1 and

	e	**LEAP-BV**	LEAP	LV	NW-SIMD	SeqAn
Edit Distance	1	0.84	1.28	1.25	38.81	6.21
	2	1.48	2.07	2.03	38.91	6.66
	3	2.39	3.14	3.11	38.61	6.28
	4	3.35	4.50	4.45	38.38	6.59
	5	4.60	6.01	6.04	38.46	6.88
Affine Gap	1	1.18	1.72	*N/A*	38.75	*N/A*
	2	3.10	4.01	*N/A*	38.15	*N/A*
	3	6.39	7.77	*N/A*	38.64	*N/A*
	4	10.91	13.31	*N/A*	38.66	*N/A*
	5	16.91	20.74	*N/A*	38.51	*N/A*

Table 3.2: Runtime for a suite of Approximate String Matching algorithms normalized to seconds per 10 million read/reference pairs from bowtie2.

	e	**LEAP-BV**	LEAP	LV	NW-SIMD	SeqAn
Edit Distance	1	0.84	1.31	1.33	38.55	5.81
	2	1.54	2.15	2.14	38.77	6.19
	3	2.36	3.31	3.42	38.84	6.52
	4	3.39	4.66	4.73	38.08	6.93
	5	4.55	6.22	6.21	38.63	7.73
Affine Gap	1	1.20	1.78	*N/A*	38.46	*N/A*
	2	3.09	4.11	*N/A*	38.75	*N/A*
	3	6.35	7.92	*N/A*	38.75	*N/A*
	4	10.96	13.46	*N/A*	38.86	*N/A*
	5	16.91	21.13	*N/A*	38.42	*N/A*

Table 3.3: Runtime for a suite of Approximate String Matching algorithms normalized to seconds per 10 million highly similar read/reference pairs.

Theorem 2 hold true even if the toad is allowed to move backward as it leaps. The key premise in proving both theorems is that all leaps and hurdles cost positive amounts of energy while moving forward without running into a hurdle costs zero energy.

However, when the toad is allowed to move backward during a leap, the naïve column-by-column dynamic programming method stops working, since a toad could leap from a "future column" that has not yet been calculated. LEAP, on the other hand, remains intact and functional under such conditions. This makes LEAP a broader solution for a more general Leaping Toad problem compared to the naïve dynamic programming method.

A major limitation of LEAP, in spite of its advantages, is that it cannot handle both negative energy bonuses and positive penalty schemes. In terms of approximate string matching, this translates to not supporting negative scores for matches along with positive penalties for mismatches and gaps (or vice versa). It only supports positive penalties for mismatches and gaps with no penalty/bonus for matches. As a result, LEAP cannot handle local alignment.

Nonetheless, LEAP shows great potential to be composed with NGS mappers where seed-and-extend methods are often used and strings are often compared with global or semi-global alignment.

Overall, LEAP provides three major benefits:

- LEAP reduces the frequency of calling the recurrence function, from $\mathcal{O}(E \times N)$ times to $\mathcal{O}(E^2)$ times (for the edit-distance case).
- LEAP incorporates a de Brujin sequence-based hash-table optimization, which further speeds up the computation of the Leaping Toad problem.
- LEAP enables greater parallelization in solving global and semi-global alignment problems.

Unlike traditional Needleman-Wunsch and Smith-Waterman parallel implementations, which focus on exploiting parallelism between elements on the same anti-diagonal line in the dynamic programming matrix, LEAP enables a more efficient parallelization approach. As we have discussed in the de Bruijn sequence optimization subsection, LEAP employs hurdle-encoded bit-vectors to calculate the position of the next hurdle. In the realm of Approximate String Matching, the bit-vector of each lane is simply the letter-wise XOR between the two strings after shifts. For the center lane, the bit-vector is indeed the letter-wise XOR between the pattern and the reference string; while for the ith lane above (or below) the center lane is the same letter-wise XOR but after shifting the pattern (or the reference) to the right i times. Given that we can further encode each letter with $\log_2(\sigma)$ bits, the entire operation of preparing all bit-vectors can be done in $(\frac{E \times L \times \log_2(\sigma)}{w})$ XORs, where w is the length of the machine word in bits. Under the *parallel random-access machine model (PRAM)*, all the XORs can be calculated in parallel for lanes under the same energy budget e. Lanes with different energy budgets however share dependency. Nonetheless, since the outer loop only iterates up to E times, (compared to the parallel Smith-Waterman or Needleman-Wunsch, whose outer loop is often iterated L times) LEAP still can provide a greater parallel speedup under the PRAM model.

3.6 Conclusion

Approximate string matching is an important and widely studied problem, and its use in critical components for a large number of applications has created a need to develop faster, efficient, and highly parallelizable solutions.

In this work, we analyzed existing approximate matching algorithms such as the Smith-Waterman and Needleman-Wunsch algorithms. We reviewed the Landau-Vishkin algorithm, an fast method for calculating edit distance. We then proposed the Leaping Toad problem, a generalization of the approximate string matching problem, as well as LEAP, a generalization of the Landau-Vishkin algorithm that solves the Leaping Toad problem under a broader selection of scoring schemes. We provided a detailed proof that LEAP solves the Leaping Toad problem.

We compared LEAP against 3 state-of-the-art approximate string matching implementations. We showed that when using a bit-vectorized de Bruijn sequence-based optimization, LEAP achieved a 7.4x speedup over the state-of-the-art bit-vector edit distance implementation and was up to 32x faster than the state-of-the-art affine-gap-penalty parallel Needleman Wunsch Implementation.

Part II

Improved Seeding Mechanisms

Chapter 4

Optimal Seed Solver: Optimizing Seed Selection

4.1 Background

As stated in Chapter 1, reducing total seed frequency is the key in improving mapper performance. Therefore, to build a fast yet error tolerant mapper with high mapping coverage, reads need to be divided into multiple, infrequently occurring seeds. In this way, a mapper can find all correct mappings of the read (mappings with small edit-distances) while minimizing the number of edit-distance calculations that need to be performed. To achieve this goal, we have to overcome two major challenges: (1) seeds are short, in general, and therefore frequent in the genome; and (2) the frequencies of different seeds vary significantly. We discuss each challenge in detail.

Assume a read has a length of L base-pairs (bp) and $x\%$ of it is erroneous (e.g., $L = 80$ and $x\% = 5\%$ implies that there are 4 edits). To tolerate $x\% \cdot L$ errors in the read, we need to select $x\% \cdot L + 1$ seeds, which renders a seed to be $L \div (x\% \cdot L + 1)$-base-pair long on average. Given that the desired error rates for many mainstream mappers have been as large as 0.05, the average seed length of a hash-table based mapper is typically not greater than 16-bp [3, 4, 71, 84, 100].

Seeds have two important properties: (1) the frequency of a seed is monotonically non-increasing with larger seed length and (2) frequencies of different seeds typically differ (sometimes significantly) [43]. Figure 4.1 shows the static distribution of frequencies of 10-bp to 15-bp fixed-length seeds from the human reference genome (GRCh37). This figure shows that the average seed frequency decreases with the increase in the seed length. With longer seeds, there are more patterns to index the reference genome. Thus each pattern, on average, is less frequent.

From Figure 4.1, we can also observe that the frequencies of seeds are not evenly distributed: for seeds with lengths between 10-bp to 15-bp, many seeds have frequencies below 100. As the figure shows, a high number of unique seeds, often over 10^3, correspond to seed frequencies below 100. However, there are also a few seeds which have frequencies greater than 100K (note that such unique seeds are very few, usually 1 per each frequency). This explains why most plots in Figure 4.1 follow a bimodal distribution; except for 10-bp seeds and perhaps 11-bp seeds, where the frequency of seeds peaks at around 100. Although ultra-frequent seeds (seeds that appear more frequently than 10^4 times) are few among all seeds, they are ubiquitous in the

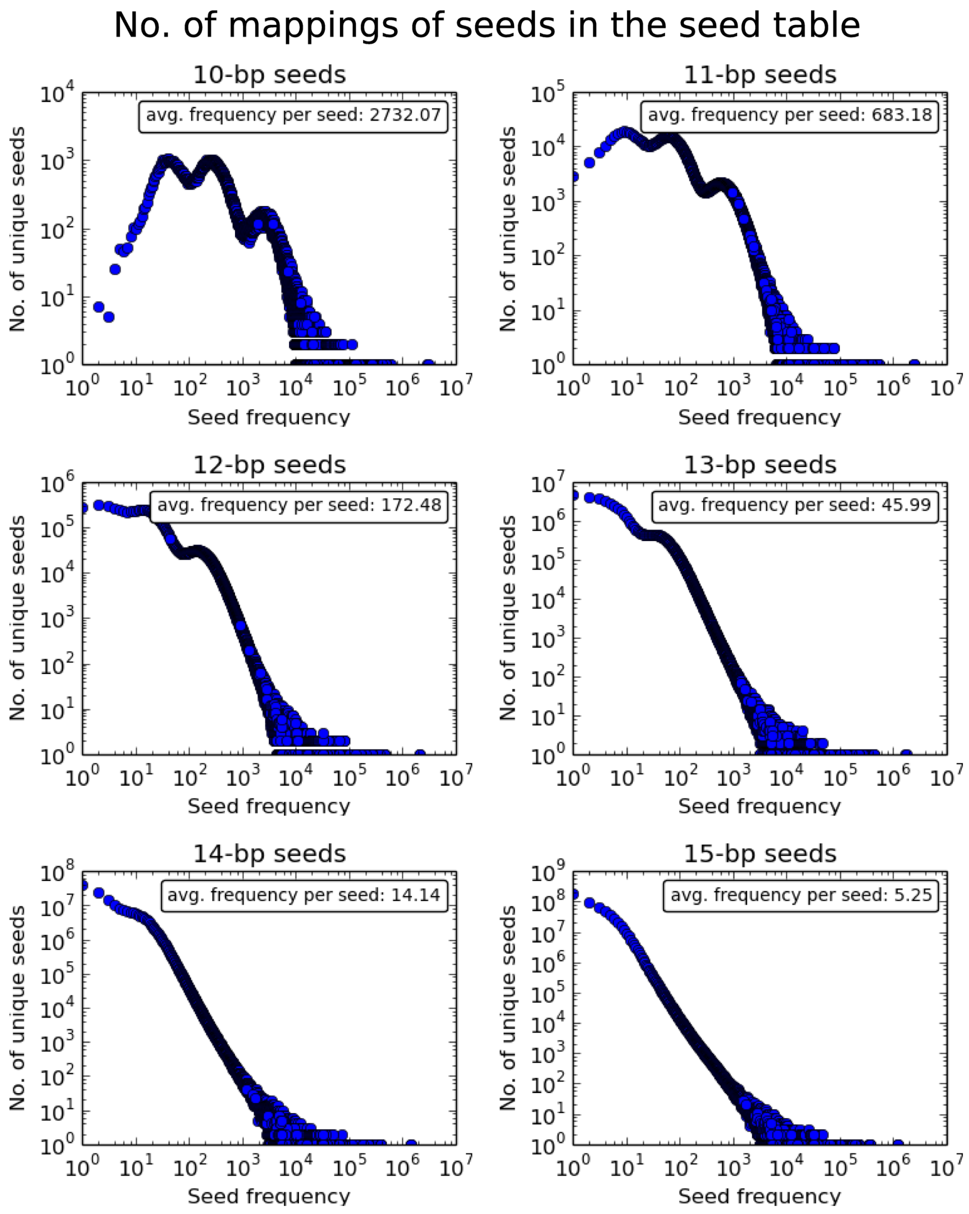


Figure 4.1: Frequency distribution of **unique** seeds in fixed-length seed (k-mers at varying 'k's) databases of human reference genome.

genome. As a result, for a randomly selected read, there is a high chance that the read contains one or more of such frequent seeds. This effect is best illustrated in Figure 4.2, which presents the frequencies of consecutively selected seeds, when we map over 4 million randomly selected 101-bp reads from the 1000 Genomes Project [1] to the human reference genome.

Unlike Figure 4.1, in which the average frequency of 15-bp seeds is 5.25, the average frequencies of selected seeds in Figure 4.2 are all greater than 2.7K. Furthermore, from Figure 4.2, we can observe that the ultra-frequent seeds are selected far more often than some of the less frequent seeds, as the selected seed count increases with seed frequencies higher than 10^4 (as

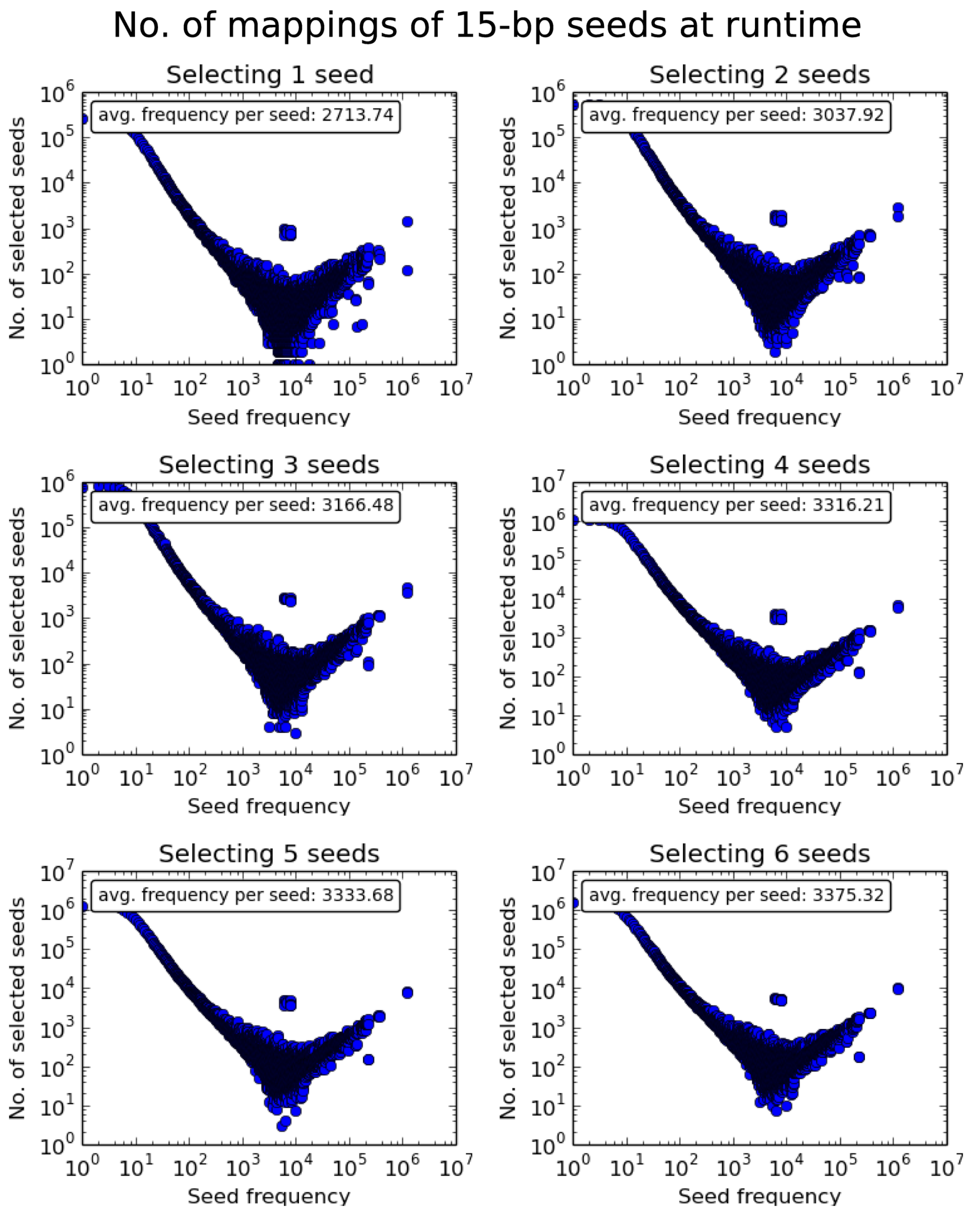


Figure 4.2: Frequency distribution of selected seeds at runtime by consecutively selecting 15-bp seeds from reads while mapping 4,031,354 101-bp reads from read set ERR240726.

opposed to Figure 4.1, where seed frequencies over 10^4 usually have seed counts below 10). This observation suggests that the ultra-frequent seeds are numerous in reads, especially considering how few ultra-frequent seed patterns there are in total in the seed database.[1] We call this phenomenon the *frequent seed phenomenon*. The frequent seed phenomenon is explained in previous works [43]. To summarize, highly frequent seed patterns are ubiquitous in the genome, therefore they appear more often in randomly sampled reads, such as reads sampled from shotgun sequencing.

[1]And the plots in Figure 4.2 no longer follow a bimodal distribution as in Figure 4.1.

Figure 4.3 through Figure 4.7 show seed frequency distributions of fixed-length seeds from 10-bp to 14-bp. From these figures, we make three observations: (1) longer selected seeds have smaller average frequencies, (2) in all figures, seed selections frequency increases again after 10^4 and (3) for all seed lengths, compared to Figure 4.1, the average frequencies of selected seeds from mapping a real read set are much larger than the average frequencies of unique seeds in the seed database.

As shown in all five figures above, after the seed frequency of 10^4, the number of selected seeds increases with greater seed frequencies, which implies that frequent seeds are often selected from reads, regardless of the seed length.

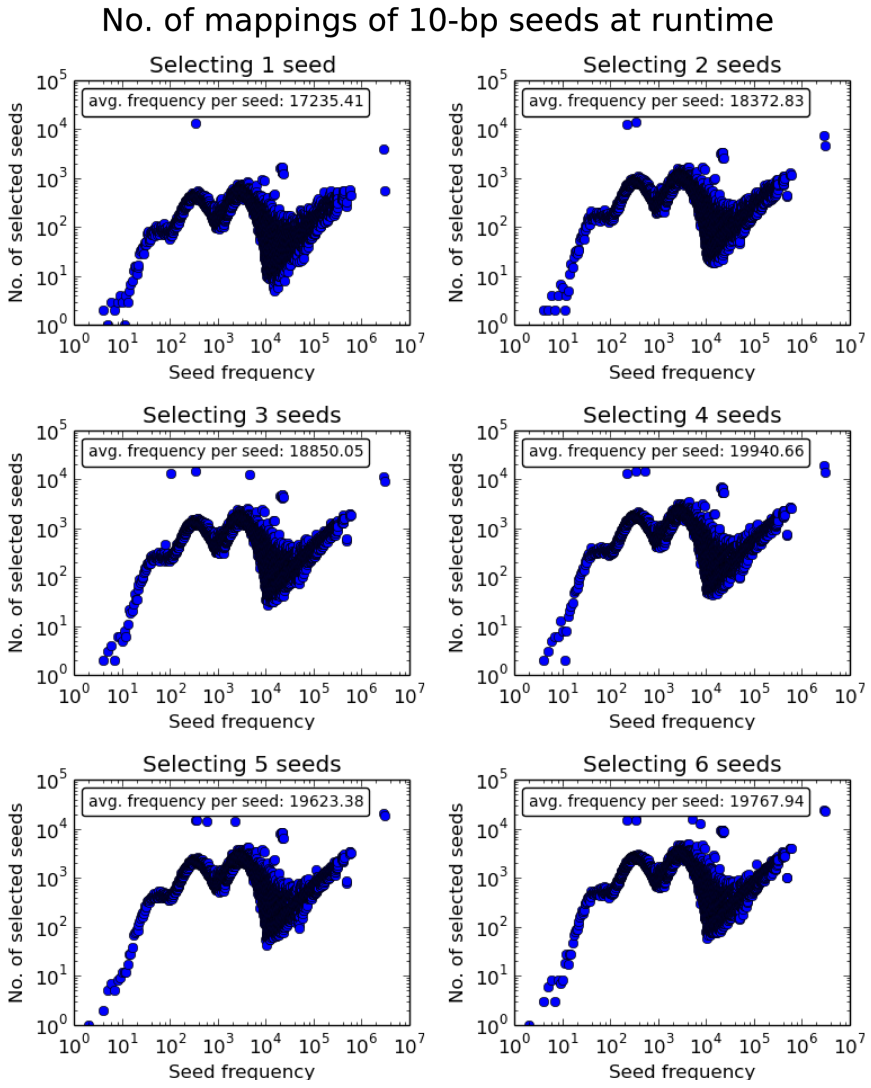


Figure 4.3: Frequency distribution of 10-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

The key takeaway from Figure 4.1 to Figure 4.7 is that although longer seeds on average are less frequent than shorter seeds, some seeds are still much more frequent than others and such more frequent seeds are very prevalent in real reads. Therefore, with a naïve seed selection

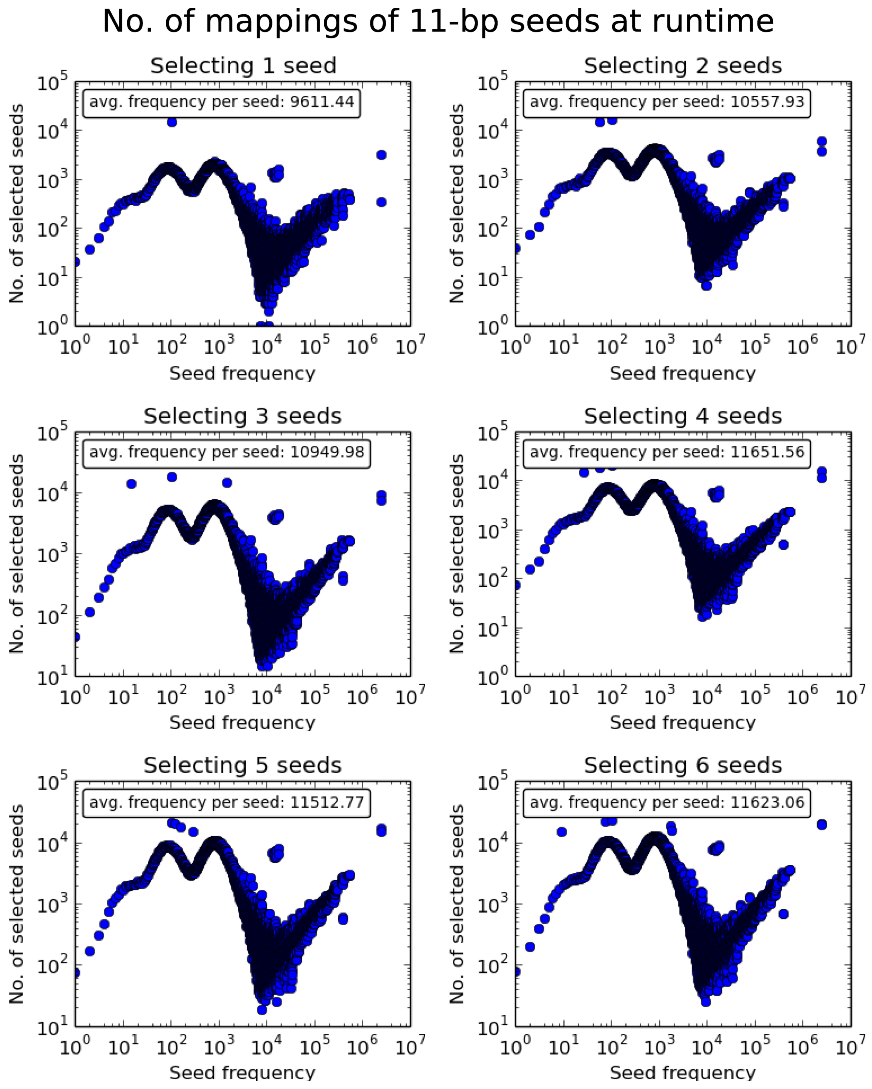


Figure 4.4: Frequency distribution of 11-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

mechanism (e.g., selecting seeds consecutively from a read), a mapper selects many frequent seeds, which increases the number of calls to the computationally expensive verification process during read mapping.

To reduce the total frequency of selected seeds, we need an intelligent seed selection mechanism to avoid using frequent patterns as seeds. More importantly, as there is a limited number of base-pairs in a read, we need to carefully choose the length of each seed. Extension of an infrequent seed does not necessarily provide much reduction in the total frequency of all seeds, but it will "consume" base-pairs that could have been used to extend other more frequent seeds. Besides determining individual seed lengths, we should also intelligently select the position of each seed. If multiple seeds are selected from a small region of the read, as they are closely packed together, seeds are forced to keep short lengths, which could potentially increase their seed frequency. Thus, seed selection must be done carefully to minimize total frequency of seed

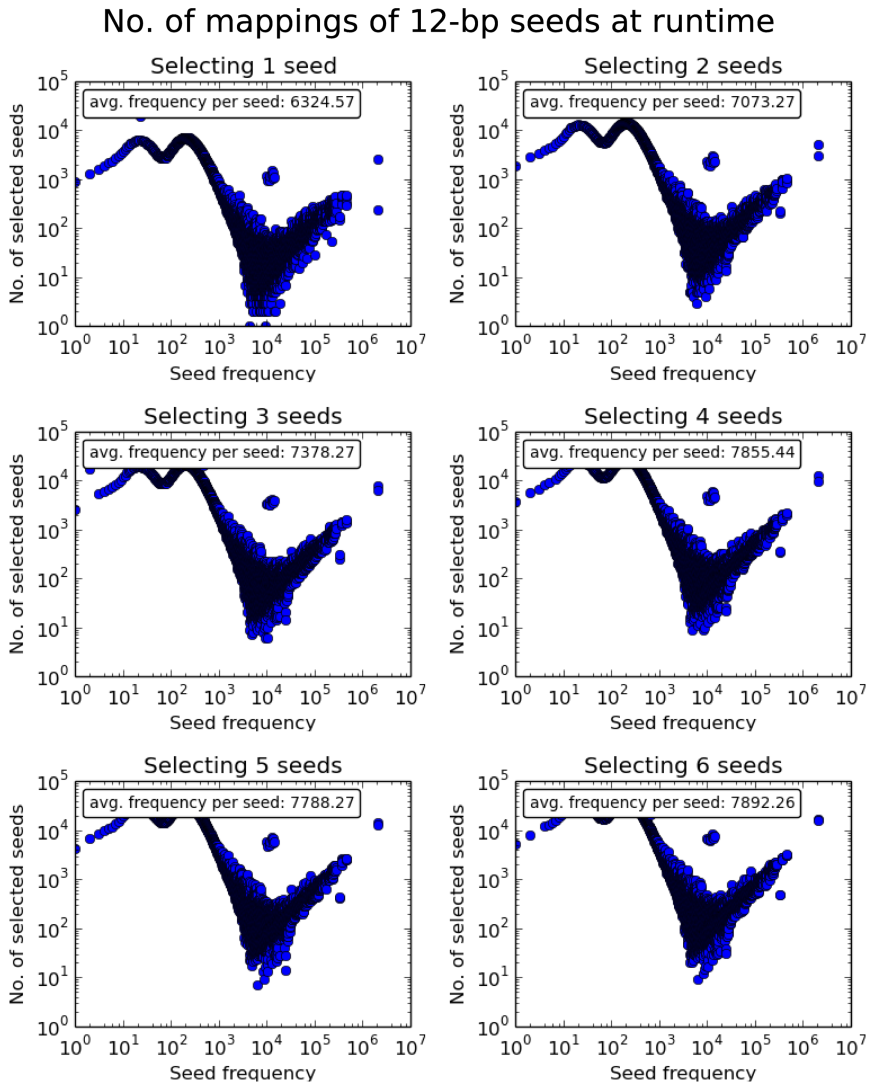


Figure 4.5: Frequency distribution of 12-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

occurrence.

Selecting the optimal set of non-overlapping seeds (i.e. the least frequent set of seeds) from a read is difficult primarily because the associated search space (all valid choices of seeds) is large and it grows exponentially as the number of seeds increases. A seed can be selected at any position in the read with any length, as long as it does not overlap with other seeds.

Based on the above observations, our goal in this work is to develop an algorithm that can calculate both the length and the placement of each seed in the read such that the total frequency of all seeds is minimized. We call such a set of seeds the *optimal seeds* of the read as they produce the minimum number of potential mappings to be verified while maintaining the sensitivity of the mapper. We call the sum of frequencies of the optimal seeds the *optimal frequency* of the read.

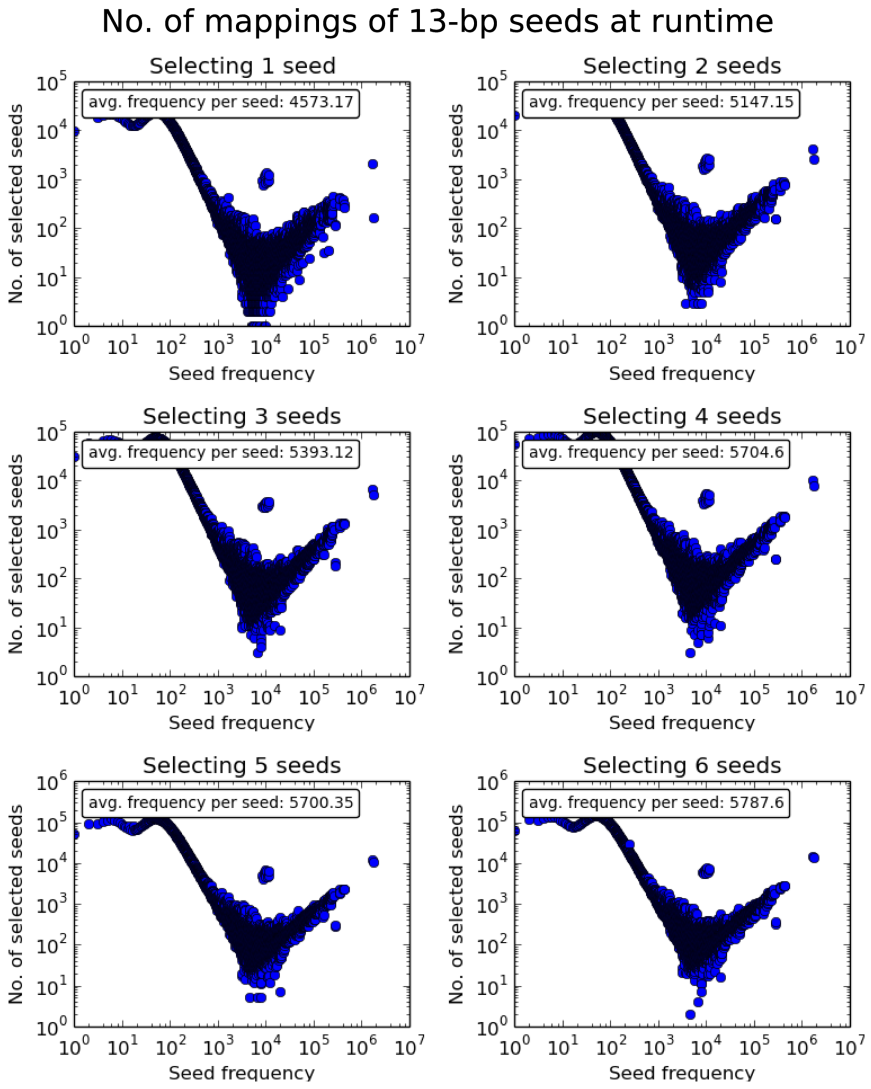


Figure 4.6: Frequency distribution of 13-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

4.2 Contributions

This work makes the following contributions:

- It examines the frequency distribution of seeds in the seed database and provides how often seeds of different frequencies are selected using a naïve seed selection scheme. We confirm the discovery of prior works [43] that frequencies are not evenly distributed among seeds and frequent seeds are selected more often under a naïve seed selection scheme. We further show that this phenomenon persists even when using longer seeds.
- It provides an implementation of an optimal seed finding algorithm, **Optimal Seed Solver**, which uses dynamic programming to efficiently find the least-frequent non-overlapping seeds of a given read. We prove that this algorithm always provides the least frequently-

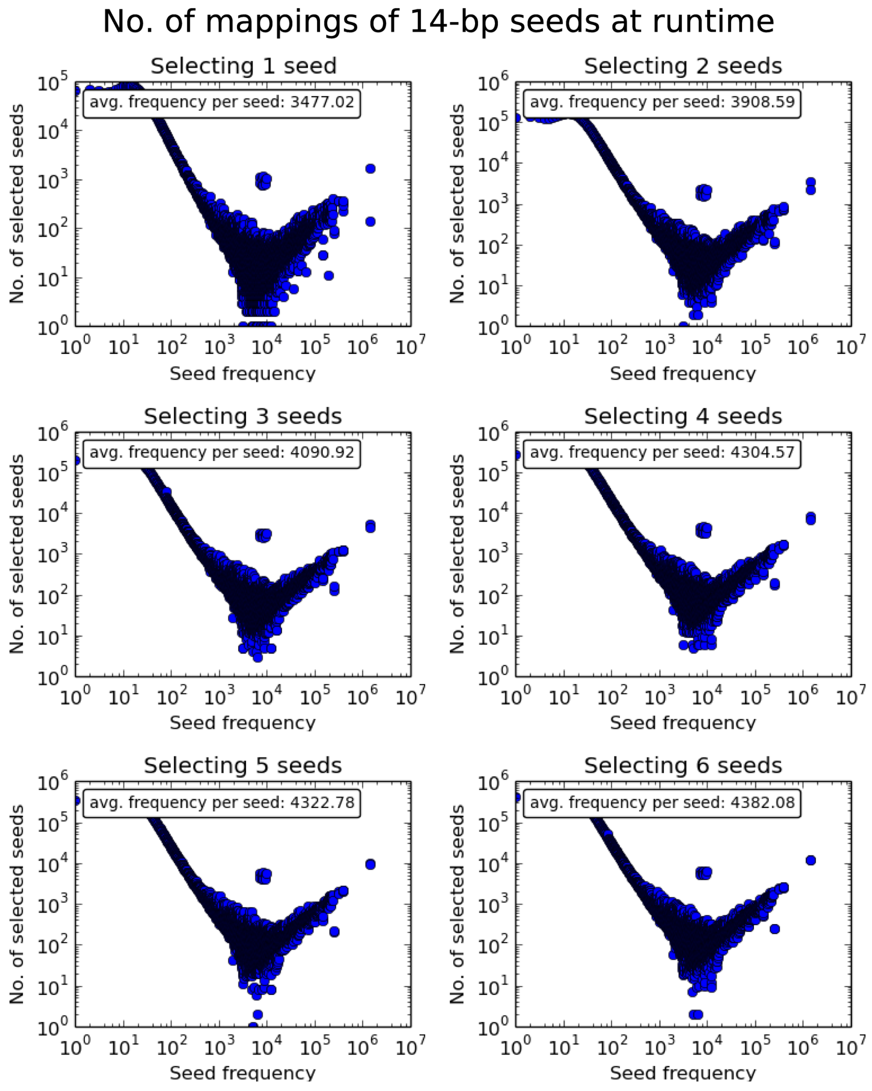


Figure 4.7: Frequency distribution of 14-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

occurring set of seeds in a read.

- It provides a comparison of the Optimal Seed Solver and existing seed selection optimizations, including Adaptive Seed Filter in the GEM mapper [71], Cheap K-mer Selection in FastHASH [102], Optimal Prefix Selection in the Hobbes mapper [3] and spaced seeds in PatternHunter [67]. We compare the complexity, memory traffic, and average frequency of selected seeds of Optimal Seed Solver with the above four state-of-the-art seed selection mechanisms. We show that the Optimal Seed Solver provides the least frequent set of seeds among all existing seed selection optimizations at reasonable complexity and memory traffic.

4.3 Methods

The biggest challenge in deriving the optimal seeds of a read is the large search space. If we allow a seed to be selected from an arbitrary location in the read with an arbitrary length, then from a read of length L, there can be $\frac{L\cdot(L+1)}{2}$ possibilities to extract a single seed. When there are multiple seeds, the search space grows exponentially since the position and length of each newly selected seed depend on the positions and lengths of all previously selected seeds. For x seeds, there can be as many as $\mathcal{O}(\frac{L^{2\cdot x}}{x!})$ seed selection schemes.

Below, we propose **Optimal Seed Solver** (OSS), a dynamic programming algorithm that finds the optimal set of x seeds of a read in $\mathcal{O}(x \cdot L)$ operations on average and in $\mathcal{O}(x \cdot L^2)$ operations in the worst case.

Although in theory a seed can have any length, in OSS, we assume the length of a seed is bounded by a range $[S_{min}, S_{max}]$. This bound is based on our observation that, in practice, neither very short seeds nor very long seeds are commonly selected as optimal seeds. Ultra-short seeds ($<$ 8-bp) are too frequent. Most seeds shorter than 8-bp have frequencies over 1000. Ultra-long seeds "consume" too many base-pairs from the read, which shorten the lengths of other seeds and increase their frequencies which leads to a higher total seed frequency. Furthermore, long seeds (e.g., 40-bp) are mostly either unique or non-existent in the reference genome.[2] Extending a unique or non-existent seed longer provides little benefit while "consuming" extra base-pairs from the read.

Bounding seed lengths reduces the search space of optimal seeds. However, it is not essential to OSS. OSS can still work without seed length limitations (to lift the limitations, one can simply set $S_{min} = 1$ and $S_{max} = L$), at the cost of extra computation.

We describe our Optimal Seed Solver algorithm in three sections. First, we introduce the core algorithm of OSS (Section 4.3.1). Then we improve the algorithm with four optimizations (Section 4.3.2), *optimal divider cascading*, *early divider termination*, *divider sprinting,* and *optimal solution forwarding*. Finally, we explain the overall algorithm and provide the pseudo-code (Section 4.3.3).

4.3.1 The Core Algorithm

A naïve brute-force solution to find the optimal seeds of a read would systematically iterate through all possible combinations of seeds. We start by selecting the first seed, by instantiating all possible positions and lengths of the seed. On top of each position and length of the first seed, we instantiate all possible positions and lengths of the second seed that is sampled after (to the right-hand side of) the first seed. We repeat this process for the rest of the seeds until we have sampled all seeds. For each combination of seeds, we calculate the total seed frequency and find the minimum total seed frequency among all combinations.

The key problem in the brute-force solution above is that it examines many obviously-suboptimal combinations. For example, in Figure 4.8, there are two 2-seed combinations, S_A and S_B, extracted from the same read, R. Both combinations end at the same position, p, in R. We call S_A and S_B *seed subsets* of the partial read $R[1..p]$. In this case, Among S_A and S_B, S_B

[2]Note that a seed with 0 frequency is still useful in read mapping as it confirms there exist at least one error in it.

has a higher total seed frequency than S_A. For any number of seeds that is greater than 2, we know that in the final optimal solution of R, seeds before position p will not be exactly like S_B, since any seeds that are appended after S_B (e.g., S'_B in Figure 4.8) can also be appended after S_A (e.g., S'_A in Figure 4.8) and produce a smaller total seed frequency. In other words, compared to S_B, only S_A has the potential to be part of the optimal solution and worth appending more seeds after. In general, among two combinations that have equal numbers of seeds and end at the same position in the read, only the combination with the smaller total seed frequency has the potential of becoming part of a bigger optimal solution (with more seeds). Therefore, for a partial read and all combinations of subsets of seeds in this partial read, only the optimal subset of this partial read (with regard to different numbers of seeds) might be relevant to the optimal solution of the entire read. Any suboptimal subset of seeds of this partial read (with regard to different numbers of seeds) is guaranteed to not lead to the optimal solution and should be pruned.

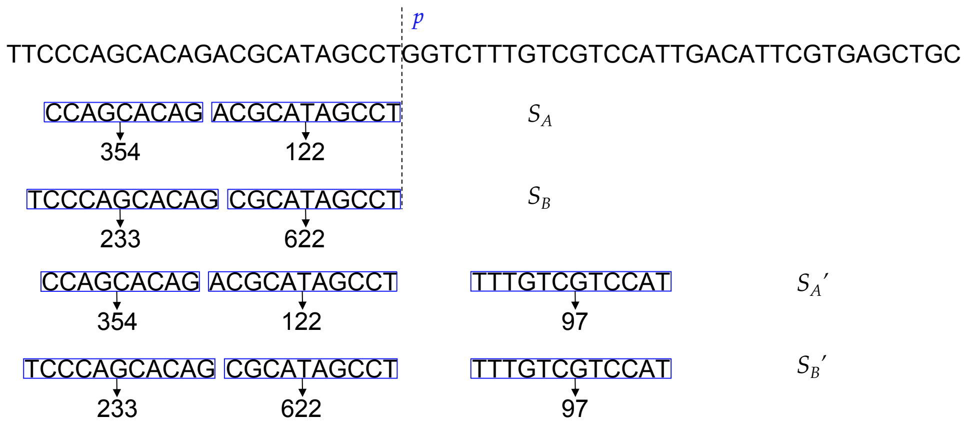


Figure 4.8: Example showing optimal seeds in a substring must be optimal seeds of the substring.

The above observation suggests that by summarizing the optimal solutions of partial reads under a smaller number of seeds, we can prune the search space of the optimal solution. Specifically, given m (with $m < x$) seeds and a substring U, only the optimal m seeds of U could be part of the optimal solution of the entire read. Any other suboptimal combinations of m seeds of U should be pruned.

Storing the optimal solutions of partial reads under a smaller number of seeds also helps speed up the computation of larger numbers of seeds. Assuming we have already calculated and stored the optimal frequency of m seeds of all substrings of R, to calculate the optimal $(m + 1)$-seed solution of a substrings, we can 1) iterate through a series of divisions of this substring; 2) calculate the seed frequency of each division using pre-calculated results and 3) find out the division that provides the minimum seed frequency. In each division, we divide the substring into two parts: We extract m seeds from the first part and 1 seed from the second part. The minimum total seed frequency of this division (or simply the *optimal frequency of the division*) is simply the sum of the optimal m-seed frequency of the first part and the optimal 1-seed frequency of the second part. As we already have both the optimal m-seed frequency of the first part and the 1-seed frequency of the second part pre-calculated and stored, the optimal frequency of this division can be computed with one addition and two lookups.

The *optimal* $(m + 1)$*-seed solution* of this substring is simply the division that yields the minimum total frequency. Given that each seed requires at least S_{min} base-pairs, for a substring of length L', there are in total $L' - (m + 1) \cdot S_{min}$ possible divisions to be examined. This relationship can be summarized as a recurrence function in Equation 4.1, in which $\texttt{Opt}(U, m)$ denotes the optimal m-seed frequency of substring U and u denotes the length of U.

$$\texttt{Opt}(U, m + 1) = \min_{i}[\texttt{Opt}(U[1 : i - 1], m) + \texttt{Opt}(U[i : u], 1)] \tag{4.1}$$

We can apply the same strategy to the entire read: to obtain the optimal $x + 1$ seeds from read R, we first examine all possible 2-part divisions of the read, which divide the read into a prefix and a suffix. For each division, we extract x seeds from the prefix, and 1 seed from the suffix. The optimal $(x + 1)$-seed solution of the read is simply the division that provides the lowest total seed frequency. As we have discussed above, for a division to be optimal, its x-seed prefix and 1-seed suffix must also be optimal (this provides the minimum total seed frequency). By the same logic, to obtain the optimal x-seed solution of a prefix, we can further divide the prefix into an optimal $(x - 1)$-seed prefix and an optimal 1-seed substring (which is no longer a suffix of the read). We can keep applying this prefix-division process until we have reached 1-seed prefixes. In other words, **by progressively calculating the optimal solutions of *all prefixes* from 1 to x seeds, we can find the optimal $(x + 1)$-seed solution of the read.**

OSS implements the above strategy using a dynamic programming algorithm: to calculate the optimal $(x + 1)$-seed solution of a read, R, OSS computes and stores optimal solutions of prefixes with fewer seeds through x iterations. In each iteration, OSS computes optimal solutions of prefixes with regard to a specific number of seeds. In the m^{th} iteration ($m \leqslant x$), OSS computes the optimal m-seed solutions of all prefixes of R, by re-using optimal solutions computed from the previous $(m-1)^{th}$ iteration. For each prefix, OSS performs a series of divisions and finds the division that provides the minimum total frequency of m seeds. For each division, OSS computes the optimal m-seed frequency by summing up the optimal $(m - 1)$-seed frequency of the first part and the 1-seed frequency of the second part. Both frequencies can be obtained from previous iterations. Overall, OSS starts from one seed and iterates to x seeds. Finally, OSS computes the optimal $(x + 1)$-seed solution of R by finding the optimal division of R and reuses results from the x^{th} iteration.

4.3.2 Further Optimizations

With the proposed dynamic programming algorithm, OSS can find the optimal $(x + 1)$ seeds of a L-bp read in $\mathcal{O}(x \cdot L^2)$ operations: In each iteration, OSS examines $\mathcal{O}(L)$ prefixes (to be exact, $L - (x + 1) \cdot S_{min}$ prefixes) and for each prefix OSS inspects $\mathcal{O}(L)$ divisions (to be exact, $L' - i \cdot S_{min}$ divisions of an L'-bp prefix for the i^{th} iteration). In total, there are $\mathcal{O}(L^2)$ divisions to be verified in an iteration.

We observe that it is not necessary to always exhaustively search every divider position in a prefix to find the optimal divider. For most reads, the optimal divider in each prefix can only appear at a few positions.

With the above observation, we propose four optimizations: optimal divider cascading, early divider termination, divider sprinting and optimal solution forwarding. We verify the effectiveness of above optimizations by checking if OSS evaluates fewer divider positions. We designate the empirically-derived average number of divider positions evaluated by OSS per read as *the average complexity* of OSS. With all four optimizations, we empirically reduce the average complexity of processing a read to $\mathcal{O}(L \cdot e)$. Below we describe the four optimizations in detail.

Optimal divider cascading.

Until this point, our assumption is that optimal solutions of prefixes within an iteration are independent from each other: the *optimal division* (the division that provides the optimal frequency) of one prefix is independent from the optimal division of another prefix, thus they must be derived independently.

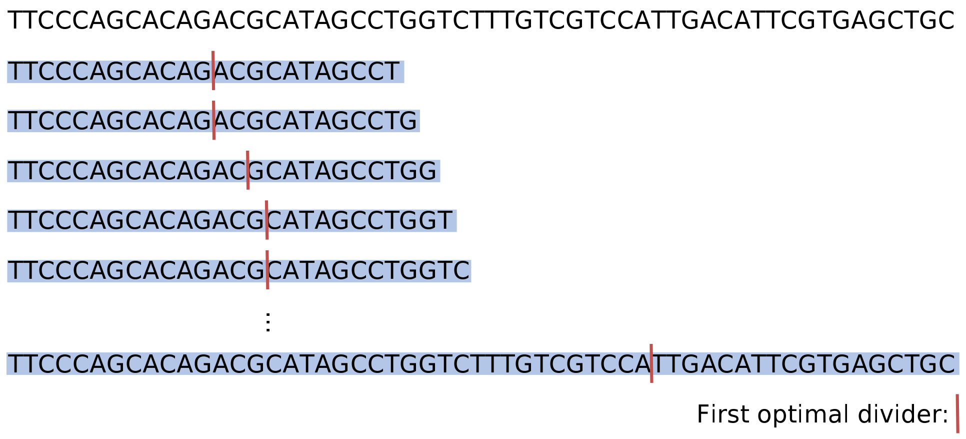

TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATTCGTGAGCTGC
TTCCCAGCACAGACGCATAGCCT
TTCCCAGCACAGACGCATAGCCTG
TTCCCAGCACAGACGCATAGCCTGG
TTCCCAGCACAGACGCATAGCCTGGT
TTCCCAGCACAGACGCATAGCCTGGTC
⋮
TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATTCGTGAGCTGC
First optimal divider:

Figure 4.9: Example of optimal divider cascading.

We observe that this assumption is not necessarily true as there exists a relationship between two prefixes of different lengths in the same iteration (under the same number of seeds): the *first optimal divider* (the optimal divider that is the closest towards the beginning of the read, if there exist multiple optimal divisions with the same total frequency) of the shorter prefix must be **at the same or a closer position towards the beginning of the read**, compared to the *first optimal divider* of the longer prefix. We call this phenomenon the *optimal divider cascading*, and it is depicted in Figure 4.9.

The optimal divider cascading phenomenon can be explained with two lemmas:

Lemma 2. *For any two prefixes from the same iteration in OSS, one prefix must include the other. Among the two prefixes, the minimum seed frequency of the outer prefix must not be greater than the minimum seed frequency of the inner prefix.*

The proof of Lemma 2 is provided below:

Proof. Since both are prefixes of the same read, one must include another, as shown in Figure 4.9.

We prove the second part of the lemma by contradiction. Assume the outer prefix has a greater *optimal frequency* (total seed frequency of the optimal seeds) than the inner prefix. Because the inner prefix is included by the outer prefix, the optimal seeds of the inner prefix are also valid seeds for the outer prefix. Yet, the total frequency of this particular set of seeds is smaller than the optimal frequency of the outer prefix, which leads to a contradiction.

$\square$

Lemma 3. *When extending two seeds of different lengths that end at the same position in the read by equal numbers of base-pairs, as one seed includes the other as shown in Figure 4.10, the frequency reduction (Δf) of extending the outer seed ($S_2 \to S_2'$) must not be greater than the frequency reduction of extending the inner seed ($S_1 \to S_1'$).*

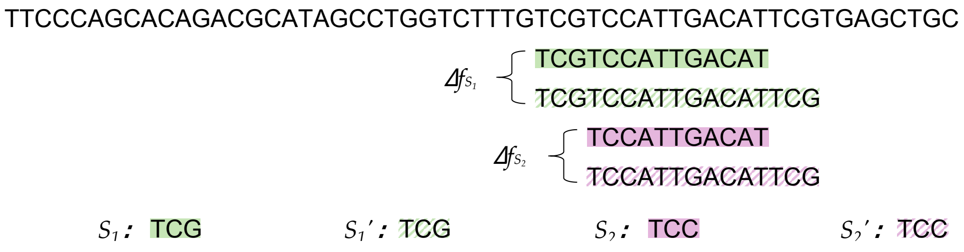


Figure 4.10: Illustration of Lemma 3 in OSS.

Lemma 3 can be proven with the monotonic non-increasing property of seed frequency with regard to a greater seed length. For example, in Figure 4.11, there are two seeds taken from the same read, S_1 and S_2, with S_1 including S_2 and both end at the same position in the read. Now, we simultaneously extend both S_1 and S_2 longer in the read (by taking more base-pairs) by 3-bp, into S_1' and S_2' respectively. With Δf denoting the change of seed frequencies before and after extension, we can claim that $\Delta f_{S_1} \leqslant \Delta f_{S_2}$.

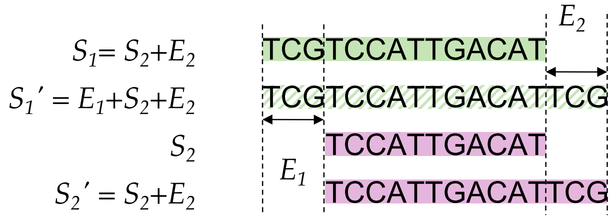


Figure 4.11: Extension of seeds S_1 and S_2.

To prove this inequality, it is essential to understand how is Δf calculated. As Figure 4.11 also shows, among the two seeds S_1 and S_2, S_1 can be considered as a "left-extension" of S_2. Therefore, S_1 can be represented as $S_1 = E_1 + S_2$, where E_1 denotes the left extension of S_1 and the "+" sign denotes a concatenation of strings. Similarly, S_1' can be represented as a "right-extension" of S_1, which can be also written as $S_1' = E_1 + S_2 + E_2$, where E_2 is the right m-bp extension of S_1. By the same token, we also have $S_2' = S_2 + E_2$. If $\texttt{freq}(S)$ denotes the frequency of a seed S, then $\Delta f_{S_1} = \texttt{freq}(S_1) - \texttt{freq}(S_1') = \texttt{freq}(E_1 + S_2) - \texttt{freq}(E_1 + S_2 + E_2)$.

Below, we provide the proof of Lemma 3:

Proof. If set $\overline{\mathbb{E}_2}$ denotes all DNA sequences that are equal in length with E_2 but excludes E_2 itself, which can be written as $\overline{\mathbb{E}_2} = \{s \mid (s \in \text{DNA sequence}) \wedge (|s| = |E_2|) \wedge (s \neq E_2)\}$, then the reduced frequency of S_1 and S_2 can also be written as:

$$\Delta f_{S_1} = \sum_{s \in \overline{\mathbb{E}_2}} \texttt{freq}(E_1 + S_2 + s)$$

$$\Delta f_{S_2} = \sum_{s \in \overline{\mathbb{E}_2}} \texttt{freq}(S_2 + s)$$

The right hand sides of both equations denote the sum of frequencies of all seeds that share the same beginning sequence $E_1 + S_2$ (or just S_2) other than the sequence $E_1 + S_2 + E_2$ itself (or $S_2 + E_2$ for S_2'), which is indeed $\texttt{freq}(E_1 + S_2) - \texttt{freq}(E_1 + S_2 + E_2)$ (or $\texttt{freq}(S_2) - \texttt{freq}(S_2 + E_2)$ for S_2).

From both equations, we can see that both Δf_{S_1} and Δf_{S_2} iterates through the same set of strings, $\overline{\mathbb{E}_2}$. For each string i in set $\overline{\mathbb{E}_2}$, we have $\texttt{freq}(E_1 + S_2 + i) \leqslant \texttt{freq}(S_2 + i)$, as the extended longer seed can only be less or equally frequent as the original and shorter seed. Therefore, we have $\Delta f_{S_1} \leqslant \Delta f_{S_2}$.

□

From Lemma 3, we can deduce Corollary 1:

Corollary 1. *When extending two substrings of different lengths that end at the same position in the read by equal number of seeds, as one substring includes the other, the frequency reduction of the optimal seed (the optimal single seed) of extending the longer substring, is strictly not greater than the frequency reduction of the optimal seed of extending the shorter substring.*

We prove Corollary 1 by cases:

Proof. Considering the four substrings from Figure 4.11, S_1, S_2, S_1' and S_2'. Among the four substrings, we have the following system of equations that describe these substring relationships:

$$\begin{aligned} S_1 &= E_1 + S_2; \\ S_1' &= E_1 + S_2 + E_2; \\ S_2' &= S_2 + E_2 \end{aligned}$$

There are three possible cases of where the optimal seed is selected in S_1': (1) from the region of $S_2 + E_2$, (2) from the region $E_1 + S_2$ and the optimal seed overlaps with E_1 and (3) from the region of $E_1 + S_2 + E_2$ and the seed overlaps with both E_1 and E_2. Below we prove that the Corollary is correct in each case.

Case 1: The optimal seed is selected exclusively from $S_2 + E_2$.

This suggests that the optimal seed in S_1' is also the optimal seed in S_2'. Based on Lemma 2, we know the optimal frequency of S_1 is not greater than S_2.

Combining the two deductions above, we can conclude that extending S_2 to S_2' provides a frequency reduction of the optimal seed that is greater than or equal to extending S_1 to S_1'.

Case 2: The optimal seed is selected from the region $E_1 + S_2$ and it overlaps with E_1.

Since the optimal seed does not overlap with E_2, the optimal seed in both S_1 and S_1' must be the same. Therefore extending S_1 to S_1' provides 0 frequency reduction of the optimal seed. As Lemma 2 suggests, the optimal seed frequency of S_2 must not be greater than the optimal seed frequency of S_2'. As the result, the Corollary holds in this case.

Case 3: The optimal seed is selected across $E_1 + S_2 + E_2$ and it overlaps with both E_1 and E_2.

Assuming that the optimal seed, s_1', in S_1' starts at position p_1 and ends at position p_2. Now assume a seed, s_1, which starts at p_1 but ends where S_1 ends, as shown in Figure 4.12. Also assume a seed, s_2', which starts at where S_2' starts and ends at p_2. From Lemma 3, we know that the reduction of seed frequency of extending s_1 to s'_1 is no greater than the seed frequency reduction of extending S_2 to s'_2. We also know that the optimal seed frequency of S_1 is no greater than the seed frequency of s_1 and the optimal seed frequency of S'_2 is no greater than the seed frequency of s'_2. As a result, the frequency reduction of the optimal seed by extending S_1 to S'_1, is strictly no greater than the frequency reduction of the optimal seed by extending S_2 to S'_2.

□

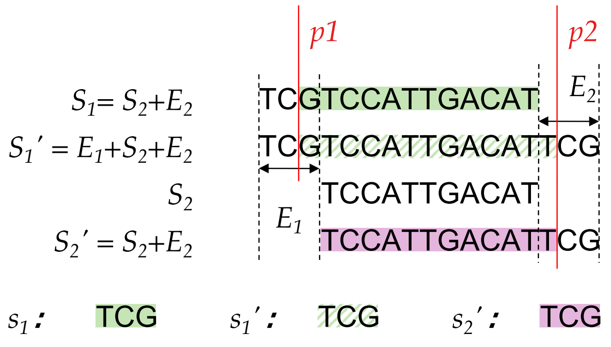


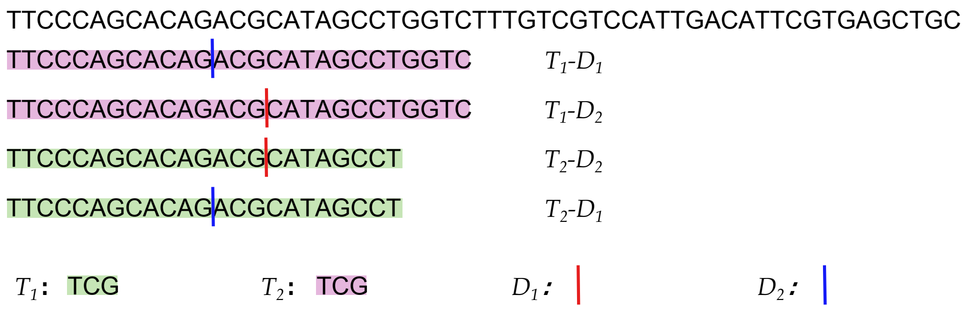
Figure 4.12: Assumed alternative optimal seeds.

Using Lemma 2, Lemma 3 and Corollary 1, we are ready to prove that the optimal divider cascading phenomenon is always true.

Theorem 1. *For two prefixes from the same iteration in OSS, as one prefix includes the other, the first optimal divider of the outer prefix must not be at the same or a prior position than the first optimal divider of the inner prefix.*

We can prove Theorem 1 by contradiction.

Proof. Assume T_1 and T_2 are two prefixes from the same iteration in "Optimal Seed Solver", with T_1 including T_2. Also assume T_1's first optimal divider, D_1, is closer to the beginning of the read than T_2's first optimal divider, D_2, as shown in Figure 4.13 ($D_1 < D_2$).



TTCCCAGCACAGACGCATAGCCTGGTCTTTGTCGTCCATTGACATTCGTGAGCTGC
TTCCCAGCACAGACGCATAGCCTGGTC
T1-D1
TTCCCAGCACAGACGCATAGCCTGGTC
T1-D2
TTCCCAGCACAGACGCATAGCCT
T2-D2
TTCCCAGCACAGACGCATAGCCT
T2-D1
T1: TCG
T2: TCG
D1:
D2:

Figure 4.13: Example of early divider termination.

Suppose we apply both divisions D_1 and D_2 to both prefixes T_1 and T_2, which renders four divisions: T_1-D_1, T_1-D_2, T_2-D_1 and T_2-D_2, as Figure 4.13 shows. We can prove that T_2-D_2 is a strictly less frequent solution than T_2-D_1. Since D_2 is the first optimal divider of T_2 and $D_1 < D_2$, the minimum frequency of dividing T_2 at D_1 must be greater than dividing T_2 at D_2.

Let $\texttt{freq}(T, D)$ denotes the optimal frequency of dividing prefix T at position D, then based on our assumptions and Lemma 3, we have the following relationships:

$$\begin{aligned}\texttt{freq}(T_1, D_1) &\leqslant \texttt{freq}(T_1, D_2)\\ \texttt{freq}(T_2, D_2) &< \texttt{freq}(T_2, D_1)\\ \texttt{freq}(T_1, D_1) &\geqslant \texttt{freq}(T_2, D_1)\\ \texttt{freq}(T_1, D_2) &\geqslant \texttt{freq}(T_2, D_2)\end{aligned}$$

Based on Corollary 1, we know that the frequency reduction of extending T_2-D_1 to T_1-D_1 is strictly not greater than the frequency reduction of extending T_2-D_2 to T_1-D_2. From Figure 4.13, we can observe that only the second parts of both T_2-D_1 and T_2-D_2 are extended into T_1-D_1 and T_1-D_2 respectively. Between T_2-D_1 and T_2-D_2, we can see that D_1 produces a longer second part than D_2. Based on the Corollary 1, the frequency reduction of extending T_2-D_2 to T_1-D_2 is no less than the frequency reduction of extending T_2-D_1 to T_1-D_1. Given that $\texttt{freq}(T_2, D_2) < \texttt{freq}(T_2, D_1)$ from above, we prove that $\texttt{freq}(T_1, D_2) < \texttt{freq}(T_1, D_1)$, which contradicts our assumption that $\texttt{freq}(T_1, D_2) \geqslant \texttt{freq}(T_2, D_2)$. Therefore, the first optimal divider of T_1 must not be at a prior position than the first optimal divider of T_2.

□

Based on the optimal divider cascading phenomenon, we know that for two prefixes in the same iteration, the first optimal divider of the shorter prefix must be no further than the first optimal divider of the longer prefix. With this relationship, we can reduce the search space of optimal dividers in each prefix by processing prefixes within an iteration from the longest to the shortest.

In each iteration, we start with the longest prefix of the read, which is the read itself. We examine all divisions of the read and find the first optimal divider of it. Then, we move to the next prefix of the length $|L-1|$. In this prefix, we only need to check dividers that are at the same or a prior position than the first optimal divider of the read. After processing the length $|L-1|$ prefix, we move to the length $|L-2|$ prefix, whose search space is further reduced to positions that are at the same or a closer position to the beginning of the read than the first optimal divider of the length $|L-1|$ prefix. This procedure is repeated until the shortest prefix in this iteration is processed.

Early divider termination.

With optimal divider cascading, we are able to reduce the search space of the first optimal divider of a prefix and exclude positions that come after the first optimal divider of the previous, 1-bp longer prefix (recall that with optimal divider cascading OSS starts with the longest prefix and gradually moves to shorter prefixes). However, the search space is still large since any divider prior to the first optimal divider of the previous prefix could be the optimal divider. To further reduce the search space of dividers in a prefix, we propose the second optimization – *early divider termination*.

The goal of early divider termination is to reduce the number of dividers we examine for each prefix. The key idea of early divider termination is to find the *early divider termination* position in the target prefix, as we are moving the divider backward one base-pair at a time, where all dividers that are prior to the termination position are guaranteed to be suboptimal and can be excluded from the search space.

The key observation that early divider termination builds on is simple: The optimal frequency of a substring monotonically non-increases as the substring extends longer in the read.

Based on optimal divider cascading, we start at the position of the first optimal divider in the previous prefix. Then, we gradually move the divider towards the beginning (or simply move backward) and check the total seed frequency of the division after each move. During this process, the first part of the division gradually shrinks while the second part gradually grows, as we show in Figure 4.14. According to Lemma 2, the optimal frequency of the first part must be monotonically non-decreasing while the optimal frequency of the second part must be monotonically non-increasing.

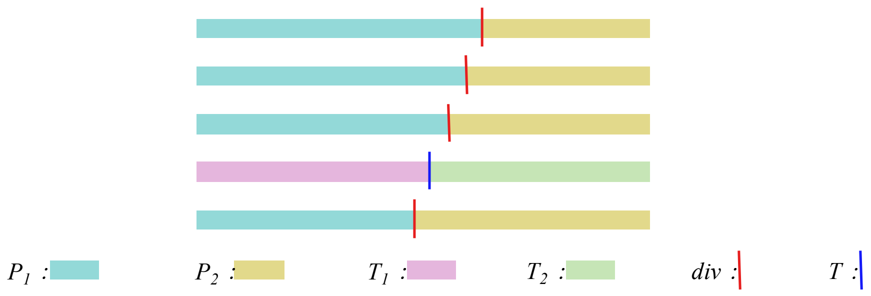


Figure 4.14: Example of combining early divider termination with optimal divider cascading.

For each position of the divider, let $\texttt{FREQ}_{P_2}$ denote the frequency of the second part (P_2, in yellow) and $\Delta\texttt{FREQ}_{P_1}$ denote the change of frequency of the first part (P_1, in blue) between current and the next move (the two moves are only 1 bp apart). Early divider termination suggests that the divider should stop moving backward, whenever $|\Delta\texttt{FREQ}_{P_1}| > |\texttt{FREQ}_{P_2}|$. All dividers that are prior to this position are guaranteed to have greater total seed frequencies. We call this stopping position the *termination position*, the division at this position – the *termination division*, denoted as T, and the above inequality that determines the termination position, the *termination inequality* ($|\Delta\texttt{FREQ}_{T_1}| > |\texttt{FREQ}_{T_2}|$). We name the first and the second part of T as T_1 and T_2, respectively.

For any divider D that comes prior to the termination position, compared to the termination division, its first part is shorter than the first part of the termination division ($|D_1| < |T_1|$) and its second part is longer ($|D_2| > |T_2|$). Hence, the optimal frequency of its first part is greater ($\texttt{FREQ}_{D_1} \geqslant \texttt{FREQ}_{T_1}$) and the optimal frequency of its second part is smaller ($\texttt{FREQ}_{D_2} \leqslant \texttt{FREQ}_{T_2}$). Let $|\Delta\texttt{FREQ}_{D_1-T_1}|$ denote the increase of the optimal frequency of the first part between current division D and the termination division T and $|\Delta\texttt{FREQ}_{D_2-T_2}|$ denote the decrease of the second part. Based on Lemma 2, we have $|\Delta\texttt{FREQ}_{D_1-T_1}| \geqslant |\Delta\texttt{FREQ}_{T_1}|$. Since the frequency of a seed can be no smaller than 0, we also have $|\texttt{FREQ}_{T_2}| \geqslant |\Delta\texttt{FREQ}_{D_2-T_2}|$. Combining these two inequalities with the termination inequality ($|\Delta\texttt{FREQ}_{T_1}| > |\texttt{FREQ}_{T_2}|$), we have $|\Delta\texttt{FREQ}_{D_1-T_1}| > |\Delta\texttt{FREQ}_{D_2-T_2}|$. This suggests that compared to the termination division, the frequency increase of the first part must be greater than the frequency reduction of the second part. Hence, the overall optimal frequency of such a division must be greater than the optimal frequency of the termination division. Therefore, a division prior to termination position cannot be optimal.

Using early divider termination, we can further reduce the search space of dividers within a prefix and exclude all positions that are prior to the termination position. Since the second part of the prefix hosts only one seed and frequencies of most seeds decrease to 1 after extending it to a length of over 20-bp, we observe that the termination position of a prefix is reached fairly quickly, only after a few moves. With both optimal divider cascading and early divider termination, from our experiments, we observe that we only need to verify 5.4 divisions on average (this data is obtained from mapping ERR240726 to human genome v37, under the error threshold of 5) for each prefix. To conclude, with both optimizations, we observe that in our experiments the empirical average complexity of our Optimal Seed Solver is reduced to $\mathcal{O}(x \cdot L)$ [3].

Divider sprinting.

According to optimal divider cascading and early divider termination, for each prefix, after inheriting the starting divider from the previous prefix, we gradually move the divider towards the beginning of the prefix, one base-pair at a time, until early divider termination is triggered. In each move, we check the optimal frequency of the two parts in the current division as well as the frequency increase of the first part compared to the previous division. We stop moving the divider when the frequency increase of the first part is greater than the optimal frequency of the second part.

We observe that it is unnecessary to always move the divider a single base-pair at a time and check for frequencies after each move. In the early divider termination method, the move terminates only when the frequency increase of the first part is greater than the optimal seed frequency in the second part. This suggests that when the frequency of the first part remains unchanged between moves, which produces no increase in frequency, we do not need to check the frequency of the second part as it will not trigger early termination. When multiple dividers in a region share the same first-part frequency, we only need to verify the last divider of this region and skip all the other dividers in the middle (an example is provided in the subsection below). The last divider always provides the least total seed frequency among all dividers in this region since it has the longest second part compared to other dividers (longer substring always provides less or equally frequent seeds) while keeping its first-part frequency the same. We call this method *divider sprinting*.

Optimal solution forwarding.

With optimal divider cascading, early divider termination and divider sprinting, we observe that the average number of divisions per prefix reduces from 5.4 (plain OSS with no optimizations) to 3.7. Nevertheless, for each prefix, we still need to examine at least two divisions (one for the inherited optimal division of the previous prefix and at least one more for early divider termination). We observe that some prefixes can also inherit the optimal solution of the previous prefix without verifying any divisions, as they share the same optimal divider with the previous prefix. Within an iteration, we recognize that there exist many prefixes that share the same second-part frequency with the previous prefix when divided by the previous prefix's optimal divider. We

[3]In the worst case, where the frequency of the second part is high and the frequency reduction is mild, the complexity can still reach $O(L^2)$, as the divider cannot terminate early.

conclude that such prefixes **must also share the same optimal divider as well as the same optimal seed frequency with the previous prefix**. We call this *optimal solution forwarding*.

From our experiment, we observe that many prefixes within the same iteration share the same optimal divider with the previous prefixes (please look at the example in the next section). Among them, most also share the same 2nd-part frequency. For such prefixes, we propose the theorem below:

Theorem 2. *A prefix that shares the same 2nd-part frequency with the previous prefix while being divided by the previous prefix's first optimal divider must have the same first optimal divider as well as the same optimal total seed frequency.*

Proof. We prove the above theorem by contradiction. Assume there exists another optimal divider, div_{new}, which is prior to the inherited optimal divider from the previous prefix, div_{prev}. Also assume div_{new} provides a solution that has either smaller or equal total seed frequency. Since the previous prefix is 1-bp longer than the current prefix, div_{new} could also be applied to the previous prefix, which generates a first part that is the same as the current prefix's first part, and a second part that is 1-bp longer than the current prefix's second part (both under div_{new}). Given that a substring always provides less or equally frequent optimal seed(s) than any of its included shorter substrings (the proof of this fact is similar to Lemma 2), we know that the second part of the previous prefix provides less or equally frequent optimal seed than the second part of the current prefix. This suggest that, under div_{new}, the previous prefix generates a total seed frequency that is smaller than or equal to the optimal total seed frequency provided by div_{prev}. As the result, div_{prev} must not be the first optimal divider of the previous prefix, which leads to a contradiction.

□

With optimal solution forwarding, for each incoming prefix, after inheriting the optimal divider from the previous prefix, we first test if the second-part frequency of the new prefix equals the second-part frequency of the previous prefix. If they are equal, then we can assert that the optimal divider of the previous prefix must also be the optimal divider of the new prefix and move on to the next read, without examining any divisions.

With optimal solution forwarding, we observe that the average number of division verifications per prefix reduces further to 0.95 from 5.4 (this data is obtained from mapping ERR240726 to human genome v37, under the error threshold of 5), providing a 5.68x potential speedup over OSS without any optimizations.

4.3.3 The Full Algorithm

Algorithms 7 and 8 show the full algorithm of the Optimal Seed Solver. Before calculating the optimal x-seed frequency of the read, R, we assume that we already have the optimal 1-seed frequency of any substring of R and it can be retrieved in an $\mathcal{O}(1)$-time lookup via the $optimalFreq(\text{substring})$ function. This assumption is valid only if seeds are stored in a large hash table. For seeds that are pre-processed by the Burrows-Wheeler transformation, OSS requires $\mathcal{O}(s)$ total steps in FM-indexing to obtain the frequency of the seed, where s is the length of the seed. In total, it requires $\mathcal{O}(e \cdot L^2)$ total steps to index all possible seeds in the read, which potentially could generate $\mathcal{O}(e \cdot L^2)$ memory accesses and $\mathcal{O}(e \cdot L^2)$ cache misses in the worst case. Although the final number of cache misses is typically much smaller, due to the fact that the FM-index pointers of a substring is often in close proximities of the FM-index pointers of its enveloping superstring so that they often share the same cacheline (details of Burrows-Wheeler transformation and FM-indexing can be found in the literature [29]), they still significantly decrease the performance of OSS as each cache miss takes a long time to process.

In practice, it is unrealistic to maintain a hash table that stores all seeds of varying lengths. Therefore OSS has to use Burrows-Wheeler transformed suffix array which significantly slows down OSS due to generating large number of cache misses. Nonetheless, OSS is still significantly faster than finding optimal seeds by scanning through all possible seed permutations which allows us to compare different greedy seed selection mechanisms against optimal seeds at large scale.

Despite being much faster than exhaustive search, OSS is still very slow due to frequent cache misses. As a result, we do not intend to use OSS in a real mapper. Instead, OSS serves as a golden standard as the lowest total seed frequency achievable by a Pigeonhole-principle based seed selection method. By reducing the complexity of finding optimal seeds, OSS enables us to measure the slack in total seed frequency between existing Pigeonhole-principle based seed selection mechanisms and the optimal set of seeds on a large scale.

Let $firstOptDivider(\text{prefix})$ be the function to calculate the first optimal divider of a prefix. Then the optimal set of seeds can be calculated by filling a 2-D array, opt_data, of size $(x-1) \cdot L$. In this array, each element stores two data: an optimal seed frequency and a first optimal divider. The element at i^{th} row and j^{th} column stores the optimal i-seed solution of the prefix $R[1...j]$, which includes the optimal i-seed frequency of the prefix and the first optimal divider of the prefix, which divides the prefix into an $i-1$-seed prefix and an 1-seed substring.

Algorithm 7 provides the pseudo-code of $optimalSeedSolver$, which contains the core algorithm of OSS and the optimal divider cascading optimization; and Algorithm 8 provides the pseudo-code of $firstOptDivider$, which contains the early divider termination, the divider sprinting and the optimal solution forwarding optimizations.

To retrieve the starting and ending positions of each optimal seed, we can backtrack the 2-D array and backward induce the optimal dividers between optimal seeds. We start with the final optimal divider of the entire read, which divides the read into a $(x-1)$-seed prefix and a suffix. Among them, the suffix makes the last (right most) optimal seed of the read. Then we examine the $(x-1)$-seed prefix from the previous step and retrieve its optimal divider, which divides the prefix into an $(x-2)$-seed prefix and a substring. Among the two, the substring makes the second last optimal seed of the read. This process is repeated until we have retrieved all x optimal seeds of the read.

4.3.4 Example

This section presents an example of OSS in action. In this example, we are mapping a 100-bp read to the human reference genome under the error threshold of 3, as shown in Figure 4.15. Based on the pigeonhole principle, to tolerate 3 errors, we need a total of 4 seeds. According to the pseudo-code described in the Methods Section, there will be 3 iterations of finding partial

Algorithm 7: optimalSeedSolver

```
Input: the read, R
Output: the optimal $x$-seed frequency of R, opt_freq and the first $x$-seed optimal divider
  of R, opt_div
Global data structure: the 2-D data array opt_data[ ][ ]
Functions:
firstOptDivider: computes the first optimal divider of the prefix
optimalFreq: retrieves the optimal 1-seed frequency of a substring
Pseudocode:
// The first iteration:  1-seed solutions
for $l = L$ to $S_{min}$ do
    prefix = R[$1...l$];
    opt_data[$1$][$l$].freq = $optimalFreq$(prefix);
// (From 2 to x-1 seeds)
for $iter = 2$ to $x - 1$ do
    prev_div = $L - S_{min} + 1$;
    for $l = L$ to $iter \cdot S_{min}$ do
        prefix = R[$1...l$];
        // Find the optimal divider
        div = $firstOptDivider$(prefix, $iter$, prev_div);
        // Get frequencies of the 2 parts
        1st_part = R[$1...div - 1$];
        2nd_part = R[$div...L$];
        1st_freq = opt_data[$iter - 1$][$div - 1$].freq;
        2nd_freq = $optimalFreq$(2nd_part);
        opt_data[$iter$][$l$].$div$ = div;
        opt_data[$iter$][$l$].$freq$ = 1st_freq + 2nd_freq;
        // Optimal seed cascading,
        prev_div = div;
prev_div = $L - S_{min} + 1$;
// Find the optimal divider of the read
opt_div = $firstOptDivider$(R, $L - S_{min} + 1$);
1st_part = R[$1...opt\_div - 1$];
2nd_part = R[$opt\_div...L$];
1st_freq = opt_data[$x - 1$][$opt\_div - 1$].freq;
2nd_freq = $optimalFreq$(2nd_part);
opt_freq = 1st_freq + 2nd_freq;
return opt_freq, opt_div;
```

optimal solutions in all prefixes of the read (1 seed, 2 seeds and 3 seeds respectively), followed by a final optimal divider search of 4 seeds in the entire read.

Algorithm 8: firstOptDivider

```
Input: the prefix; the iteration count, $iter$; the previous prefix divider, prev_div
Output: the first optimal divider of the prefix, opt_div
Global data structure: the 2-D data array opt_data[ ][ ], the optimal 2nd-part frequency
  of the previous prefix, opt_2nd_freq
Functions:
$optimalFreq$: retrieves the optimal 1-seed frequency of a substring
Pseudocode:
// optimal solution forwarding
2nd_part = prefix[$prev\_div ... end$];
2nd_freq = $optimalFreq$(2nd_part);
if opt_2nd_freq = 2nd_freq then
    return $prev\_div$;
first_div = prev_div;
min_freq = MAX_INT;
prev_1st_freq = MAX_INT;
prev_2nd_freq = MAX_INT;
// Move divider backward until termination
for $div$ = prev_div to $(iter - 1) \cdot S_{min}$ do
    1st_part = prefix[$1 ... div - 1$];
    2nd_part = prefix[$div ... end$];
    1st_freq = opt_data[$iter$][$div - 1$].freq;
    // The 1st-part-freq of the next move
    next_1st_freq = opt_data[$iter$][$div - 2$].freq;
    // divider sprinting
    if next_1st_freq = 1st_freq then
        $continue$;
    2nd_freq = $optimalFreq$(2nd_part);
    // early divider termination,
    if (1st_freq − prev_1st_freq) > prev_2nd_freq then
        $break$;
    freq = 1st_freq + 2nd_freq;
    // update the optimal divider
    if (freq $\leqslant$ min_freq then
        min_freq = freq;
        first_div = div;
        opt_2nd_freq = 2nd_freq;
    prev_1st_freq = 1st_freq;
    prev_2nd_freq = 2nd_freq;
return first_div;
```

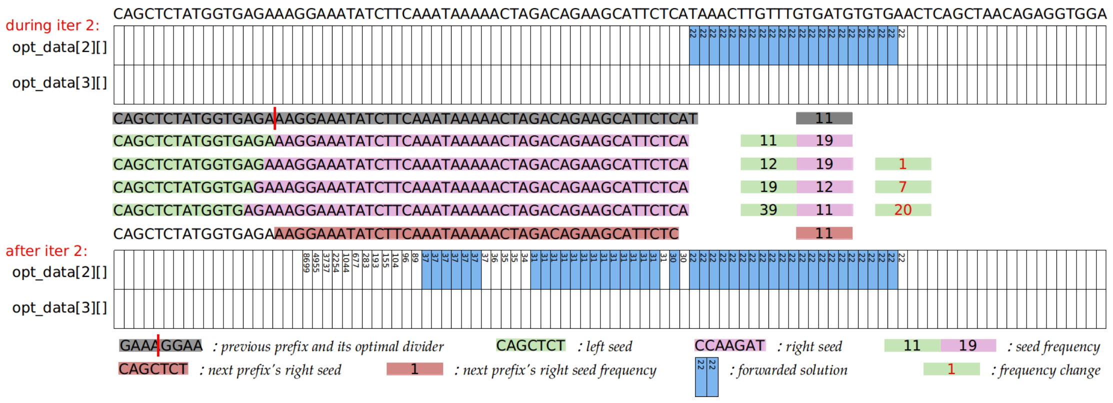

CAGCTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGTTTGTGATGTGTGAACTCAGCTAACAGAGGTGGA
during iter 2:
opt_data[2][]
opt_data[3][]
after iter 2:
: previous prefix and its optimal divider
: left seed
: right seed
: seed frequency
: next prefix's right seed
: next prefix's right seed frequency
: forwarded solution
: frequency change

Figure 4.15: An example of applying OSS to a 100-bp read in its second iteration.

In the first iteration, OSS searches for optimal 1-seed solution of all prefixes. Since the frequency of a seed monotonically decreases with longer seed lengths, for a prefix, the least frequent seed in it would be itself. In the second iteration, OSS searches for optimal 2-seed solutions for all prefixes. In this iteration, OSS starts with the longest prefix and gradually progresses to shorter prefixes [4].

Figure 4.15 shows how to derive the 2-seed optimal divider of a prefix. First, OSS inherits the optimal divider (marked in red) from the previous prefix (in gray), based on *optimal divider cascading*. Then, OSS divides the current prefix using the same divider and checks if its second part (in pink) has the same frequency as the second part of the previous prefix's division. In this example, the second part of the previous prefix has an optimal frequency of 11 while the second part of the current prefix has an optimal frequency of 19. Based on *optimal solution forwarding*, the two second parts from the two prefixes are not equal, therefore we cannot forward the optimal solution from the previous prefix. Next, OSS starts moving the divider towards the beginning of the prefix and queries the optimal 1-seed frequencies of the two parts (numbers with green and pink backgrounds, respectively) as well as the frequency differences of the first part (green background with numbers highlighted in red) between two moves. When the frequency increase of the first part is greater than the optimal frequency of the second part, according to *early divider termination*, OSS stops moving the divider and selects the divider with the minimum total seed frequency and goes to the next prefix. In this example, the least frequent division is the first division, with the total seed frequency of 30. Hence, $opt\_data[2][59]$ is filled with 30. For the next prefix, after inheriting the optimal divider from the current prefix, we observe that the optimal frequency of its second part (in brown) is equal to the optimal frequency of the second part of the current prefix. In this case, OSS forwards the optimal divider of the current prefix as the optimal divider of the next prefix; it inherits the optimal frequency and moves on the next prefix.

This process is repeated until all prefixes are processed in the second iteration. Figure 4.15 shows the $opt\_data[2][]$ array after the second iteration is finished. In this array, all prefixes that

[4]In our implementation, we do not start with the entire read but only start with the prefix that ends at $L-(x-i)\cdot S_{min}$, where x is the total number of required seeds and i is the current iteration. Any longer prefixes will not contribute to the final result. In this example, we have $S_{min}=10$.

inherit the optimal solution from the previous prefix are marked with a blue background.

The third iteration is similar to the second iteration, except that the first part of the division now provides two seeds. This information is provided by $opt\_data[2][]$. Figure 4.16 shows an example prefix in the third iteration.

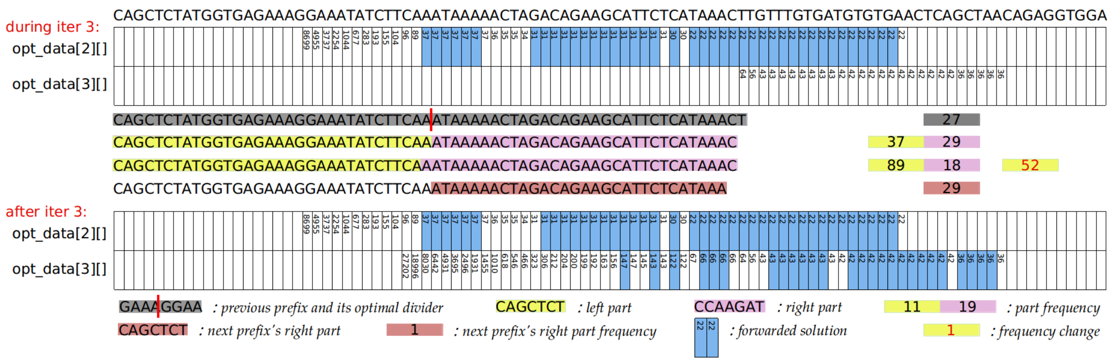


Figure 4.16: An example of applying OSS to a 100-bp read in its third iteration.

Finally, OSS searches for the optimal divider of the entire read: the divider that divides the read into a 3-seed prefix and a 1-seed suffix that produces the least total seed frequency. This process is the same as searching for optimal prefix dividers in previous iterations, which starts from the rightmost position and gradually moves towards the beginning of the read. This process is also obliged by early divider termination.

Figure 4.17 shows the process of the final optimal divider search. This figure also showcases *divider sprinting*. In Figure 4.17, we can see that from $opt\_data[2][90]$ to $opt\_data[2][86]$, the optimal frequency is always 36; from $opt\_data[2][85]$ to $opt\_data[2][74]$, the optimal frequency is always 42; from $opt\_data[2][73]$ to $opt\_data[2][66]$, the optimal divider is always 43. This suggests that OSS only needs to evaluate the dividers at the beginning and end of each interval (90, 86, 85, 74, 73, 66 and 65) while skipping computations within the intervals. For the read provided in the example, the least total seed frequency is 47.

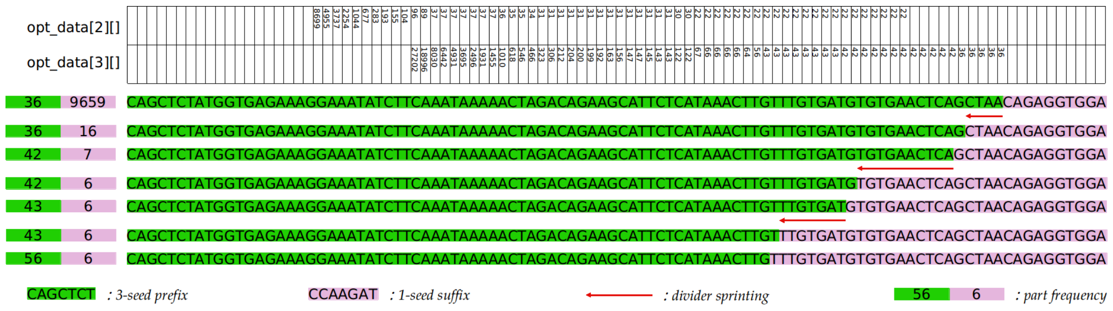


Figure 4.17: An example of divider sprinting.

4.3.5 Backtracking in Optimal Seed Solver

The pseudo-code of the backtracking process is provided in Algorithm 9. The key idea behind the backtracking algorithm is simple: the element of the i^{th} row and the j^{th} column of opt_data

stores the optimal divider, div, of the substring $R[1...j]$. This div suggests that by optimally selecting $i-1$ seeds from $R[1...div-1]$ and one seed from $R[div...j]$, we can obtain the least frequent i seeds from $R[1...j]$. From div we can learn that substring $R[div...j]$ provides the i^{th} optimal seeds. Similarly, by repeating this process for the element of $opt\_data[i-1][div-1]$, we can learn the position and length of the $(i\text{-}1)^{th}$ optimal seeds. We can repeat this process until we have learnt all optimal dividers of the read.

Algorithm 9: Backtracking

```
Input: the final optimal divider of the read, opt_div
Output: an array of optimal dividers of the read, div_array
Global data structure: the 2-D data array opt_data[ ][ ]
Pseudocode:
// Push in the last divider
div_array.push(opt_div);
prev_div = opt_div;
for $iter = x - 1$ to 2 do
    div = opt_data[$iter$][$prev\_div - 1$].$div$;
    div_array.push(opt_div);
    prev_div = div;
return div_array;
```

4.4 Results

The primary contribution of this work is a dynamic programming algorithm that derives the optimal non-overlapping seeds of a read in $\mathcal{O}(x \cdot L)$ operations on average. To our knowledge, this is the first work that finds the optimal seeds and the optimal frequency of a read.

The most related prior works are optimizations to the seed selection mechanism that reduce the sum of seed frequencies of a read using greedy algorithms.

We first quickly distinguish OSS from other methods [45, 47, 51] which solve similar yet unrelated problems. These previous works either determine the number and length of erroneous seeds such that the total number of branches in bwt-backtracking is minimized for each seed [45] or simply select seeds and their locations through probabilistic methods without providing error tolerance guarantees (e.g., bowtie2 [47] and BWA-MEM [51]). By contrast, OSS finds the number and lengths of non-overlapping seeds such that the total frequency of all seeds is minimized. These prior mechanisms are not designed for seed-and-extend based mappers that rely on non-overlapping seeds following the pigeonhole principle. In this work, we only compare seed selection mechanisms that follow the pigeonhole principle.

In the remainder of this section, we qualitatively and quantitatively compare the Optimal Seed Solver (OSS) to four state-of-the-art works first introduced in chapter 1.2. They are: *Cheap K-mer Selection (CKS)* in FastHASH [102], *Optimal Prefix Selection (OPS)* in Hobbes [3], *Adaptive Seed Filter (ASF)* in the GEM mapper [71] and *spaced seeds* in PatternHunter [67]. Among the four prior works, ASF represents works from the first category; CKS and OPS represent works from the second category and spaced seeds represents works from the third category.

Among the four implementations, OPS bears the most resemblance to OSS. The basis of OPS is also a dynamic programming algorithm that implements a simpler recurrence function. The major difference between OPS and OSS is that OPS does not need to derive the optimal length of each seed, as the seed length is fixed to k-bp. This reduces the search space of optimal fixed-length seeds to a single dimension, i.e., only seed placement. The worst case/average complexity of OPS is $\mathcal{O}(L \cdot x)$.

We compare the average case complexity, memory traffic and effectiveness of OSS against ASF, CKS, OPS and spaced seeds as well as the naïve mechanism, which selects fixed seeds consecutively. Memory traffic is measured by the number of required seed frequency lookups to map a single read. The effectiveness of a seed selection scheme is measured by the average seed frequency of mapping 4,031,354 101-bp reads from a read set, ERR240726 from the 1000 Genomes Project, under different numbers of seeds.

Table 4.1 summarizes the average-case complexity and memory traffic of each seed selection optimization. From the table, we can observe that OSS requires the most seed frequency lookups ($\mathcal{O}(L^2)$) with the worst average-case complexity, ($\mathcal{O}(x \cdot L)$), which is the same as that of OPS.

We do not measure the execution time of each mechanism because different seed selection optimizations are combined with different seed database implementations. CKS, OPS and spaced seeds use hash tables for short, fixed-length seeds while ASF and OSS employ slower but more memory-efficient BWT and FM-index for longer, variable-length seeds. However, this combination is inter-changeable. CKS and OPS can also work well with BWT and FM-index and ASF, OSS can also be combined with a large hash-table, given sufficient memory space. Besides, different existing implementations have their unique, implementation-specific seed database optimizations, which introduces more variations to the execution time. Due to these reasons, we only compare the complexity and memory traffic of each seed selection scheme, without measuring their runtime performance.

We benchmark each seed optimization scheme with multiple configurations. We benchmark ASF with multiple frequency thresholds, 5, 10, 100, 500 and 1000. If a read fails to provide enough seeds in ASF, due to having many long seeds under small thresholds, the read will be processed again in CKS with a fixed seed length of 12-bp. We benchmark CKS, OPS and the naïve seed selection mechanism under three fixed seed lengths, 12, 13 and 14. We benchmark spaced seeds with the default bit-mask provided in the PatternHunter paper [67], "110100110010101111", which hashes 18-bp long seeds into 11-bp long signatures.

All seed selection mechanisms are benchmarked using an in-house seed database, which supports varying seed lengths between $S_{min} = 10$ and $S_{max} = 30$.

	OSS	ASF	CKS	OPS	Spaced seeds	naïve
Complexity	$\mathcal{O}(x \cdot L)$	$\mathcal{O}(x)$	$\mathcal{O}(x \cdot log\frac{L}{k})$	$\mathcal{O}(x \cdot L)$	$\mathcal{O}(x)$	$\mathcal{O}(x)$
Number of lookups	$\mathcal{O}(x \cdot L)$	$\mathcal{O}(x)$	$\mathcal{O}(\frac{L}{k})$	$\mathcal{O}(L)$	$\mathcal{O}(x)$	$\mathcal{O}(x)$

Table 4.1: An average-case complexity and memory traffic comparison of OSS with other seeding mechanisms.

Nonetheless, OSS is the most effective seed selection scheme, as Figure 4.18 shows. Among all seed selection optimizations, OSS provides the largest frequency reduction of seeds on average, achieving a 3x larger frequency reduction compared to the second-best seed selection scheme, OPS.

As shown in Figure 4.18, the average seed frequencies of OSS, CKS and OPS increase with larger seed numbers. This is expected, as there is less flexibility in seed placement with more seeds in a read. For OSS, more seeds also means shorter average seed length, which also contributes to greater average seed frequencies. For ASF, average seed frequencies remain similar for three or fewer seeds. When there are more than three seeds, the average seed frequencies increase with more seeds. This is because up to three seeds, all reads have enough base-pairs to accommodate all seeds, since the maximum seed length is $S_{max} = 30$. However, once beyond three seeds, reads start to fail in ASF (due to having insufficient base-pairs to accommodate all seeds) and the failed reads are passed to CKS instead. Therefore, the seed frequency increase after three seeds is mainly due to the increase in CKS. For $t = 10$ with six seeds, we observe from our experiment that 66.4% of total reads fail in ASF and are processed in CKS instead.

For CKS and OPS, the average seed frequency decreases with increasing seed length when the number of seeds is small (e.g., < 4). When the number of seeds is large (e.g., 6), it is not obvious if greater seed lengths provide smaller average seed frequencies. In fact, for 6 seeds, the average seed frequency of OPS rises slightly when we increase the seed length from 13-bp to 14-bp. This is because, for small numbers of seeds, the read has plenty of space to arrange and accommodate the slightly longer seeds. Therefore, in this case, longer seeds reduce the average seed frequency. However, for large numbers of seeds, even a small increase in seed length will significantly decrease the flexibility in seed arrangement. In this case, the frequency reduction of longer seeds is surpassed by the frequency increase of reduced flexibility in seed arrangement. Moreover, the benefit of having longer seeds diminishes with greater seed lengths. Many seeds are already infrequent at 12-bp. Extending the infrequent seeds longer does not introduce much reduction in the total seed frequency. This result corroborates the urge of enabling flexibility in both individual seed length and seed placement.

Overall, OSS provides the least frequent seeds on average, achieving a 3x larger frequency reduction than the second best seed selection scheme, OPS.

As discussed earlier, OSS uses Burrows-Wheeler transformation as the underlying data structure for seeds, which generates a large number of cache misses in practice. We benchmarked the execution time of OSS that uses a Burrows-Wheeler Transformed suffix array from the SDSL library [30] as the underlying data structure for seeds against OPS that uses an in-house single-level hash table. We observed that on average OSS is 21.3x times slower than OPS.

4.5 Conclusion

In this work, we confirmed the *frequent seed phenomenon* discovered in previous works [43], which suggests that in a naïve seed selection scheme, mappers tend to select frequent seeds from reads, even when using long seeds. To solve this problem, we proposed the *Optimal Seed Solver* (OSS), a dynamic-programming algorithm that finds the optimal set of seeds that has the minimum total frequency. We introduced four optimizations to OSS: *optimal divider cascading*,

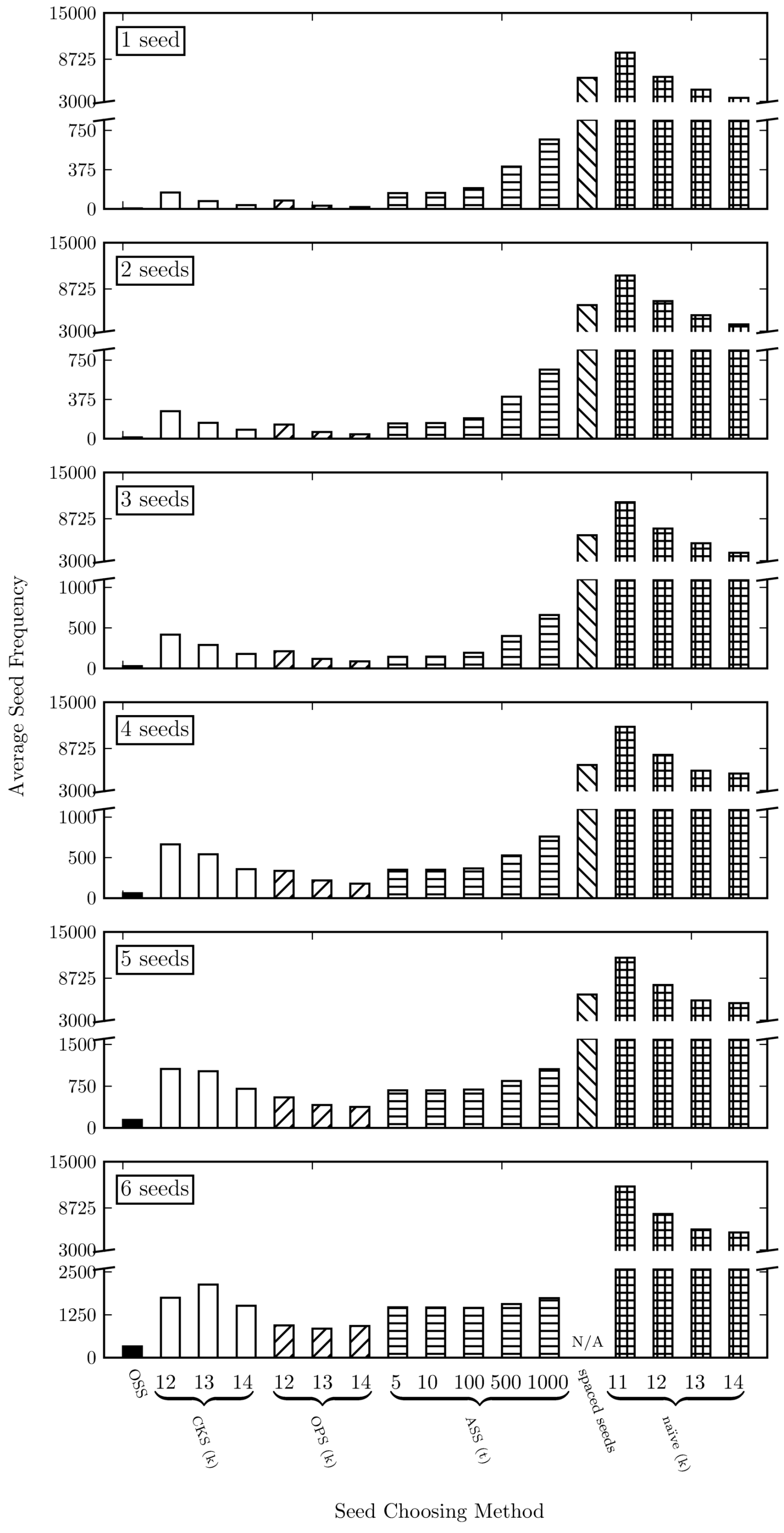


Figure 4.18: Average seed frequency comparison of Optimal Seed Solver (OSS) with other seed selection mechanisms.

early divider termination, *divider sprinting,* and *optimal solution forwarding*. Using all four optimizations, we reduced the empirical average-case complexity of OSS to $\mathcal{O}(x \cdot L)$, where x is the total number of seeds and L is the length of the read; and achieved a $\mathcal{O}(x \cdot L^2)$ worst-case complexity. We compared OSS to four prior studies, Adaptive Seed Filter, Cheap K-mer Selection, Optimal Prefix Selection and spaced seeds and showed that OSS provided a 3-fold seed frequency reduction over the best previous seed selection scheme, Optimal Prefix Selection. We conclude that OSS is an efficient algorithm to find the best set of seeds in large scale, although in practice OSS has limited performance due to generating large quantity of cache misses.

Chapter 5

Error Resilient Seeds: Fast and Comprehensive Read Mapping with Fewer Seeds

5.1 Background

As discussed in Chapter 1, using longer seeds as well as fewer seeds improves the speed of a mapper since the number of seed locations to verify is reduced. The major drawback of using longer, fewer seeds, however, is that it reduces the sensitivity of the mapper, since longer seeds consume more base pairs while there is only a limited number of base pairs in a read.

The key problem that prevents ubiquitously using long seeds (even with OSS, a mapper has to balance long seeds with short seeds), is that each seed can tolerate only a single error. This means a seed either has no error or has *one or more errors*. Hence, to guarantee tolerating e errors a mapper to draw $e + 1$ non-overlapping seeds, as in the worst case if all seeds are erroneous, there will be at least $e + 1$ errors in the read. To further increase the sensitivity, a mapper has to draw more non-overlapping seeds, reducing average seed length.

If each seed can sustain more than a single error, in the sense that a seed guarantees that there must be multiple errors in it when the seed is not error-free, then the total error tolerance of mapping a read can exceed the total number of non-overlapping seeds extracted from this read. For instance, if we have N seeds from a read, and for each seed we know that either it is error-free or it contains at least x number of errors ($x > 1$). Then if all seeds contain errors then there must be $N \cdot x > N$ errors in the read.

When a seed is erroneous, then the true mapping of the read must be excluded from the locations of the erroneous seed. For a seed s, we call *the minimum number of errors that a seed must have when the read is mapped to any true location beyond the seed locations of s, as* ***the error resilience*** *of s*.

Currently, naïve seeds used in modern mappers only have an error resilience of one. If a read is mapped to a location loc which does not belong to seed s, the only knowledge available is that s must not match 100% to the reference at loc. There is no telling how many errors must there be in s when the read is mapped to loc.

In this chapter, we present a new seeding concept, called *error resilient seeds* (ERS). Error resilient seeds attaches each seed with an extra *error resilience* attribute which indicates the least number of errors residing in the seed if the read is mapped to any place outside of its seed locations.

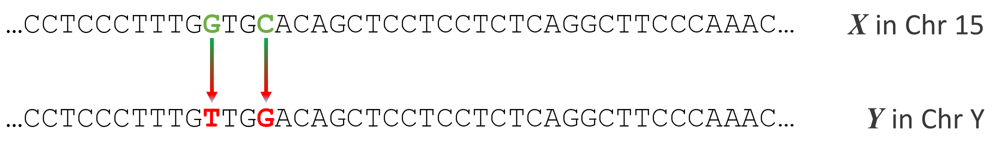

…CCTCCCTTTGGTGCACAGCTCCTCCTCTCAGGCTTCCCAAAC…
X in Chr 15
…CCTCCCTTTGTTGGACAGCTCCTCCTCTCAGGCTTCCCAAAC…
Y in Chr Y

Figure 5.1: An example of using error resilient seeds in read mapping.

With error resilience seeds, mappers can easily achieve high sensitivity with just a few long seeds. The total error tolerance of mapping a read equals to the sum of error resilience of all seeds. If there exists a mapping at loc that is not included by any seed, based on the definition of error resilience of seeds, there must be at least

$$\sum_{s_i \in seeds} E(s_i)$$

errors in the read at loc, where $E(s_i)$ denotes the error resilience of seed s_i. Figure 5.1 shows an example of achieving higher error tolerance through error resilient seeds, in lieu of naïve seeds.

This is the first effort to introduce the concept of error resilient seeds, which removes the tight decoupling of error tolerance guarantees and number of non-overlapping seeds. The key method of generating error resilient seeds is to pre-compute an error resilience database, which stores the error resilience information for all seeds. At runtime, a mapper simply extracts long seeds from a read in the form of MEMs (maximum exact matches) and queries the error resilience database. The total error tolerance of mapping a read simply equals to the sum of error resilience of all seeds.

Since seed error resilience is defined as the minimum number of errors in the seed when the read is optimally aligned to the reference at any location other than its own seed locations, thus the error resilience of a seed is simply the edit distance to its most similar sequence in the reference. Any other sequence in the reference is guaranteed to carry an equal amount or even more errors than the most similar sequence to the seed. The edit distance between the seed and its most similar sequence hence marks the error resilience of the seed.

However, due to the nature of strings, any sequence is one edit away from its left or right immediate substrings, which are substrings that have the left most or right most letter removed. Similarly, any sequence is one edit away from its left or right immediate superstrings and two edits away from any immediate overlapping strings (immediate overlapping string S' of seed S is a string whose prefix is a suffix of S or vice versa). Given a string S, Figure 5.2 shows examples of immediate substrings, immediate superstrings and immediate overlapping strings of S.

While immediate substrings and superstrings do contain edits, they do not necessarily meet the criteria to produce a meaningful error resilience of the seed. Often the seed locations of S', a substring (or superstring or overlapping string) of seed S, overlaps completely with seed locations of S. In such cases, the distance $d(S, S')$ is not the error resilience of S. There does not exist a seed location $loc_{S'}$ of S' that does not overlap with any seed location in $\{loc_S\}$ of S.

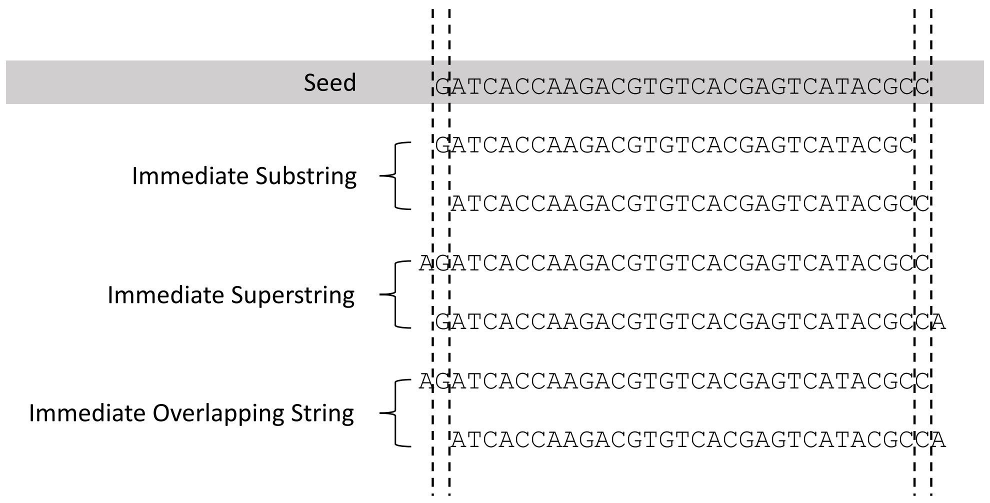

Seed
GATCACCAAGACGTGTCACGAGTCATACGCC
Immediate Substring
GATCACCAAGACGTGTCACGAGTCATACGC
ATCACCAAGACGTGTCACGAGTCATACGCC
Immediate Superstring
AGATCACCAAGACGTGTCACGAGTCATACGCC
GATCACCAAGACGTGTCACGAGTCATACGCCA
Immediate Overlapping String
AGATCACCAAGACGTGTCACGAGTCATACGC
ATCACCAAGACGTGTCACGAGTCATACGCCA

Figure 5.2: Examples of immediate substrings, immediate superstrings and immediate overlapping strings.

Given a seed S, we call sequences without locations exclusive to itself and not overlapping with any locations of S as *trivial neighbors* of S. **Trivial neighbors do not contribute to computing the error resilience of a seed.** We call sequences other than trivial neighbors as *non-trivial neighbors*. We provide a formal definition of trivial neighbors in Section 5.3.2.

The computational problem that we propose to solve, therefore, is the following:

Problem: For all sequences S in the reference, find the edit distance between S and its most similar non-trivial neighbor.

5.2 Contribution

ERS provides the following contributions:

- We introduce a novel seeding methodology, error resilient seeds, which enables using long seeds while maintaining high error tolerance in read mapping.
- We provide an algorithm to compute the minimum edit-distances to a seed's most similar non-trivial neighbor in the reference.
- We propose a greedy seed selection mechanism that provides error tolerance guarantees using ERS.
- We compare the greedy seed selection mechanism using ERS to OSS and show that it is 1.17x more efficient than the best existing seeding algorithm.

5.3 Methods

Figure 5.3 showcases a non-trivial neighbor, ***Y***, of seed ***X*** with minimum edit distance. ***Y*** has a total of two edits compared to ***X***, as shown in the figure. This suggests that besides reference

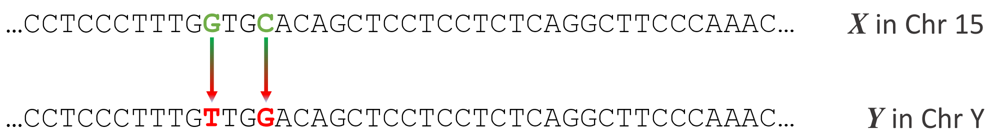

…CCTCCCTTTGGTGCACAGCTCCTCCTCTCAGGCTTCCCAAAC…
X in Chr 15
…CCTCCCTTTGTTGGACAGCTCCTCCTCTCAGGCTTCCCAAAC…
Y in Chr Y

Figure 5.3: An example of a non-trivial neighbor to a seed.

sequences in close proximities of seed locations of ***X***, any other sequence in the reference must have an edit distance of at least two, when compared to ***X***.

Therefore, to construct the error resilience database for a reference genome, one can search for the minimum edit distance among all non-trivial neighbors for all seeds. While finding the most similar non-trivial neighbor provides a tight error resilience lower bound; in practice, overly excessive error resilience is not always necessary. Since mappers are only required to find all mapping within a small error threshold (usually within 5% of the read length), a lower bound of seed error resilience up to the mapper's error threshold is sufficient to meet the error tolerance requirement. In fact, any lower bound of error resilience above one provides a **net gain** to read mapping. An error resilience of one only reduces an error resilient seed to an naïve seed which are widely used in current mappers. In other words, error resilient seeds never negatively impact read mapping.

Hence, given any seed in the reference, our goal is to 1) determine whether there exists a non-trivial neighbor $N_{non\_trivial}$ of seed S whose edit distance to S is below the read mapping error threshold D; and 2) if there exist such neighbors, find the minimum edit distance $d_{min} < D$ between S and its non-trivial neighbors.

With above principle in mind, we construct the error resilience database in a two-step fashion: First, we construct a neighbor database, which stores all neighbors of all seeds within the edit distance threshold D. Then we find the non-trivial neighbors of each seed and obtain the minimum edit distance to each seed's non-trivial neighbors, if there exist any.

5.3.1 Construction of Neighbor Database

We find all neighbors of a sequence in a reference by first building a suffix trie of the reference and then traversing the suffix trie in a breadth-first fashion. Specifically, given P, which is the maximum length of error resilient seeds to profile, we travel the suffix trie up to depth of P. Each time when we visit a node v, we find all neighbors of v whose edit distance from v is below D. If there exist a total of M pairs of sequences in the reference with up to length $P + D$, where the edit distance between the two sequences is smaller than D, then our algorithm can find all such neighbor pairs in time linear to M.

Let $T = (V, E)$ be a suffix trie of depth P on alphabet Σ. Let $r \in V$ be the root of T. For any vertex $v \in V$, we denote by $s(v)$ be the string concatenating the letters on all edges following the unique path from r to v. For simplicity, we denote the edit distance between the sequences $s(u)$ and $s(v)$ of two nodes u and v as $d(u, v)$. We aim to solve the following problem.

Problem 1 (Neighbors in Suffix Trie). *Given a suffix trie $T = (V, E)$ and an integer D, to compute all pairs of vertices $u, v \in V$ such that $d(u, v) \leqslant D$.*

For any $v \in V$, we denote by $p(u)$ be the parent vertex of v. We denote by $\sigma(u, v)$ the letter on edge $(u, v) \in E$. We first prove the following lemmas.

Lemma 4. *Let $u, v \in V$. We have $d(u, v) \leqslant D$ only if $d(p(u), p(v)) \leqslant D$.*

Proof. Provided in Landau & Vishkin's paper [46]. □

Lemma 4 states that we only need to examine these pairs of vertices whose parents is within a distance of at most D. This gives a way to avoid all-against-all pairwise comparison.

Lemma 5. *Let $u, v \in V$. We have*

$$d(u, v) = \min \begin{cases} d(p(u), p(v)) + \delta \\ d(p(u), v) + 1 \\ d(u, p(v)) + 1 \end{cases}$$

where $\delta = 1$ if $\sigma(p(u), u) \neq \sigma(p(v), v)$ and $\delta = 0$ if $\sigma(p(u), u) = \sigma(p(v), v)$.

Proof. First proved in Smith-Waterman paper [92]. □

Lemma 5 states that the distance between two vertices can be computed from the distances related with their parents. This gives us a way to avoid computing the edit distance from scratch.

Based on the above the lemmas, we design an algorithm for Problem 1 that has a time complexity that is linear to the number of unique sequence pairs whose edit distance is smaller than D. We first assign each vertex in V an integral *rank* from $\{1, 2, \cdots, |V|\}$ following a top-down, left-to-right order. Specifically, the root r of T has rank of 1, and then the second level of T (i.e., children of r) have ranks of $2, 3, \cdots$, from the leftmost child to the rightmost child, and so on. In this ordering, higher level nodes always rank higher than lower level nodes. Among nodes of the same level, children of a higher ranking node rank higher than children of a lower ranking node. A breadth-first-search traversing T will assign such rank for all vertices.

For any $v \in V$, we define $X_v := \{u \in V \mid d(u, v) \leqslant D\}$ as the set of vertices that are within a distance of at most D to v, including v itself, and define $Y_v := \{d(u, v) \mid u \in X_v\}$ as the accompanying set that stores the corresponding distance for each vertex in X_v. Our algorithm will compute and maintain X_v and Y_v for each vertex $v \in V$ from low ranks to high ranks. We use two arrays to store X_v and Y_v for each vertex $v \in V$, respectively. Without loss of generality, we also use X_v and Y_v to represent the corresponding arrays in our algorithm.

Our algorithm for Problem 1 is given in Algorithm **??**. The overall framework of this algorithm is for each vertex $v \in V$ from low rank to high rank to compute $X_{v'}$ and $Y_{v'}$ for each child v' of v (from line 4 to line 17), as shown in Figure 5.4. This algorithm keeps the following three invariants:

1. For any vertex $v \in V$, array X_v are always sorted according to their ranks, i.e., vertices that are added to X_v are always in ascending order *w.r.t.* their ranks.
2. Right before processing vertex v (i.e., before line 4 of Algorithm **??**), we have that X_v and Y_v are already completely computed and sorted *w.r.t.* their ranks.
3. Right after processing vertex v (i.e., after line 17 of Algorithm **??**), we have that $X_{v'}$ and $Y_{v'}$ are completely computed and sorted *w.r.t.* their ranks for each child v' of v.

Algorithm ??: Linear Time Algorithm for Problem 1

```
Input: Suffix trie $T = (V, E)$ and integer $D$
Output: $X_v$ and $Y_v$ for each $v \in V$
 1. Assign ranks for all vertices using breadth-first search on $T$
 2. Initialze $X_r$ and $Y_r$ for root $r \in V$.
 3. FOR each vertex $v \in V$ in ascending order:
 4.     Initialize pointer $k_v = 0$ for arrays $X_v$ and $Y_v$.
 5.     Initialize arrays $X_{v'}$ and $Y_{v'}$ for each child $v'$ of $v$ as empty arrays.
 6.     Initialize pointer $k_{v'} = -1$ for arrays $X_{v'}$ and $Y_{v'}$ for each child $v'$ of $v$.
 7.     FOR $k = 0 \to |X_v|$:
 8.         LET $u := X_v[k]$.
 9.         FOR each child $u'$ of $u$:
10.             FOR each child $v'$ of $v$:
11.                 LET $\delta = 1$ if $\sigma(u, u') \neq \sigma(v, v')$ and $\delta = 0$ if $\sigma(u, u') = \sigma(v, v')$. Compute
                    $d_1 = Y_v[k] + \delta$, i.e., $d_1 = d(v, u) + \delta$.
12.                 Increase $k_v$ until $X_v[k_v] \geqslant u'$. IF we have $X_v[k_v] = u'$, i.e., $u' \in X_v$,
                    THEN compute $d_2 = Y_v[k_v] + 1$, i.e., $d_2 = d(v, u') + 1$; otherwise set
                    $d_2 = \infty$.
13.                 Increase $k_{v'}$ until $X_{v'}[k_{v'}] \geqslant u$. IF we have $X_{v'}[k_{v'}] = u$, i.e., $u \in X_{v'}$,
                    THEN compute $d_3 = Y_{v'}[k_{v'}] + 1$, i.e., $d_3 = d(v', u) + 1$; otherwise set
                    $d_3 = \infty$.
14.                 Compute $d(v', u') = \min\{d_1, d_2, d_3\}$. IF $d(v', u') \leqslant D$, THEN add $u'$ to the
                    end of $X_{v'}$ and add $d(u', v')$ to the end of $Y_{v'}$.
15.             END FOR
16.         END FOR
17.     END FOR
18. END FOR
```

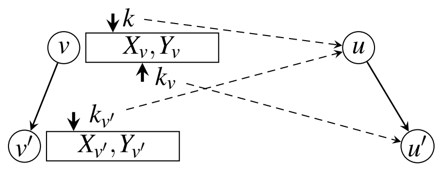


Figure 5.4: Illustration of processing a single vertex v (i.e., line 4 to line 17 of Algorithm **??**).

In the following we give necessary explanation about Algorithm **??** and verify that the above three invariants hold through the whole algorithm. The initialization step of our algorithm (line 2) is to compute X_r and Y_r for root r. This is to add all vertices in the first D levels of T to X_r from low to high ranks, and the distance for each such vertex is simply the level number subtract 1 (we assume that r locates at level 1). Notice that root r is also in X_r with $d(r, r) = 0$. Clearly, the first and the second invariant hold for root r.

For the current vertex $v \in V$, lines 4–17 of Algorithm **??** compute $X_{v'}$ and $Y_{v'}$ for any child v' of v. Line 4 and Line 6 initialize the pointers that will be used to fetch the existing edit distances $d(v, u')$ and $d(v', u)$ stored in Y_v and $Y_{v'}$, which are necessary to compute $d(v', u')$ according to Lemma 5. Specifically, k_v tracks the position of u' in array X_v (i.e., the index of u' in array X_v), and $k_{v'}$ tracks the position of u in array $X_{v'}$, for each child v' of v. Line 11 computes $d_1 := d(v, u) + \delta$, in which $d(v, u)$ is fetched from Y_v indexed by k. Line 12 computes $d_2 := d(v, u') + 1$, in which $d(v, u')$ is fetched from Y_v indexed by k_v. Line 13 computes $d_3 := d(v', u) + 1$, in which $d(v', u)$ is fetched from $Y_{v'}$ indexed by $k_{v'}$. Line 14 computes $d(v', u') := \min\{d_1, d_2, d_3\}$, and add u' to $X_{v'}$ and add $d(v', u')$ to $Y_{v'}$ if we have $d(v', u') \leqslant D$.

We now show that Algorithm **??** keeps the first invariant, i.e., for each child v' of v, the vertices that are added to $X_{v'}$ (and the corresponding distances that are added to $Y_{v'}$) are sorted according to their ranks. This is because, according to the second invariant, vertices in X_v (i.e., u examined in Algorithm **??**) are sorted *w.r.t.* their ranks. Also, if the rank of vertex u_1 is larger than that of vertex u_2, then we have that the rank of any child of u_1 is larger than that of u_2. These imply that u' examined in Algorithm **??** will be in ascending order *w.r.t.* their ranks. Following the fact that Algorithm **??** keeps the first invariant, we know that it suffices to increase k_v and $k_{v'}$ in order to locate u' and u in X_v and $X_{v'}$, respectively.

We now show that Algorithm **??** also keeps the third invariant. According to Lemma 4, for each child v' of v, each vertex $u' \in X_{v'}$ must satisfy that u' is a child of u such that $u \in X_v$. Algorithm **??** examines all such possibility of u'. Therefore, $X_{v'}$ and $Y_{v'}$ will be completely computed and sorted for each child v' of v after line 17. Following the fact that Algorithm **??** keeps the third invariant, we also have that Algorithm **??** keeps the second invariant, as we process vertex v in ascending order (line 3).

Let $M := \{(u, v) \mid d(u, v) \leqslant D\}$. Combining the above facts, we summarize the following theorem.

Theorem 3. *Algorithm* **??** *correctly computes* X_v *and* Y_v *for each* $v \in V$ *in* $O(|\Sigma|^2 \cdot M)$ *time.*

Proof. The correctness of Algorithm **??** directly follows from the fact that Algorithm **??** keeps the third invariant. We now analyze the running time of Algorithm **??**. We first show that for each $v \in V$, lines 4-17 computes $X_{v'}$ and $Y_{v'}$ for each child v' of v in $O(|X_v| \cdot |\Sigma|^2 + \sum_{v':p(v')=v} |X_{v'}|)$ time. According to the above fact that pointers of k_v and $k_{v'}$ can only move forward, we have that the operations of increasing these pointers within lines 12-13 cost $|X_v| + \sum_{v':p(v')=v} |X_{v'}|$ in total. Other than increasing pointers, operations in lines 11–14 cost constant time. Hence, lines 7–17 cost $|X_v| \cdot |\Sigma|^2$, as the number of children of each vertex is bounded by $|\Sigma|$. The overall running time of Algorithm **??** is thus bounded by $\sum_{v \in V} O(|X_v| \cdot |\Sigma|^2 + \sum_{v':p(v')=v} |X_{v'}|) = O(|\Sigma|^2 \cdot M)$. $\square$

5.3.2 Computing Error Resilience From the Neighbor Database

A neighbor u of seed v is a trivial neighbor if everywhere $s(u)$ appears in the reference, $s(v)$ also appears in close proximity. However, unless $u = v$, otherwise there will be small gaps between each pair of occurrences of $s(u)$ and $s(v)$ in the references. We formally define triviality of neighbors as follows: Suppose u and v are nodes in the suffix tree and loc_u is a seed location of $s(u)$ while loc_v is a seed location of $s(v)$. Further assume that set $\{loc_u\}$ contains all seed locations of $s(u)$ and $\{loc_v\}$ contains all seed locations of $s(v)$. We say $loc_u \ntriangleright \{loc_v\}$, if and only if $\nexists loc_v \in \{loc_v\}$ such that $|loc_u - loc_v| \leqslant d(u, v)$. Otherwise, $loc_u \rhd \{loc_v\}$. A sequence $s(u)$ is a non-trivial of $s(v)$, if and only if $\exists loc_u \in \{loc_u\}$ such that $loc_u \ntriangleright \{loc_v\}$.

After obtaining all neighbors of all nodes, We aim to solve the following problem.

Problem 2 (Finding Minimum Distance to Non-Trivial Neighbors)**.** *Given all neighbors X_v of node v, find $d_{min}(u, v)$ among vertices $u \in X_v$ where $s(u)$ is a non-trivial neighbor of $s(v)$.*

To solve problem 2, we first prove the following lemmas.

Lemma 6. *If $s(u)$ is a superstring of $s(v)$, then $s(u)$ is a trivial neighbor of $s(v)$.*

Proof. Lemma 6 is trivial. Since $s(v)$ is a substring of $s(u)$, we have $\forall loc_u \in \{loc_u\}$, $\exists loc_v$ within $[loc_u - d(u, v), loc_u + d(u, v)]$. □

Lemma 7. *If $s(u)$ is a substring of $s(v)$ and $s(u)$ is a non-trivial neighbor of $s(v)$, then $\exists w$, such that $s(w)$ does not overlap $s(v)$, $s(w) \neq s(v)$, $|s(w)| = |s(v)|$ while $d(w, v) \leqslant d(u, v)$.*

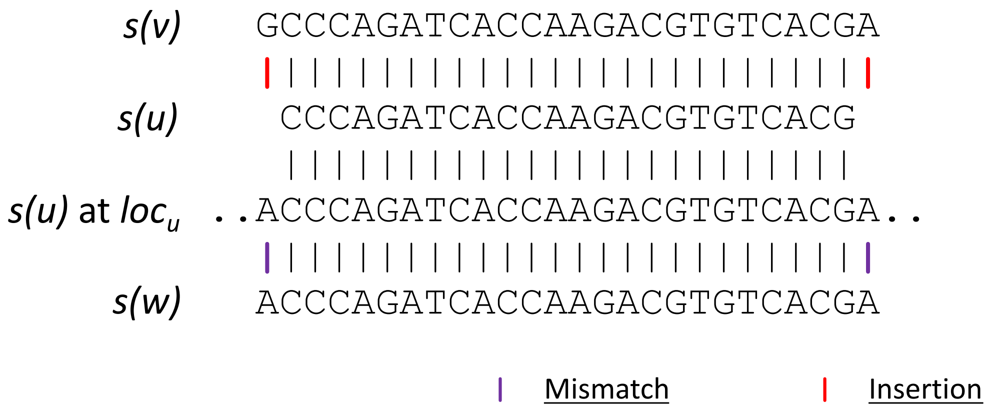

s(v)
GCCCAGATCACCAAGACGTGTCACGA
s(u)
CCCAGATCACCAAGACGTGTCACG
s(u) at loc_u
..ACCCAGATCACCAAGACGTGTCACGA..
s(w)
ACCCAGATCACCAAGACGTGTCACGA
Mismatch
Insertion

Figure 5.5: Illustration of Lemma 7.

Proof. Since $s(u)$ is a non-trivial neighbor of $s(v)$, $\exists loc_u \in \{loc_u\}$ but $loc_u \notin \{loc_v\}$. Because $s(u)$ is a substring of $s(v)$, $d(u, v) = |s(v)| - |s(u)|$.

Align $s(v)$ against $s(u)$ at loc_u and extract the reference sequence $s(w)$ that flanks $s(u)$ at loc_u with $|s(w)| = |s(v)|$ while keeping the alignment between $s(u)$ and $s(v)$, so that $s(w)$ is not an overlapping string of $s(v)$, as shown in Figure 5.5. Since $loc_u \ntriangleright \{loc_v\}$, $s(w) \neq s(v)$, as well as $s(v)$ and $s(w)$ share the common string $s(u)$, we have $d(w, v) \leqslant d(u, v)$.

□

Lemma 8. *If $s(u)$ is an overlapping string of $s(v)$ and $s(u)$ is a non-trivial neighbor of $s(v)$, then $\exists w$ such that $d(w, v) < d(u, v)$.*

Proof. Assume $s(w) = MEM_{u,v}$ marks the max-length common suffix-prefix string shared between $s(u)$ and $s(v)$, shown in Figure 5.6. Then $s(w)$ must be a substring of $s(v)$ while $d(w, v) < d(u, v)$ (one only needs a few deletions to convert $s(v)$ into $s(w)$, while it requires additional insertions to further convert $s(w)$ into $s(u)$).

Because $s(u)$ is a non-trivial neighbor of $s(v)$, $\exists loc_u \in \{loc_u\}$ while $loc_u \ntriangleright \{loc_v\}$. Since $s(w)$ is a substring of $s(u)$, therefore $s(w)$ must also be a non-trivial neighbor of $s(v)$.

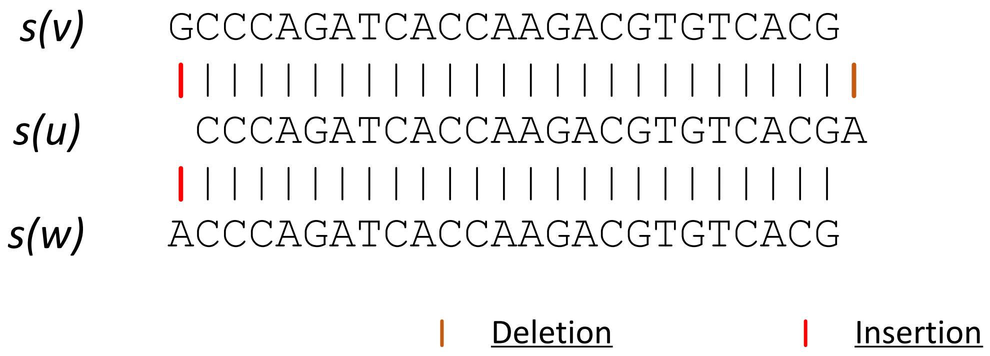

s(v) GCCCAGATCACCAAGACGTGTCACG
s(u) CCCAGATCACCAAGACGTGTCACGA
s(w) ACCCAGATCACCAAGACGTGTCACG
Deletion
Insertion

Figure 5.6: Illustration of Lemma 8.

Given that $s(w)$ is a non-trivial neighbor of $s(v)$, according to Lemma 7, there must exist a non-trivial neighbor $s(t)$ of $s(v)$ such that $d(t, v) \leqslant d(w, v) < d(u, v)$.

□

Lemma 6 7 and 8 state that there must exist node w such that $s(w)$ is not a superstring, substring or overlapping string of $s(v)$, while $d(u, v)$ is the minimum edit distance of v to its non-trivial neighbors. Hence, the error resilience of a seed equals to the minimum edit-distance to its neighbors that exclude the seed's own substrings, superstrings and overlapping strings.

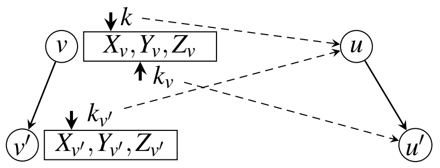


Figure 5.7: Illustration of adding $Z_v := \{Flag(u, v) \mid u \in X_v\}$ to each node.

To keep track of substring, superstring and overlapping string relationships among seeds in the reference, we associate each node in the suffix trie a new vector $Z_v := \{Flag(u, v) \mid u \in X_v\}$, where $Flag(u, v)$ stores whether $s(u)$ is a substring, a superstring or an overlapping string of $s(v)$. The updated workflow is illustrated in Figure 5.7.

The generation of $Flag(u, v)$ can be piggybacked on top of the computation of $d(u, v)$. There are a total of nine flag categories:

1. $s(u)$ is a prefix of $s(v)$.
2. $s(u)$ is a suffix of $s(v)$.
3. $s(v)$ is a prefix of $s(u)$.
4. $s(v)$ is a suffix of $s(u)$.
5. A prefix of $s(u)$ is a suffix of $s(v)$.
6. A suffix of $s(u)$ is a prefix of $s(v)$.
7. $s(u)$ is neither a prefix nor a suffix but a substring of $s(v)$.
8. $s(v)$ is neither a prefix nor a suffix but a substring of $s(u)$.

9. The rest.

In above definitions, we require the prefix or suffix of a string to be strictly a substring. In other words, suffix A of string B suggests that $A \neq B$.

A seed $s(u)$ may be a neighbor of multiple categories to another seed $s(v)$. For simplicity, we create the following rules: we initialize $s(v)$ with tags of categories 1, 2, 3 and 4 to itself for any node v in T (This is only a temporary compromize for the convenience of explanation. Although according to our definition, a node is not a prefix nor suffix to itself, we still tag each node to itself so that we can detect self-neighbors, which are obviously trivial neighbors. However, it does not mean a node is a prefix or suffix to itself). The empty string (the root node r of the suffix trie) is a prefix to any string. Any string is a suffix to the empty string. Finally, the empty string is not an overlapping string or a substring of any string.

$Flag(u, v)$ of node u to v can be computed in constant time if the flags of each node to their parent nodes are known. For example, in Figure 5.7, $s(u')$ is a prefix of $s(v')$, only if a) $u' = v'$ or b) u' is a prefix of v (u' belongs to category 1 of v). Similarly, $s(u')$ is a suffix of $s(v')$, only if a) $u' = v'$ or b) $s(u)$ is a suffix of $s(v)$ (u belongs to category 2 of v) and $\sigma(p(u), u) = \sigma(p(v), v)$. Category 3 and 4 follow suit, with u and v swapped role. Category 5 holds true for $s(u')$ to $s(v')$ only if a) the same category also holds for $s(u')$ to $s(v)$ or b) $s(u)$ is a suffix of $s(v')$ while $u \neq v'$ and $u \neq r$. The same description can also be applied to category 6 with u and v swapped role. The relationship of $s(u')$ to $s(v')$ belongs to Category 7 only if a) the same category also marks $s(u')$ to $s(v)$ or b) $s(u')$ is a suffix of $s(v)$ while $u' \neq v$ and $u \neq r$. Similarly, category 8 can be described just as category 7, just with the role of u and v swapped. Finally if $s(u')$ to $s(v')$ does not fit in to any category between 1 to 8, then it is tagged as a neighbor of category 9.

As shown above, $Flag(u', v')$ can be easily computed in constant time as long as flags between nodes and their parents are known. This invariant, requiring prior results of nodes with regard to each other's parents, is the same in computing $d(u', v')$. As a result, by piggybacking the computation $Flags(u', v')$ along computing $d(u', v')$, we do not increase the complexity of the algorithm.

Finally, the error resilience of seed $s(v)$ of node v is the minimum edit distance to its neighbors in X_v who belong to category 9. The result can be computed through a simple linear synchronized scan across X_v, Y_v and Z_v. The total time complexity of constructing the entire error resilience database hence is $O(|\Sigma|^2 \cdot M)$ time.

5.3.3 Applying Error Resilient Seeds to Read Mapping

We propose a greedy seed selection method for read mapping, which simply consecutively extracts MEMs from a read as seeds. After the termination of each MEM, we heuristically skip the next two base pairs in an effort to dodge potential errors (a MEM terminates prematurely only when it has bumped into an error). After drawing each seed, we obtain its error resilience from the error resilience database. After extracting all seeds, we sort them based on their frequency and gradually pick seeds starting from the least frequent seed. With each selected seed, we sum its error resilience to the overall error tolerance of all seeds. We stop extracting seeds once the selected seeds have enough error tolerance to meet the requirement.

Figure 5.8 shows an example of extracting error resilient seeds from a read. In this example,

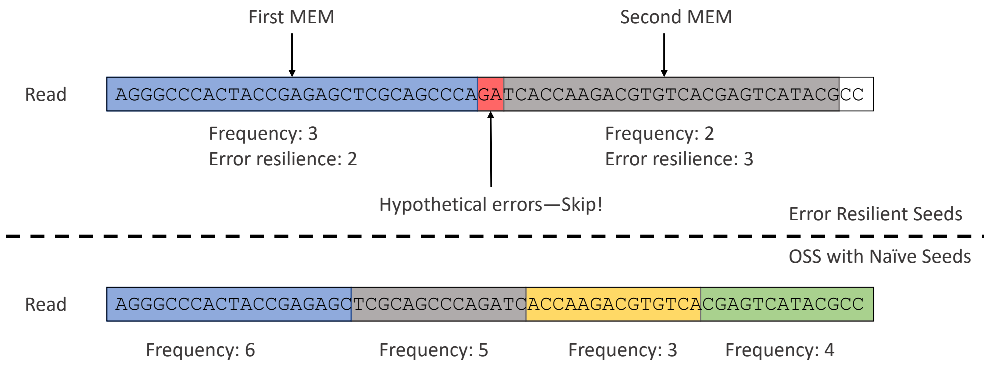


Figure 5.8: An example of drawing error resilient seeds from a read.

we require a total error tolerance of 3. As shown in the figure, the first seed has an error resilience of 2 and the second seed has an error resilience of 3. Notice that after the first seed, we skip the next two base pairs as there is a high probability that they might contain errors. Among the two seeds, the first seed has a lesser frequency. Therefore it is selected first. Since the first seed only provides an error resilience of 2, which is below our error resilience requirement of 3, we proceed to select the second seed. Combined, they provide a total error tolerance of 5, which meets our error tolerance requirement, hence seed selection terminates.

Compared to OSS, which selects naïve seeds, also shown in Figure 5.8, the new greedy seed selection algorithm selects fewer but longer seeds while achieving an even higher error tolerance guarantee. In the case of OSS, to satisfy the requirement of tolerating 3 errors, at least 4 seeds are selected. As a result, compared to selecting error resilient seeds, OSS with naïve seeds lead to a) selecting more seeds and b) selecting shorter and more frequency seeds.

When seeds do not produce enough error resilience to satisfy the error tolerance requirement, we revert to using the optimal seed solver with canonical naïve seeds. While the above naïve seeding mechanism might not fully utilize the potential of error resilient seeds, we observe that in practice, it rarely fails to meet the error tolerance requirement.

The error reslience database can be maintained in a two-dimensional table of $O(N+P)$ rows, where $N+P$ is the number of unique sequences at level P. Each row represents a node in the suffix trie at depth P. In the entry of node v at depth P, it stores the error resilience of a selection of Q seeds, $s(u)$, where u are nodes in the suffix trie at Q selective depths along the path to node v. For each selected u, the error resilience entry stores the error resilience of $s(u)$ in $log_2 D$ bits. Overall, with a number of Q selected depths, the entire error resilience database can be stored in $O(N \cdot Q \cdot log_2 D)$ bits.

5.4 Results

We compare our greedy seed selection method using error resilient seeds against our previously proposed, state-of-the-art seed selection mechanism, OSS using naïve seeds. We tested both seed selection mechanisms on a 22-million, 100-bp E. coli read set from EMBL-EBI, ERX008638-1. To use error resilient seeds, we built an error resilience database for the *E. coli* reference genome. We scanned through the entire reference genome and gathered all seeds of lengths up to 60 base pairs (we stop at length 60 because most seeds at length 60 have maximum error resilience of 4). For each seed, we searched for the minimum distance to its closes non-trivial neighbor within an edit distance of 4. If a seed has no neighbors in 4 edits, we designate the error resilience of such seed as 4, which is a lower bound of its true error resilience.

We measure the effectiveness of both seed selection mechanisms by counting the average total frequency of selected seeds under error tolerance requirements from 0 to 4.

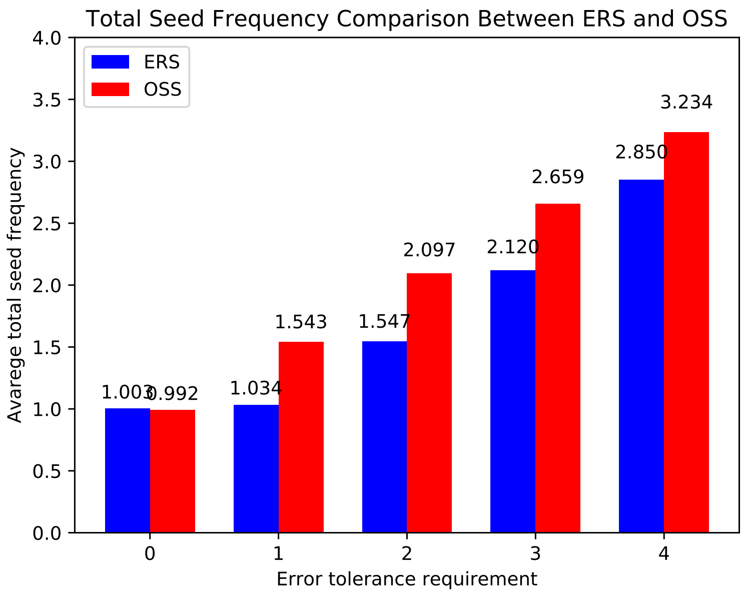


Figure 5.9: Comparison of the total seed frequency per read between ERS and OSS.

Figure 5.9 shows the average total seed frequency comparison between the greedy seed selection mechanism with ERS and OSS with naïve seeds. As shown in the figure, OSS performs slightly better under small error tolerance requirements and quickly exceeds ERS as error tolerance requirement increases. OSS out-performs ERS under small error tolerance requirements because the naïve ERS seed selector always start with the first MEM and picks seeds consecutively while OSS is more flexible as it scans through all MEMs and picks the less frequent ones. When the error requirement is large, however, the flexibility of OSS reduces as it is under the stress to distribute a limited amount of base pairs among more seeds. As a result, OSS start to select increasingly shorter seeds. ERS, on the other hand, prevails with long seeds as long seeds provide high error resilience. When the error tolerance requirement is 4, ERS provides a total of 1.17x reduction in average seed frequency.

While the total seed frequency reduction of ERS might not be significant on *E. coli*, ERS is still computationally more efficient. To obtain the optimal set of naïve seeds through OSS, we need to obtain the frequency of all possible seeds. That requires L^2 accesses to the seed database. Assuming the seed database is organized as a suffix tree or a Burrows-Wheeler transformed suffix array, each access to the seed database requires many accesses to the main memory. Since each memory access takes hundreds of CPU cycles to complete, in practice, OSS has a high operational cost. ERS on the contrary, is a greedy seed selection algorithm and only greedily extracts consecutive MEMs from a read. Therefore, ERS is much more light-weight in practice than OSS. We benchmarked both OSS and ERS using the Burrows-Wheeler transformed suffix array from the SDSL library and observe that ERS on average is 12.4x faster than OSS.

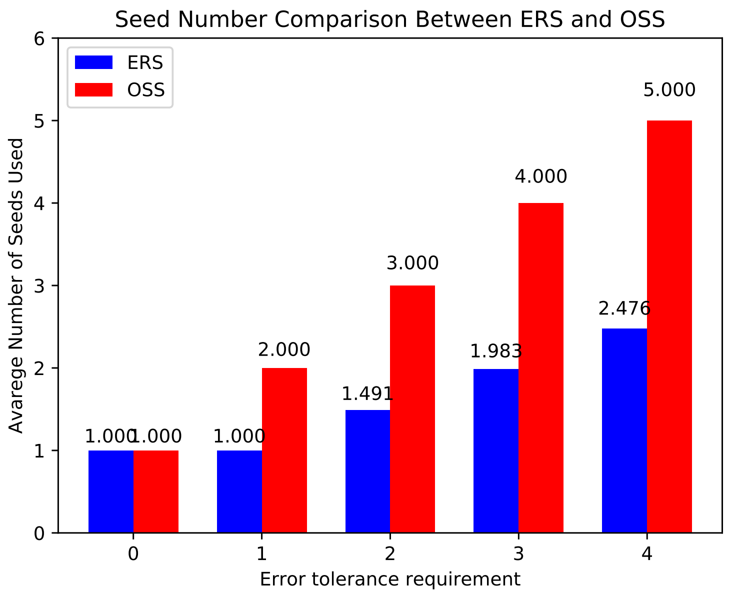


Figure 5.10: Comparison of the average number of seeds required to satisfy the error tolerance requirement in ERS and OSS.

A major reason that OSS achieves comparable results with ERS on *E. coli* is because of the small reference size of *E. coli*. Figure 5.10 displays the average number of seeds needed to meet the error tolerance requirement in ERS and OSS, respectively. Comparing Figure 5.9 and Figure 5.10, one can observe that in OSS the average total seed frequency is sometimes smaller than the number of required seeds. This suggests OSS can deliberately select seeds with errors in them, which may have zero frequencies. ERS, on the contrary, cannot select seeds that do not appear in the reference. Hence with ERS, the average seed frequency is always greater than or equal to the number of seeds selected. However, with larger and more complex reference genomes, seeds only have zero frequencies when they have sufficient length. Therefore it is harder for OSS to select erroneous seeds without consuming many base pairs. Therefore, we expect with larger and more complex reference genomes, the benefit of ERS over OSS to be more profound.

5.5 Discussion

Error Resilient Seeds has two major limitations. First, Error resilient seeds does not work well in regions that have genomic repeats. Due to evolution, mobile elements, which are the majority of genomic repeats, tend to differ slightly with each other. This suggests that many sequences in regions of genomic repeats have highly similar non-trivial neighbors. As a result, seeds from genomic repeats are inclined to have low error resilience. Even worse, due to their repetitive nature, seeds from genomic repeats often also have high frequency. Hence, error resilient seeds provide little improvement over mapping reads from repeats. A potential remedy of this challenge would be simply remove genomic repeats from the reference and keep them in a separate repeat database. The repeat database keeps a single representative instance for each repeat. During mapping, if a seed of a read appears in the repeat database, then we first attempt to map the read separately to the repeat database. If a mapping is found in the repeat database, then we brand the read as part of a repeat and store it separately for later analysis. Otherwise, we proceed to mapping the read to the repeat-masked reference genome using error resilient seeds.

While being a challenge for mapping, sharing highly similar non-trivial neighbors also marks a unique trait for genomic repeats. As a result, this enables us to use error resilient seeds to be used to identify new genomic repeats whose instances might not have high frequencies individually, but there are many slightly different instances and collectively they constitute large frequencies.

Second, constructing the neighbor database has a high computational cost. While most long sequences do not have many non-trivial neighbors, when seeds are short, 9 base pairs for example, each seed has many neighbors. This is because, at low depths, the suffix trie is almost full, where every sequence permutation exists in the suffix trie. For low depth nodes in the suffix trie, the number of its neighbors basically equals to the number of unique permutations of the sequence after applying up to D edits. Because the suffix trie is almost full, each permutation after editing is almost guaranteed to occur in the suffix trie.

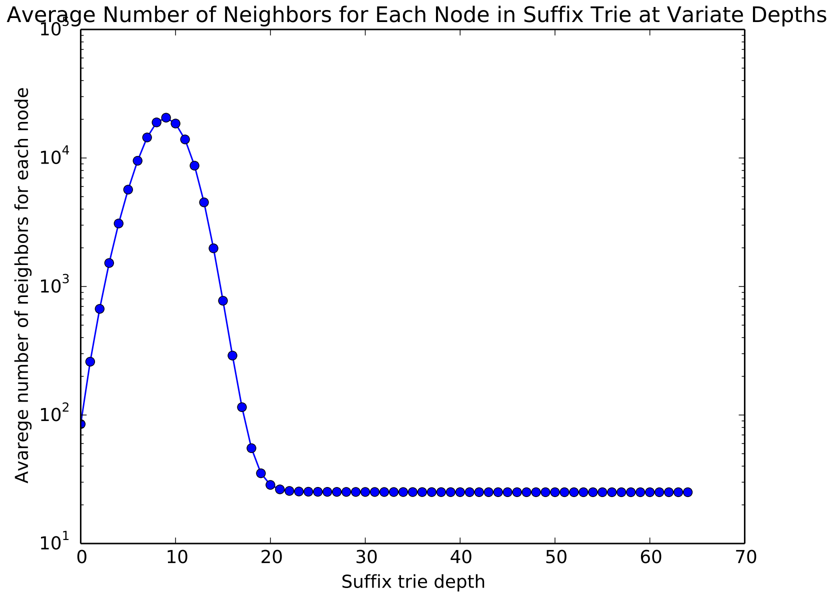


Figure 5.11: The average number of neighbors for each node at variate depths of the suffix trie.

Figure 5.11 shows the average number of neighbors for each node at variate depths in the suffix trie of *E. coli* with $D = 4$. From the figure we can observe that the average number of neighbors per node peaks at around depth 9. Figure 5.12 shows the occupancy rate of nodes

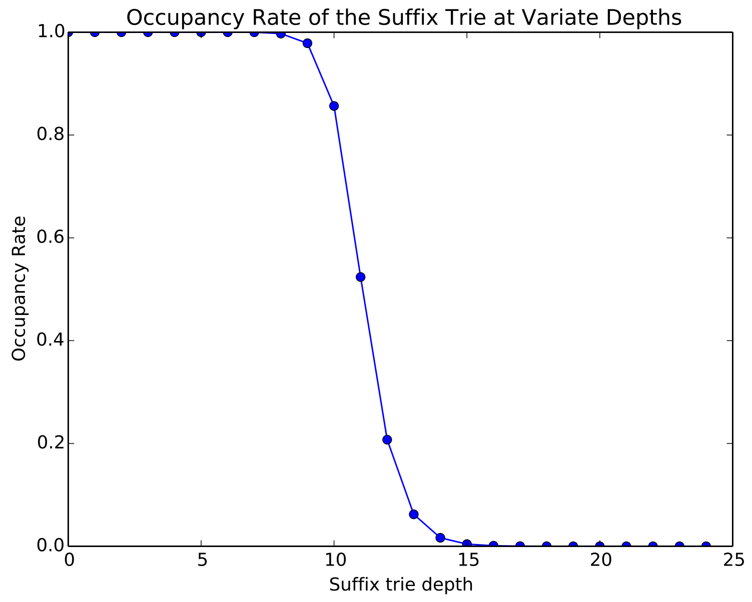


Figure 5.12: The occupancy rate of the suffix trie at variate depths.

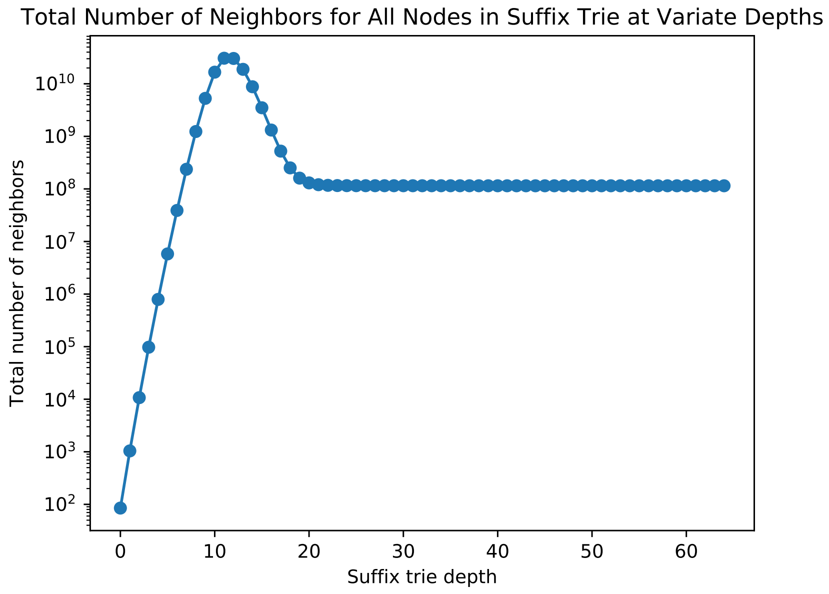


Figure 5.13: The total number of neighbors for all nodes at variate depths of the suffix trie.

at variate depths in the suffix trie. The occupancy rate at depth p is defined as the number of nodes at the depth divide by 4^p, which is the maximum number of nodes possible at depth p. As shown in the figure, up until depth 9, the suffix trie almost have a full occupancy (the occupancy rate of depth 9 is 97%) and it quickly drops as the depths increases. For *E. coli*, after depth 14, the number of nodes in the suffix trie at a depth stabilizes at 4.57 million, which is almost the reference length of *E. coli*. Figure 5.13 shows the total number of neighbors for all nodes at variate depths. As shown in Figure 5.13, the total number of neighbors of all nodes peaks between depth 11 and 12 at around 31 billion, and quickly decreases until stabilizing at around 115 million.

It takes around 20 hours to generate the neighbor database with $D = 4$ and $P = 60$ for *E. coli* on a server of 40 cores and 80 threads. With larger organisms (e.g. human) and larger D. The profiling time will increase significantly. To summarize, even though the time complexity of constructing the neighbor database is $O(|\Sigma|^2 \cdot M)$, building the neighbor database can still be a computationally expensive process simply due to the large magnitude of M.

Fortunately, the neighbor database only needs to be constructed once for each reference. Further because the computations of neighbors for nodes within the same depth of the suffix trie are independent, the process of building the neighbor database for larger organisms can be deployed on scale-out cloud architectures (such as spark) to maximize parallelism.

5.6 Conclusion

In this work, we reveal the key limitation of using long seeds in read mapping: seeds are not error resilient. To solve this problem, we proposed a new concept, error resilient seeds, which enables each seed to tolerate more than one error. As a result, even with long seeds, mappers may achieve high error tolerance. We proved that the error resilience of a seed is equivalent to the minimum edit distance among its non-trivial neighbors. We also proposed an algorithm that finds all neighbors within a small edit distance radius for all seeds in linear time. Finally, we introduced a seed selection mechanism using error resilient seeds and show that it out performs current state-of-the-art seed selection mechanism using naïve seeds.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The observations and computational techniques included in this dissertation help us obtain a deeper understanding of computational challenges in mapping NGS reads and pave the way to further improve the performance of NGS read mappers. Especially, we provide solutions to reduce the impact of the vast amount of false mappings generated by mappers with high error tolerance.

In part one, we displayed that by drawing more and shorter seeds from reads, in an effort to provide a greater error tolerance guarantee, as well as to support a wider scope in searching for secondary mappings, fully-sensitive read mappers examine a vast and rapidly increasing number of false mappings. We further show that the majority of such false mappings exhibit little similarity between the read and the reference. To alleviate the problem, we developed a SIMD-friendly bit-vectorized filter, shifted hamming distance (SHD), that quickly rejects false mappings. SHD examines if there exist multiple identical substrings between the read and the reference that can be stitched together to cover the entire read. We further optimized our implementation to take advantage of modern computer architectures and drastically improved the speed of our algorithm. We compared our implementation against state-of-the-art vectorized verification implementations and showed that SHD achieved significant speedups with great filtering accuracy.

Besides quickly filtering out obviously false mappings, we also extended a previously proposed, efficient dynamic programming algorithm, the Landau-Vishkin algorithm. While fast, the Landau-Vishkin algorithm has limited adoptions by modern mappers due to its inflexibility to support penalty scoring schemes besides edit distance. Most importantly, it has not been shown to support affine gap penalties. In our work, LEAP, we proposed the leaping toad problem, which is a generalization of the approximate string matching problem with non-negative penalty scoring schemes. We show that approximate string matching is a sub-problem of the leaping toad problem and the Landau-Vishkin algorithm can be applied to solve the leaping toad problem. Thus, we indirectly prove that the Landau-Vishkin algorithm can be extended to support affine gap penalty scoring schemes. We also provided an efficient implementation of the Landau-Viskin algorithm with the optimization of finding the leading bit of ‘1’ in a binary sequence using a bit-vectorized de Bruijn sequence technique. We showed that our implementation is significantly faster than state-of-the-art vectorized verification implementations.

In part two, we illustrated that reference locations are unevenly distributed among seeds. We showed that while extending a seed longer reduces the frequency of the seed, the frequency reduction is drastically divergent across different seeds at different lengths. We further showed that always drawing seeds of equal lengths is inefficient as infrequent seeds could have yielded base pairs to extend the length of frequent seeds and decrement the overall seed frequency. We introduced the concept of optimal seeds which are set of non-overlapping seeds with variate seed lengths that as a whole have the smallest overall seed frequency. To overcome the vast search space of all possible seed combinations in a read, we proposed optimal seeds solver, an efficient dynamic programming algorithm that finds the optimal seeds in a read in linear complexity, under the assumption that seed frequencies can be obtained in constant time. We compared optimal seeds solver against state-of-the-art seeding mechanisms and showed that optimal seeds solver drastically reduces the total seed frequencies in seed-and-extend read mappers, while attaining high error tolerance.

To further reduce seed frequencies, we observed that read mappers do not always need to follow the pigeonhole principle and draw $e + 1$ non-overlapping seeds to tolerate e errors. We recognized that a mapper can achieve the same level of error tolerance with fewer seeds as long as some seeds require multiple errors to become faulty. Based on this observation, we proposed error resilient seeds, a novel concept which enables each seed to tolerate more than one error. We derived that the error resilience of a seed equals the smallest edit-distance to its nearest non-trivial neighbor. To efficiently construct the error resilience database, we developed a novel algorithm that finds all neighbors of all seeds within a small edit distance radius in linear time. We further augmented the algorithm to piggyback the computation of triviality of its neighbors without increasing the computational complexity. We proposed a greedy seeding mechanism with error resilient seeds and showed that under equal seed tolerance requirement, error resilient seeds out-perform optimal seeds solver. We concluded the chapter by discussing limitations as well as further potentials of error resilient seeds.

6.2 Discussion

It has been over ten years since the advent of the first NGS platform. Since then, there has been numeral advances in sequencing technology. One of the most influential improvements to NGS platforms is the invention of paired-end sequencing technology. Paired-end sequencing technology generates reads in pairs which are the two ends of a longer DNA fragment. The lengths of such DNA fragments are usually between 500 to 1000 base pairs. With paired-end sequencing technology, we can now accurately map many reads from short genomic repeats, which were previously impossible to map with high precision. Since short genomic repeats are usually on the scale of a hundred base pairs and long DNA fragments of paired-end sequencing technology usually have more than 500 base pairs, it is unlikely that both ends of the DNA fragment are included in a single short genomic repeat. The non-repetitive end of the read pair then can serve as a guide to help with the mapping of the repetitive end.

Paired-end sequencing technology also helps improve the mapping efficiency of normal reads

(reads not enveloped by genetic repeats). As the two ends must not be separated by more than a few thousands of base pairs, a candidate mapping position must gain support of seeds from of both reads. As a result, a mapper can map the easier-to-map end first and use the mapping result as a guide to locate the mapping position of the harder-to-map end. When both ends are hard to map, a mapper can resort to proximity filters (including variants used in SNAP and BWA-MEM alike), which usually have better performance than in mapping single-end reads (potential mappings must gain support from at least one seed from each read, which is a much tighter requirement than in mapping single-end reads).

Both LEAP and ERS can be applied to improve mapping of paired-end reads. LEAP improves the speed of the approximate-string-match process, which is essential for both paired-end sequencing technology and single-end sequencing technology; while ERS allows a mapper to confidently use longer, less frequent yet error resilient seeds, which further reduces the workload of proximity filters in mapping paired-end reads.

With paired-end sequencing technology, however, we still cannot accurately map read pairs from longer genomic repeats such as Long Interspersed Nucleotide Elements (LINE) or tandem repeats. To solve this problem, third generation sequencing technologies are developed which aim to generate longer (on the scale of multiple thousands to tens of thousands of base pairs) reads. Currently, third generation sequencing technology can be grouped into two separate categories: 1) generating continuous long reads with high error rates, such as Nanopore sequencing technology [20] and 2) outputting many short reads that originate from a single long DNA fragment (also called *linked-reads* technology), such as 10X sequencing technology [106].

Both LEAP and ERS can be easily applied to linked-reads technology as they serve similar purposes in linked-reads technology as they do in paired-end sequencing technology. ERS could also potentially be applied to improve mapping of highly erroneous long reads, as a single highly error resilient seed from the read might contain enough information to pin-point high quality mapping candidates. However, generating error resilient database with large error-resilient thresholds is challenging due to the super-linear growth of neighbor counts per node in the suffix trie as the error resilient threshold increases. Therefore, the potential of applying ERS in mapping erroneous long reads needs to be further investigated.

Overall, computational techniques proposed in this dissertation is built around the seed-and-extend mapping methodology, which works best with reads that have low error rates. Among the four techniques, LEAP and ERS are more technologically independent and has the most potential to be applied to future sequencing technologies, especially to linked-reads technologies.

6.3 Future Work

There are four potential directions to further extend the studies in this dissertation. First, while the computational techniques included in this dissertation each provides compelling results, it is unknown if they share any synergy with each other and can be combined together into a single read mapper. Specifically, it is important to study and understand how to tailor customized mapping strategies for each read to achieve optimal performance. Second, it is worth exploring the potential of applying error resilient seeds to pseudoalignment. Currently, mappers that employ pseudoalignment use systematic sampling methods to detect spread-out error-free seeds in a long read [42, 53]. Most of the sampling mechanisms exploit MinHash sketches[16] to achieve a steady sampling coverage across the entire reference genome to improve sampling efficiency [15, 70, 80]. Error resilient seeds adds a new dimension to devising the sampling method. As seeds with higher error resilience are less likely to be affected by errors, a potential seed sampling mechanism that is more inclined to sample seeds with greater error resilience may increase the likelihood of selecting seeds that are truly error free. A potential option is to replace the hash function with ERS, which samples seeds based on error resiliency and consistently sample highly resilient seeds. Seeds with high error resiliency is more likely to be error-free, while capturing more error-free seeds increases the confidence of finding correct mappings with pseudoalignment. Third, it is worthwhile to investigate how to optimally divide a read into error resilient seeds. If a read is divided correctly, a mapper may obtain not only long, infrequent seeds, but also highly error resilient seeds as well. However this also complicates the seed selection process, as extending and/or reducing the length of a seed impacts both the frequency as well as the error resilience of the seed. Finally, it is intriguing to examine if the essence of error resilient seeds, namely finding similar non-trivial sequences in the reference, can be applied to other problems, such as RNA quantification with pseudoalignment, identifying homologous genome snippets across organisms, or finding motifs in genomes.

Bibliography

[1] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467:1061–1073, 2010. 1, 3.4, 4.1

[2] 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, Nov 2012. doi: 10.1038/nature11632. URL `http://dx.doi.org/10.1038/nature11632`. (document), 2.1, 2.4

[3] 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:e41, 2011. 1.2, 1.3, 2.3.2, 4.1, 4.2, 4.4

[4] Can Alkan, Jeffrey M Kidd, Tomas Marques-Bonet, Gozde Aksay, Francesca Antonacci, Fereydoun Hormozdiari, Jacob O Kitzman, Carl Baker, Maika Malig, Onur Mutlu, S. Cenk Sahinalp, Richard A Gibbs, and Evan E Eichler. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nat Genet*, 41:1061–1067, 2009. 1.1.1, 2.1, 2.3.2, 2.4, 4.1

[5] Francesca Antonacci, Jeffrey M Kidd, Tomas Marques-Bonet, et al. Characterization of six human disease-associated inversion polymorphisms. *Hum Mol Genet*, 18:2555–2566, 2009. 1

[6] Francesca Antonacci, Jeffrey M Kidd, Tomas Marques-Bonet, et al. A large and complex structural polymorphism at 16p12.1 underlies microdeletion disease risk. *Nat Genet*, 42: 745–750, 2010. 1

[7] J. A. Bailey, A. M. Yavor, H. F. Massa, B. J. Trask, and E. E. Eichler. Segmental duplications: organization and impact within the current human genome project assembly. *Genome Res*, 11:1005–1017, 2001. 1

[8] J. A. Bailey, J. M. Kidd, and E. E. Eichler. Human copy number polymorphic genes. *Cytogenet Genome Res*, 123:234–243, 2008. 1

[9] Jeffrey A Bailey and Evan E Eichler. Primate segmental duplications: crucibles of evolution, diversity and disease. *Nat Rev Genet*, 7:552–564, 2006. 1

[10] Jeffrey A Bailey, Zhiping Gu, Royden A Clark, Knut Reinert, Rhea V Samonte, Stuart Schwartz, Mark D Adams, Eugene W Myers, Peter W Li, and Evan E Eichler. Recent segmental duplications in the human genome. *Science*, 297:1003–1007, 2002. 1

[11] Jeffrey A Bailey, Amy M Yavor, Luigi Viggiano, Doriana Misceo, Juliann E Horvath, Nicoletta Archidiacono, Stuart Schwartz, Mariano Rocchi, and Evan E Eichler. Human-

specific duplication and mosaic transcripts: the recent paralogous structure of chromosome 22. *Am J Hum Genet*, 70:83–100, 2002. 1

[12] Jeffrey A Bailey, Robert Baertsch, W. James Kent, David Haussler, and Evan E Eichler. Hotspots of mammalian chromosomal evolution. *Genome Biol*, 5:R23, 2004. 1

[13] Brona Brejova, Daniel G. Brown, and Tomas Vinar. Vector seeds: An extension to spaced seeds. *Journal of Computer and System Sciences*, 70(3):364 – 380, 2005. ISSN 0022-0000. doi: https://doi.org/10.1016/j.jcss.2004.12.008. URL `http://www.sciencedirect.com/science/article/pii/S0022000004001527`. Special Issue on Bioinformatics II. 1.2

[14] Sydney Brenner, Maria Johnson, John Bridgham, George Golda, David H. Lloyd, Davida Johnson, Shujun Luo, Sarah McCurdy, Michael Foy, Mark Ewan, Rithy Roth, Dave George, Sam Eletr, Glenn Albrecht, Eric Vermaas, Steven R. Williams, Keith Moon, Timothy Burcham, Michael Pallas, Robert B. DuBridge, James Kirchner, Karen Fearon, Jen i Mao, , and Kevin Corcoran. Gene expression analysis by massively parallel signature sequencing (mpss) on microbead arrays. *Nat Biotechnol*, 18(6):630–4, 2000. 1

[15] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8132-2. URL `http://dl.acm.org/citation.cfm?id=829502.830043`. 6.3

[16] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In Raffaele Giancarlo and David Sankoff, editors, *Combinatorial Pattern Matching*, pages 1–10, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45123-5. 6.3

[17] Daniel G. Brown. A survey of seeding for sequence alignment, 2007. 1.2

[18] S. Canzar and S. L. Salzberg. Short read mapping: An algorithmic tour. *Proceedings of the IEEE*, 105(3):436–458, March 2017. ISSN 0018-9219. doi: 10.1109/JPROC.2015.2455551. 1.1.1

[19] Jian-Qun Chen, Ying Wu, Haiwang Yang, Joy Bergelson, Martin Kreitman, and Dacheng Tian. Variation in the ratio of nucleotide substitution and indel rates across genomes in mammals and bacteria. *Molecular Biology and Evolution*, 26(7):1523–1531, 2009. doi: 10.1093/molbev/msp063. URL `+http://dx.doi.org/10.1093/molbev/msp063`. 3.1

[20] James Clarke, Hai-Chen Wu, Lakmal Jayasinghe, Alpesh Patel, Stuart Reid, and Hagan Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nat Nanotechnol*, 4:265–270, 2009. 6.2

[21] Miklós Csűrös and Bin Ma. Rapid homology search with two-stage extension and daughter seeds. In Lusheng Wang, editor, *Computing and Combinatorics*, pages 104–114, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31806-4. 1.2

[22] Miklós Csűrös and Bin Ma. Rapid homology search with neighbor seeds. *Algorithmica*, 48(2):187–202, Jun 2007. ISSN 1432-0541. doi: 10.1007/s00453-007-0062-y. URL `https://doi.org/10.1007/s00453-007-0062-y`. 1.2

[23] Jeff Daily. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17:81, 2016. doi: 10.1186/s12859-016-0930-z. URL `http://dx.doi.org/10.1186/s12859-016-0930-z`. 1.3, 3.4

[24] Edans F. De O. Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba C. M. A. De Melo. Masa: A multiplatform architecture for sequence aligners with block pruning. *ACM Trans. Parallel Comput.*, 2(4):28:1–28:31, February 2016. ISSN 2329-4949. doi: 10.1145/2858656. URL `http://doi.acm.org/10.1145/2858656`. 1.3

[25] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucl. Acids Res.*, 1999. 2.3.2

[26] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9:11, 2008. doi: 10.1186/1471-2105-9-11. URL `http://dx.doi.org/10.1186/1471-2105-9-11`. 1.3, 2.1, 2.2, 2.4, 2.5, 3.4

[27] Lavinia Egidi and Giovanni Manzini. Multiple seeds sensitivity using a single seed with threshold. *Journal of Bioinformatics and Computational Biology*, 13(04):1550011, 2015. doi: 10.1142/S0219720015500110. URL `http://www.worldscientific.com/doi/abs/10.1142/S0219720015500110`. PMID: 25747382. 1.2

[28] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23:156–161, 2007. 1.3, 1.3, 2.1

[29] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0850-2. URL `http://dl.acm.org/citation.cfm?id=795666.796543`. 4.3.3

[30] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. 4.4

[31] Sara Goodwin, John D. McPherson, and W. Richard McCombie. Coming of age: Ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, 6 2016. ISSN 1471-0056. doi: 10.1038/nrg.2016.49. 1

[32] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705 – 708, 1982. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(82)90398-9. URL `http://www.sciencedirect.com/science/article/pii/0022283682903989`. 1.3

[33] Richard E Green, Johannes Krause, Adrian W Briggs, Tomislav Maricic, Udo Stenzel, Martin Kircher, Nick Patterson, Heng Li, Weiwei Zhai, Markus Hsi-Yang Fritz, Nancy F Hansen, Eric Y Durand, Anna-Sapfo Malaspinas, Jeffrey D Jensen, Tomas Marques-Bonet, Can Alkan, Kay Prfer, Matthias Meyer, Hernn A Burbano, Jeffrey M Good, Rigo Schultz, Ayinuer Aximu-Petri, Anne Butthof, Barbara Hber, Barbara Hffner, Madlen Siegemund, Antje Weihmann, Chad Nusbaum, Eric S Lander, Carsten Russ, et al. A draft sequence of the Neandertal genome. *Science*, 328:710–722, 2010. 1

[34] 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. *Nat Methods*, 7:576–577, 2010. 1.1.1

[35] Ayat Hatem, Doruk Bozdağ, Amanda E. Toland, and Ümit V. Çatalyürek. Benchmarking short sequence mapping tools. *BMC Bioinformatics*, 14(1):184, Jun 2013. ISSN 1471-2105. doi: 10.1186/1471-2105-14-184. URL `https://doi.org/10.1186/1471-2105-14-184`. 1.1.1

[36] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. Bless: Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, 30(10):1354–1362, 2014. doi: 10.1093/bioinformatics/btu030. URL `+http://dx.doi.org/10.1093/bioinformatics/btu030`. 1

[37] Yun Heo, Anand Ramachandran, Wen-Mei Hwu, Jian Ma, and Deming Chen. Bless 2: accurate, memory-efficient and fast error correction method. *Bioinformatics*, 32(15):2369–2371, 2016. doi: 10.1093/bioinformatics/btw146. URL `+http://dx.doi.org/10.1093/bioinformatics/btw146`. 1

[38] K. Hou, H. Wang, and W. C. Feng. Aalign: A simd framework for pairwise sequence alignment on x86-based multi-and many-core processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 780–789, May 2016. doi: 10.1109/IPDPS.2016.115. 1.3

[39] Heikki Hyyro, Yoan J. Pinzon, and Ayumi Shinohara. Fast bit-vector algorithms for approximate string matching under indel distance. In Peter Vojts, Mria Bielikov, Bernadette Charron-Bost, and Ondrej Skora, editors, *SOFSEM*, volume 3381 of *Lecture Notes in Computer Science*, pages 380–384. Springer, 2005. ISBN 3-540-24302-X. URL `http://dblp.uni-trier.de/db/conf/sofsem/sofsem2005.html#HyyroPS05`. 1.3

[40] Ignacio L. Ibarra and Francisco Melo. Interactive software tool to comprehend the calculation of optimal sequence alignments with dynamic programming. *Bioinformatics*, 26 (13):1664–1665, 2010. doi: 10.1093/bioinformatics/btq252. URL `+http://dx.doi.org/10.1093/bioinformatics/btq252`. 1.1

[41] Intel. Intel architecture instruction set extensions programming reference. Technical Report 319433-014, Intel, August 2012. URL `http://download-software.intel.com/sites/default/files/319433-014.pdf`. 2.1

[42] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. In S. Cenk Sahinalp, editor, *Research in Computational Molecular Biology*, pages 66–81, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56970-3. 6.3

[43] S.M. Kiełbasa, Raymond Wan, Kengo Sato, Paul Horton, and Martin C. Frith. Adaptive seeds tame genomic sequence comparison. *Genome Research*, 21(3):487–493, 2011. doi: 10.1101/gr.113985.110. 1.2, 4.1, 4.1, 4.2, 4.5

[44] Matija Korpar and Sikic Mile. Sw#-gpu-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, 2013. doi: 10.1093/bioinformatics/btt410. URL

+http://dx.doi.org/10.1093/bioinformatics/btt410. 1.3

[45] Gregory Kucherov, Kamil Salikhov, and Dekel Tsur. Approximate string matching using a bidirectional index. In AlexanderS. Kulikov, SergeiO. Kuznetsov, and Pavel Pevzner, editors, *Combinatorial Pattern Matching*, volume 8486 of *Lecture Notes in Computer Science*, pages 222–231. Springer International Publishing, 2014. ISBN 978-3-319-07565-5. doi: 10.1007/978-3-319-07566-2_23. URL http://dx.doi.org/10.1007/978-3-319-07566-2_23. 4.4

[46] Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of algorithms*, 10(2):157–169, 1989. 3.1, 5.3.1

[47] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Method*, 9:357–359, 2012. 1.1.1, 2.3.2, 4.4

[48] Wan-Ping Lee, Michael P. Stromberg, Alistair Ward, Chip Stewart, Erik P. Garrison, and Gabor T. Marth. Mosaik: A hash-based algorithm for accurate next-generation sequencing short-read mapping. *PLOS ONE*, 9(3):1–11, 03 2014. doi: 10.1371/journal.pone.0090581. URL https://doi.org/10.1371/journal.pone.0090581. 1.1.1

[49] Charles E Leiserson, Harald Prokop, and Keith H Randall. Using de bruijn sequences to index a 1 in a computer word. *Available on the Internet from http://supertech. csail. mit. edu/papers. html*, 1998. 3.3.5

[50] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 1966. 1.1

[51] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. 2013. 1.1.1, 4.4

[52] Heng Li. Bfc: correcting illumina sequencing errors. *Bioinformatics*, 31(17):2885–2887, 2015. doi: 10.1093/bioinformatics/btv290. URL +http://dx.doi.org/10.1093/bioinformatics/btv290. 1

[53] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016. doi: 10.1093/bioinformatics/btw152. URL +http://dx.doi.org/10.1093/bioinformatics/btw152. 6.3

[54] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26:589–595, 2010. 2.3.2

[55] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010. doi: 10.1093/bib/bbq015. URL +http://dx.doi.org/10.1093/bib/bbq015. 1.1.1

[56] Heng Li, Jue Ruan, and Richard Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18:1851–1858, 2008. 1.1.1

[57] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25:2078–2079, 2009. 1.1

[58] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide

alignment program. *Bioinformatics*, 24:713–714, 2008. 1.1.1

[59] Ruiqiang Li, Chang Yu, Yingrui Li, Tak Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang 0004. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. 1.1.1, 2.3.2

[60] Eun-Cheon Lim, Jonas Mller, Jrg Hagmann, Stefan R. Henz, Sang-Tae Kim, and Detlef Weigel. Trowel: a fast and accurate error correction module for illumina sequencing reads. *Bioinformatics*, 30(22):3264–3265, 2014. doi: 10.1093/bioinformatics/btu513. URL `+http://dx.doi.org/10.1093/bioinformatics/btu513`. 1

[61] Hao Lin, Zefeng Zhang, Michael Q Zhang, Bin Ma, and Ming Li. ZOOM! zillions of oligos mapped. *Bioinformatics*, 24:2431–2437, 2008. 1.1.1

[62] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. In *Journal of biomedicine & biotechnology*, 2012. 1

[63] Y. Liu and B. Schmidt. Swaphi: Smith-waterman protein database search on xeon phi coprocessors. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 184–185, June 2014. doi: 10.1109/ASAP.2014.6868657. 1.3

[64] Y. Liu, T. T. Tran, F. Lauenroth, and B. Schmidt. Swaphi-ls: Smith-waterman algorithm on xeon phi coprocessors for long dna sequences. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 257–265, Sept 2014. doi: 10.1109/CLUSTER.2014.6968772. 1.3

[65] Yongchao Liu and Bertil Schmidt. Gswabe: faster gpu-accelerated sequence alignment with optimal alignment retrieval for short dna sequences. *Concurrency and Computation: Practice and Experience*, 27(4):958–972, 2015. ISSN 1532-0634. doi: 10.1002/cpe.3371. URL `http://dx.doi.org/10.1002/cpe.3371`. 1.3

[66] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC Bioinformatics*, 14(1):117, Apr 2013. ISSN 1471-2105. doi: 10.1186/1471-2105-14-117. URL `https://doi.org/10.1186/1471-2105-14-117`. 1.3

[67] Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002. 1.2, 4.2, 4.4, 4.4

[68] Denise Mak, Yevgeniy Gelfand, and Gary Benson. Indel seeds for homology search. *Bioinformatics*, 22(14):e341–e349, 2006. doi: 10.1093/bioinformatics/btl263. URL `+http://dx.doi.org/10.1093/bioinformatics/btl263`. 1.2

[69] Svetlin A Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9 Suppl 2:S10, 2008. 1.3, 2.1

[70] Guillaume Marçais, David Pellow, Daniel Bork, Yaron Orenstein, Ron Shamir, and Carl Kingsford. Improving the performance of minimizers and winnowing schemes. *Bioinformatics*, 33(14):i110–i117, 2017. doi: 10.1093/bioinformatics/btx235. URL

+http://dx.doi.org/10.1093/bioinformatics/btx235. 6.3

[71] Santiago Marco-Sola, Michael Sammeth, Roderic Guig, and Paolo Ribeca. The gem mapper: fast, accurate and versatile alignment by filtration. *Nat Methods*, 9(12):1185–1188, 2012. doi: 10.1038/nmeth.2221. URL http://dx.doi.org/10.1038/nmeth.2221. 1.1.1, 1.2, 4.1, 4.2, 4.4

[72] Tomas Marques-Bonet, Jeffrey M Kidd, Mario Ventura, Tina A Graves, Ze Cheng, LaDeana W Hillier, Zhaoshi Jiang, Carl Baker, Ray Malfavon-Borja, Lucinda A Fulton, Can Alkan, Gozde Aksay, Santhosh Girirajan, Priscillia Siswara, Lin Chen, Maria Francesca Cardone, Arcadi Navarro, Elaine R Mardis, Richard K Wilson, and Evan E Eichler. A burst of segmental duplications in the genome of the African great ape ancestor. *Nature*, 457:877–881, 2009. 1

[73] Zaharia Matei, Bolosky William J., Curtis Kristal, Fox Armando, Patterson David, Shenker Scott, Stoica Ion, Karp Richard M., and Sittler Taylor. Faster and more accurate sequence alignment with snap. *eprint arXiv*, 2011. 1.1.1

[74] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999. ISSN 0004-5411. doi: 10.1145/316542.316550. URL http://doi.acm.org/10.1145/316542.316550. 1.3, 1.3, 2.4

[75] 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*, 48:443–453, 1970. 1.1, 1.3

[76] Laurent Noé. Spaced seeds bibliography. URL http://cristal.univ-lille.fr/~noe/spaced_seeds.html. 1.2

[77] David Reich, Richard E Green, Martin Kircher, Johannes Krause, Nick Patterson, Eric Y Durand, Bence Viola, Adrian W Briggs, Udo Stenzel, Philip L F Johnson, Tomislav Maricic, Jeffrey M Good, Tomas Marques-Bonet, Can Alkan, Qiaomei Fu, Swapan Mallick, Heng Li, Matthias Meyer, Evan E Eichler, Mark Stoneking, Michael Richards, Sahra Talamo, Michael V Shunkov, Anatoli P Derevianko, Jean-Jacques Hublin, Janet Kelso, Montgomery Slatkin, and Svante Pbo. Genetic history of an archaic hominin group from Denisova Cave in Siberia. *Nature*, 468:1053–1060, 2010. 1

[78] Knut Reinert, Ben Langmead, David Weese, and Dirk J. Evers. Alignment of next-generation sequencing reads. *Annual Review of Genomics and Human Genetics*, 16(1):133–151, 2015. doi: 10.1146/annurev-genom-090413-025358. URL https://doi.org/10.1146/annurev-genom-090413-025358. PMID: 25939052. 1.1.1

[79] Paolo Ribeca. *Short-Read Mapping*, pages 107–125. Springer New York, New York, NY, 2012. ISBN 978-1-4614-0782-9. doi: 10.1007/978-1-4614-0782-9_7. URL https://doi.org/10.1007/978-1-4614-0782-9_7. 1.1.1

[80] 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. doi: 10.1093/bioinformatics/bth408. URL +http://dx.doi.org/10.1093/bioinformatics/bth408. 6.3

[81] M. Roytberg, A. Gambin, L. Noe, S. Lasota, E. Furletova, E. Szczurek, and G. Kucherov. On subset seeds for protein alignment. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(3):483–494, July 2009. ISSN 1545-5963. doi: 10.1109/TCBB.2009.4. 1.2

[82] Steve Rozen, Helen Skaletsky, Janet D Marszalek, Patrick J Minx, Holland S Cordum, Robert H Waterston, Richard K Wilson, and David C Page. Abundant gene conversion between arms of palindromes in human and ape Y chromosomes. *Nature*, 423:873–876, 2003. 1

[83] Matthew Ruffalo, Thomas LaFramboise, and Mehmet Koyutrk. Comparative analysis of algorithms for next-generation sequencing read alignment. *Bioinformatics*, 27(20):2790–2796, 2011. doi: 10.1093/bioinformatics/btr477. URL `http://dx.doi.org/10.1093/bioinformatics/btr477`. 1.1.1

[84] Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: Accurate mapping of short color-space reads. *PLoS Comput Biol*, 5:e1000386, 2009. 1.1.1, 2.3.2, 4.1

[85] Aylwyn Scally, Julien Y Dutheil, LaDeana W Hillier, Gregory E Jordan, Ian Goodhead, Javier Herrero, Asger Hobolth, Tuuli Lappalainen, Thomas Mailund, Tomas Marques-Bonet, Shane McCarthy, Stephen H Montgomery, Petra C Schwalie, Y. Amy Tang, Michelle C Ward, Yali Xue, Bryndis Yngvadottir, Can Alkan, Lars N Andersen, Qasim Ayub, Edward V Ball, Kathryn Beal, Brenda J Bradley, Yuan Chen, Chris M Clee, Stephen Fitzgerald, Tina A Graves, Yong Gu, Paul Heath, Andreas Heger, et al. Insights into hominid evolution from the gorilla genome sequence. *Nature*, 483:169–175, 2012. 1

[86] Melanie Schirmer, Umer Z. Ijaz, Rosalinda D'Amore, Neil Hall, William T. Sloan, and Christopher Quince. Insight into biases and sequencing errors for amplicon sequencing with the illumina miseq platform. *Nucleic Acids Research*, 43(6):e37, 2015. doi: 10.1093/nar/gku1341. URL `+http://dx.doi.org/10.1093/nar/gku1341`. 1

[87] Melanie Schirmer, Rosalinda D'Amore, Umer Z. Ijaz, Neil Hall, and Christopher Quince. Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data. *BMC Bioinformatics*, 17(1):125, Mar 2016. ISSN 1471-2105. doi: 10.1186/s12859-016-0976-y. URL `https://doi.org/10.1186/s12859-016-0976-y`. 1

[88] Sergey Sheetlin, Yonil Park, Martin C. Frith, and John L. Spouge. Alp & falp: C++ libraries for pairwise local alignment e-values. *Bioinformatics*, 32(2):304–305, 2016. doi: 10.1093/bioinformatics/btv575. URL `+http://dx.doi.org/10.1093/bioinformatics/btv575`. 1.3

[89] D. F. Simola and J. Kim. Sniper: improved SNP discovery by multiply mapping deep sequenced reads. *Genome Biol.*, 12(6):R55, Jun 2011. 1.1.1

[90] Enrico Siragusa, David Weese, and Knut Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Research*, 41(7):e78, 2013. doi: 10.1093/nar/gkt005. URL `+http://dx.doi.org/10.1093/nar/gkt005`. 1.1.1

[91] Andrew D. Smith, Zhenyu Xuan, and Michael Q. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9(1):128, Feb 2008. ISSN 1471-2105. doi: 10.1186/1471-2105-9-128. URL `https://doi.org/10.1186/1471-2105-9-128`. 1.1.1

[92] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–195, 1981. 1.1, 1.3, 2.4, 5.3.1

[93] Martin Sosic and Mile Sikic. Edlib: A c/c++ library for fast, exact sequence alignment using edit distance. *bioRxiv*, 2016. doi: 10.1101/070649. URL `https://www.biorxiv.org/content/early/2016/08/23/070649`. 1.3

[94] Adam Szalkowski, Christian Ledergerber, Philipp Krahenbuhl, and Christophe Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.e. and x86/SSE2. *BMC Research Notes*, 1(1):107+, 2008. ISSN 1756-0500. doi: 10.1186/1756-0500-1-107. URL `http://dx.doi.org/10.1186/1756-0500-1-107`. 1.3, 2.1, 2.2, 2.4

[95] Todd J Treangen and Steven L Salzberg. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nat Rev Genet*, 13(1):36–46, 2012. doi: 10.1038/nrg3117. URL `http://dx.doi.org/10.1038/nrg3117`. 1

[96] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 1985. 1.3

[97] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, January 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90143-4. URL `http://dx.doi.org/10.1016/0304-3975(92)90143-4`. 1

[98] Mario Ventura, Claudia R Catacchio, Can Alkan, Tomas Marques-Bonet, Saba Sajjadian, Tina A Graves, Fereydoun Hormozdiari, Arcadi Navarro, Maika Malig, Carl Baker, Choli Lee, Emily H Turner, Lin Chen, Jeffrey M Kidd, Nicoletta Archidiacono, Jay Shendure, Richard K Wilson, and Evan E Eichler. Gorilla genome structural variation reveals evolutionary parallelisms with chimpanzee. *Genome Res*, 21:1640–1649, 2011. 1

[99] David Weese, Anne-Katrin Emde, Tobias Rausch, Andreas Doring, and Knut Reinert. RazerS fast read mapping with sensitivity control. *Genome Research*, 19:1646–1654, 2009. 1.1.1

[100] David Weese, Manuel Holtgrewe, and Knut Reinert. RazerS 3: Faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599, 2012. URL `http://dblp.uni-trier.de/db/journals/bioinformatics/bioinformatics28.html#WeeseHR12`. 1.1.1, 2.3.2, 4.1

[101] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu. Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics*, 31(10):1553–1560, May 2015. 1.4.2

[102] 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. ISSN 1471-2164. doi: 10.1186/1471-2164-14-S1-S13. URL `http://www.`

biomedcentral.com/1471-2164/14/S1/S13. 1.2, 1.3, 2.1, 2.2, 2, 2.4, 4.2, 4.4

[103] Hongyi Xin, Sunny Nahar, Richard Zhu, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Optimal seed solver: optimizing seed selection in read mapping. *Bioinformatics*, 32(11):1632–1642, 2016. doi: 10.1093/bioinformatics/btv670. URL +http://dx.doi.org/10.1093/bioinformatics/btv670. 1.4.2

[104] Hongyi Xin, Jeremie Kim, Sunny Nahar, Can Alkan, and Onur Mutlu. Leap: A generalization of the landau-vishkin algorithm with custom gap penalties. *bioRxiv*, 2017. doi: 10.1101/133157. URL https://www.biorxiv.org/content/early/2017/05/07/133157. 1.4.2

[105] Mengyao Zhao, Wan-Ping Lee, Erik P. Garrison, and Gabor T. Marth. Ssw library: An simd smith-waterman c/c++ library for use in genomic applications. *PLOS ONE*, 8(12), 12 2013. doi: 10.1371/journal.pone.0082138. URL https://doi.org/10.1371/journal.pone.0082138. 1.3

[106] Grace X. Y. Zheng, Billy T. Lau, Michael Schnall-Levin, Mirna Jarosz, John M. Bell, Christopher M. Hindson, Sofia Kyriazopoulou-Panagiotopoulou, Donald A. Masquelier, Landon Merrill, Jessica M. Terry, Patrice A. Mudivarti, Paul W. Wyatt, Rajiv Bharadwaj, Anthony J. Makarewicz, Yuan Li, Phillip Belgrader, Andrew D. Price, Adam J. Lowe, Patrick Marks, Gerard M. Vurens, Paul Hardenbol, Luz Montesclaros, Melissa Luo, Lawrence Greenfield, Alexander Wong, David E. Birch, Steven W. Short, Keith P. Bjornson, Pranav Patel, Erik S. Hopmans, Christina Wood, Sukhvinder Kaur, Glenn K. Lockwood, David Stafford, Joshua P. Delaney, Indira Wu, Heather S. Ordonez, Susan M. Grimes, Stephanie Greer, Josephine Y. Lee, Kamila Belhocine, Kristina M. Giorda, William H. Heaton, Geoffrey P. McDermott, Zachary W. Bent, Francesca Meschi, Nikola O. Kondov, Ryan Wilson, Jorge A. Bernate, Shawn Gauby, Alex Kindwall, Clara Bermejo, Adrian N. Fehr, Adrian Chan, Serge Saxonov, Kevin D. Ness, Benjamin J. Hindson, and Hanlee P. Ji. Haplotyping germline and cancer genomes with high-throughput linked-read sequencing. *Nature Biotechnology*, 34:303 EP –, Feb 2016. URL http://dx.doi.org/10.1038/nbt.3432. 6.2