Accelerating DNA Read Mapping and Alignment

Fall 2015

Sunny Nahar

April 18, 2016

Contents

1 Introduction **5**
- 1.1 What is DNA Sequencing? 5
- 1.2 Sequencing Platform 5
- 1.3 Complexity of Read Mapping 5
- 1.4 Handling Errors 6
- 1.5 Prefix Trie Method 6
- 1.6 Seed and Extend Method 6
- 1.7 Measuring Read Mapping 7

2 Research Question **8**
- 2.1 Uniform Selection 8
- 2.2 Cheap Kmer Selection (CKS) 8
- 2.3 Threshold 8
- 2.4 Hobbes 8
- 2.5 Optimal Seeds (OSS) 9
- 2.6 Goal 9

3 Repository structure **10**
- 3.1 Link 10
- 3.2 Directory 10
- 3.3 database/inc 10
  - 3.3.1 debug.h 10
  - 3.3.2 functionTimer.h 10
  - 3.3.3 logging.h 10
- 3.4 database/core 10
- 3.5 database/seedselection 10
  - 3.5.1 database/seedselection/predictor 11
  - 3.5.2 database/seedselection/selector 11
  - 3.5.3 database/seedselection/test 11
  - 3.5.4 database/seedselection/util 11
- 3.6 database/util 11

4 Updates to HashTree **12**
- 4.1 Overview 12
- 4.2 High Throughput Load and Store 12
  - 4.2.1 Design Flaw 12
  - 4.2.2 Working Solution 12
  - 4.2.3 Further Improvements 12
  - 4.2.4 Major Design Flaw 13
  - 4.2.5 Segmented Memory Allocator 13
- 4.3 OpenMP Integration 13
  - 4.3.1 Multithreaded Store 13
  - 4.3.2 Multithreaded Load 14
- 4.4 Future Optimizations 15

4.4.1 Centralized Memory Pool 15
4.4.2 Memory Usage 15
4.4.3 Multithreaded HashTree Access 15
4.4.4 Improving parallel performance 15

5 Bidirectional Frequency Predictor **16**
5.1 Motivation 16
5.2 Description 16
5.3 Implementation 16
5.4 Frequency Average Test 17
5.4.1 Mechanics 17
5.4.2 Selected Results 17
5.4.3 Analysis 19
5.5 Frequency Ratio Test 20
5.5.1 Mechanics 20
5.5.2 Selected Results 20
5.5.3 Analysis 22
5.6 Frequency Comparative Test 23
5.6.1 Mechanics 23
5.6.2 Selected Results 23
5.6.3 Analysis 23

6 Seed Selectors **25**
6.1 Baseline Methods 25
6.2 Uniform Seed Selector 25
6.2.1 Mechanics 25
6.2.2 Results 25
6.2.3 Analysis 25
6.3 Threshold Seed Selector 25
6.3.1 Mechanics 25
6.3.2 Results 26
6.3.3 Analysis 26
6.4 Hobbes Seed Selector 26
6.4.1 Mechanics 26
6.4.2 Results 27
6.4.3 Analysis 27
6.5 Bidirectional + Hobbes Seed Selector 27
6.5.1 Bidirectional Predictive Seed Extension 27
6.5.2 Mechanics 28
6.5.3 Results 28
6.5.4 Analysis 28
6.6 Bidirectional + Hobbes (Centralized prediction) 28
6.6.1 Centralized Prediction 28
6.6.2 Mechanics 29
6.6.3 Results 29

6.6.4 Analysis 29
6.7 Bidirectional + Uniform (Centralized prediction) 30
6.7.1 Uniform Centralized Prediction 30
6.7.2 Mechanics 30
6.7.3 Results 30
6.7.4 Analysis 30

7 Conclusions **32**

8 Future Work **33**
8.1 Iterative Prediction 33
8.2 Optimizing for Comparative Prediction 33
8.3 Extension Patterns 33
8.4 Randomized Division 33
8.5 Probabilistic Models 33
8.6 Machine Learning 34

9 Appendix A **36**
9.1 Results of Frequency Average Test 36
9.2 Results of Frequency Ratio Test 42
9.3 Results of Frequency Comparison Test 48

1 Introduction

1.1 What is DNA Sequencing?

In the better part of the last decade, DNA sequencing has arisen as an integral tool in the study of biology and medicine. DNA sequencing allows researchers to determine the exact sequence of nucleotides (base-pairs), *Adenine, Guanine, Thymine*, and *Cytosine*, in a strand of DNA and subsequently determine the sequence of any structure from a small gene to an entire genome. These sequences have a myriad of applications in numerous fields: discovering evolutionary patterns among species, tracing human evolution, identifying genetic components to determine the causes of diseases and construct remedying medicine, and identifying primary characteristics of individuals in forensic studies [1, 2, 3, 4]. This has led to a huge demand for fast and efficient genome sequencing algorithms. Since the technology is becoming a ubiquitous research tool among researchers, universities, and corporations alike, sequencing solutions which do not rely on excessively expensive or custom infrastructure are needed. There is a strong necessity for the ability to sequence DNA on commodity machines and not in days or weeks, but a few hours.

1.2 Sequencing Platform

The advent of largely parallel and high-throughput sequencing (HTS) platforms such as Illumina and SOLiD have revolutionized the field. These platforms sequence DNA by splitting long strands into small chunks of contiguous base-pairs, typically less than 100 base-pairs (bp), called **reads**. Reads are generated in the billions at a very high rate, and the objective is to reconstruct the original DNA sequence by aligning these short reads together [1, 2]. This is an unparalleled challenge on its own, since the computational complexity of aligning billions of reads is enormous. Sequencing the first human genome took more than a decade and over a billion dollars [3]. Fortunately, there are assembled genomes available today, and given the high rate of similarity between genomes, one can be used as a basis for constructing another. Therefore, reads are mapped to a reference genome to construct the overall DNA sequence.

1.3 Complexity of Read Mapping

However, this still poses a very computationally challenging problem. The reference genome is extremely large: the size of the human genome is **3.2 billion** base-pairs long. Reads are short and occur very frequently in the genome, so there are a vast number of possibilities where a single read could map to. In addition, all DNA sequences are not same: different organisms have different DNA structure, and within the same organism, like humans, there can be variations in the DNA due to mutations. This is compounded by possible errors introduced by the sequencing process: insertions, deletions, and substitutions in the read [3]. Therefore, the process of mapping reads to the reference genome is computationally hard since the mapping process needs to account for DNA variation and sequencing errors and allow a certain number of mismatches. The complexity increases with the number of mapping locations and error rate.

Given that there are a multitude of possible mapping locations for a given read, there are two different types of mapping algorithms depending on how many mappings are required. In the first case, a mapper may only need to find some of possible mappings for a read. This is the case in single nucleotide polymorphism (SNP) studies [3], where a few mappings are sufficient. In the other case, all possible mappings for a read may be required. For example in RNA-seq applications, finding all mapping positions is necessary for quantifying the level of gene expression, and in ChIP-seq applications, finding all mapping positions is necessary for characterizing protein binding sites. Mappers are often optimized to handle one of the two approaches [3].

1.4 Handling Errors

There are also multiple standards used for handling read mapping with variation and errors in DNA. Typically, the number of errors in a read is approximately 5% of its length [4]. Therefore there needs to be efficient methods to verify whether a read matches a substring of the reference genome within a specified number of mismatches. Candidate mapping locations for a read are valid only if they pass a similarity metric measuring the number of errors. The first commonly used metric for computing similarity between two strings is **Hamming distance**, the number of differences between one string compared to another. This works well with substitutions, but is not able to capture insertions and deletions effectively. The second metric is **edit-distance**, the smallest number of insertions, deletions, and substitutions needed to change one string to another. This metric is able to capture the full scope of errors in a read. However, this is at the cost of speed, as edit distance calculations are far more complex and expensive than Hamming distance calculations. Algorithms commonly used to compute edit-distance include Smith-Waterman and Needleman-Wunsch [3, 4].

1.5 Prefix Trie Method

There are two primary methods currently used to achieve efficient read mapping: using **suffix arrays/prefix tries** or using **gram-based/seed-and-extend** methods with **hash tables** [3, 1, 2]. Suffix array methods create an efficiently searchable tree data structure consisting of suffixes of the target genome. The tree stores suffixes by encoding characters as edges. A suffix is the concatenation of characters on the path from a leaf to the root. Each node in the tree stores all locations in the reference where that suffix appears. Therefore, searching for mapping locations of a read is equivalent to a search in a tree. In addition, the tree stores information obtained from applying the Burrows-Wheeler Transform and FM-indexing to the reference, allowing the tree to be used as a suffix array. This allows searching the tree with a significantly lower memory footprint. The industrial standard of mappers using this technique include BWA, Bowtie, and SOAP2 [2, 3].

1.6 Seed and Extend Method

In addition to suffix arrays, the other most popular method used is the seed-and-extend / gram-based method. When a mapper is matching a read to the target genome, it must account for errors in the read. Due to the possibility of errors, it is unlikely that the read will match somewhere in the reference exactly. However, it is extremely likely for a set of substrings of a read to match the genome exactly. These substrings are called **grams** or **seeds**. The mapper chooses a certain number of seeds from a read and searches the genome exactly for these seeds. Once the locations are determined, the mapper tries to determine whether the original read fits at a particular location by extending the seeds into the original read and checking whether it meets the error threshold. To make the algorithms efficient, hash tables are often used to store all possible seeds and their mapping locations in the reference genome. Seeds are often short, typically between $10 - 13$ base-pairs in length. Mappers which use this method include MAQ, mrFAST/mrsFAST, SHRiMP, Hobbes, and RazerS [3].

1.7 Measuring Read Mapping

To compare the effective of different algorithms, DNA read mapping methods are generally compared using three metrics. The first is **speed**: how long a mapper takes to map and align a set of reads to the target genome. The second is **sensitivity**: the percentage of reads for which the mapper can find at least one mapping. The last is **comprehensiveness**: the number of correct locations the mapper finds for a given read [3]. Typically, suffix array based mappers are very fast, but have poor sensitivity and comprehensiveness. They are efficient in finding a few mappings as opposed to all possible mappings. Seed-and-extend based methods are typically much slower, but have high sensitivity and comprehensive. They work better at finding all possible mappings of a read.

2 Research Question

In this research, we focus on improving the complexity and runtime of seed-and-extend based mappers. As discussed in section 1.6, seed-and-extend mappers align reads by splitting reads into smaller seeds. These seeds are searched in the reference and used to align the reads. The complexity of the alignment is in correlation with the how often the seeds occur in the reference, or the **frequency of the seeds**.

If seeds occur more frequently, a larger number of locations need to checked for potential mapping sites of the read, which increases complexity. Therefore it is imperative that we choose seeds which have the **lowest sum of frequencies** to minimize the number of mapping locations, and hence computation.

Therefore the goal of this research is to develop state-of-the-art seed selection schemes which are able to select seeds from a read in a memory efficient and cache efficient manner while minimizing the sum of the frequency of the seeds. These new heuristics and algorithms will be compared with existing seed selection schemes found in seed-and-extend mappers such as Hobbes and FastHash, among others.

We now review current seed selection methods.

2.1 Uniform Selection

This is the simplest of selection schemes. The read is split equally into seeds. Therefore this requires no overhead or additional data. Since it does not use any information about seed frequencies or the HashTree it suffers in practice and does not perform well.

2.2 Cheap Kmer Selection (CKS)

This method is used in the FastHASH mapper. The read in split into N contiguous chunks, and the frequency of each chunk is measured through the HashTree. Then the least frequent set of k chunks is selected for k seeds. The seeds can be expanded to fill up the read, since increasing the size of the read does not increase the frequency. By considering k out of the N contiguous chunks, the method is fast, but does not provide much frequency reduction compared to Uniform selection.

2.3 Threshold

This is another common selection method. Given a start position, the length of the seed continually increases until the frequency of the seed, obtainable through the HashTree class, reaches below a given threshold. Seeds are selected contiguously. This method suffers from finding suitable values for this threshold to work well over all reads.

2.4 Hobbes

Hobbes expands on the idea of CKS. It considers all n-length contiguous substrings of the read, and uses dynamic programming to find the optimal set of non-overlapping seeds which are each of length n. This makes it slower as it needs to look up more seed frequencies in the reference and requires a quadratic time dynamic programming calculation. Hobbes performs better than all the previous algorithms in finding a set of seeds with low overall frequency. But it is still limited as it considers only fixed length seeds.

2.5 Optimal Seeds (OSS)

The Optimal Seeds Solver expands on Hobbes by considering variable length seeds. It finds the theoretically optimal division into seeds with the lowest possible frequency. But this is at the price of a high number of HashTree lookups and increased runtime complexity. OSS runs a dynamic programming method to find the optimal divisions as well.

2.6 Goal

Therefore, the goal is to create a hybrid seed selection scheme which performs better than Hobbes, CKS, and Threshold but much faster than OSS, as speed is the limiting factoring for OSS.

3 Repository structure

3.1 Link

The repository for this project is hosted on GitHub at `https://github.com/xhongyi/buildingblock`.

3.2 Directory

All the necessary code in located in the `database` folder.

3.3 database/inc

This includes common utility header files used in most programs.

3.3.1 debug.h

This contains assert constructs for style and correctness when debugging. It is enabled through the `-DDEBUGGING` compile time flag.

3.3.2 functionTimer.h

This header defines useful functions for measuring time for measuring execution runtime. There is a function for measuring the current wall time and a function for measuring the current clock time. Clock time is aggregated over all threads, so multithreaded execution time should be measured through wall time for physical time and through clock time for compute time.

3.3.3 logging.h

Defines logging constructs to be used for execution and debugging. Logging is color coded onto the console for 4 different modes, `INFO`, `WARNING`, `ERROR`, and `DEBUG`. The debug logging is enabled through the `-DDEBUGGING` compile time flag. Enabling the debug logging also adds the file name and function from which the logging is called.

3.4 database/core

The folder contains the code for constructing the HashTree representation of the genome. The genome is stored by storing all 30-character long prefixes into a tree, i.e. a prefix trie. The first 10 characters are used to hash into a hashtable which then points to a tree on the remaining 20 characters.

3.5 database/seedselection

This folder, which is the main contribution and substance of this research, contains four subfolders, `predictor`, `selector`, `test`, and `util`.

3.5.1 database/seedselection/predictor

This folder contains the implementation of the BiDirectional Frequency Predictor, which is discussed in section 5.

3.5.2 database/seedselection/selector

This folder contains implementations of numerous seed selectors. Further discussion in section 6.

3.5.3 database/seedselection/test

This folder contains tests for the frequency predictor and the seed selectors.

3.5.4 database/seedselection/util

Contains utility classes for generating random reads and seeds and generating seeds based on actual read files such as `ERR240726_1.filt.fastq`.

3.6 database/util

Contains the implementation of a basic memory pool. Discussed in section 4.1.

4 Updates to HashTree

4.1 Overview

The HashTree is a class which stores the genome as a prefix trie. It is stored as a hashtable of trees, where the first 10 characters of a prefix index into the hashtable, and the remaining characters index into the corresponding tree. It stores the frequency of each prefix in addition to other metadata.

The main operation on a HashTree is querying the frequency of read. This operation is linear in the length of the read, as it needs to traverse the tree to the location of the read at which it can return the frequency.

4.2 High Throughput Load and Store

A major bottleneck during testing was loading and storing the HashTree representation. Generating the entire tree on the genome was infeasible as it required 4 hours, and this would make testing an arduous process. A export binary format had been created before to speed up this process by writing the HashTree to a file, and reading the HashTree from file during testing. This did speed it up by approximately 2x, to where storing and loading took 2hrs, but it remained a major bottleneck. The binary form of the test genome, `human_g1k_v1.fasta`, was extremely large at 78GB.

4.2.1 Design Flaw

One of the design flaws in the implementation of the load and store functions were small writes to disk interleaved with computation which were creating unnecessary overhead in the write process in the number of function calls.

4.2.2 Working Solution

This was solved by writing the output to the store and load to an internal buffered, which could then be used to read and write directly to the file. This reduced the number of read / write calls to just one.

4.2.3 Further Improvements

Although this provides a performance improvement, this is at the cost of memory usage. Since the entire file is stored in memory, an additional 78GB of memory is required to facilitate that process, whereas the initial method required no overhead. This can likely be reduced by making the buffer the size of the file write buffer, but it depends on the implementation of the file write. Currently, there is not much issue with the memory overhead as the system can support it along with the HashTree.

4.2.4 Major Design Flaw

The previous improvement lowered the execution time of storing the tree, but did not affect loading. The crucial part of testing is loading the HashTree, as storing the tree has to be done only once, after generating the tree. A crucial design flaw in the load was the continual calls to `new`. The load function constructs the tree, so nodes have to be created and links between nodes. Consistent calls to `new`, on the order of 10 billion, severely slows down the execution. The reason for this is that `new` is thread safe, so it implicitly requires a lock which is held and then released each time `new` is called.

4.2.5 Segmented Memory Allocator

The solution to the continual use of `new` is creating a type of memory pool, which we call a segmented memory allocator. A memory pool is a small allocator which the program can use instead of `new`. Internally, it requests a large chunk of memory using `new`, and slowly uses up that memory when allocation calls to memory allocator are received.

In this application, the program only needs to allocate and not free memory. This is because the construction of the HashTree only needs to allocate memory for each node but not free. Hence, this simplifies the design of the allocator. As there are no calls to free, memory is not segmented, and hence there are no free blocks to manage. Therefore, the memory allocator can simply keep track of a pointer which points to the start of the unused sub-block of memory, and during and allocation call, update the pointer. When the block of memory runs out, it simply calls `new` to allocate another chunk.

Therefore calls to new are much faster. This lead to a major improvement in execution time of HashTree load, reducing the time from 2 hours to 7 minutes. In addition freeing the memory after using the HashTree becomes much faster since instead of the destructor calling free's on billions of pointers, it only calls free on the pointers to each chunk.

The implementation of the allocator is located in `database/util`. The API interface defines a constructor which takes in a chunk size, and an `alloc` function which. Each time the allocator runs out of memory, a new internal block is allocated.

4.3 OpenMP Integration

To further speed up the process, multithreading functionality was added to load and store.

4.3.1 Multithreaded Store

Parallelizing storing the HashTree as one file is infeasible since the order of bytes in the storage binary matters. Although this could potentially be done by using multiple file descriptors each with its own write location in the file, the number of bytes written by each thread needs to be known beforehand, and this is difficult to obtain correctly.

Therefore the natural extension is to split the HashTree across multiple files, with each thread handling a set of files. This complements well with multithreaded load, where each thread loads from its set of files. As each tree in the hashtable of the HashTree is disjoint, it is natural to split the hashtable among the threads.

Converting each tree into a binary can be done in parallel, since they are independent. However the writes to the file must be such that the entire file is written to before another, to prevent disk thrashing.

The number of threads which seemed to provide the most performance was between 8-12. After that, increasing the number of threads results in more threads sleeping, waiting on disk, and finishing the parallel binary compute task.

This portion can be further optimized by investigating the relationship between number of threads and number of splits of the HashTree.

4.3.2 Multithreaded Load

The construction of the multithreaded load was more complicated. Each file had to be read completely by a thread and into a buffer to prevent disk thrashing. Once the file was read into a buffer, the thread could construct that portion of the HashTree, and link it with the others. This worked well in conjunction with store as the HashTree was stored in separate parts, so there was no need for synchronization between files.

Once again, the use of the segmented memory allocator was crucial for performance. Since the memory architecture of the system was nonuniform, a local memory allocator had to be created for each thread, so that cache coherence between processors would not be a limiting factor. However being locally allocated, the memory has to managed as the allocator leaves scope after the thread exits.

A centralized management of allocators needs to be created to manage this memory so that the proper blocks can be freed after use.

Once again, the optimal performance was using only 4-8 threads, out of the 80 possible threads capable of running on the machine. This could be improved through parameter tuning, but it seems sufficient as of now.

4.4 Future Optimizations

4.4.1 Centralized Memory Pool

As discussed above, a centralized system for managing instances of the segmented allocator is needed.

4.4.2 Memory Usage

Memory usage of the HashTree can be reduced by optimizing the size of the nodes in the tree. Some have potentially redundant information which can be removed. In addition, the current structure uses a class based construction of nodes, as opposed to using structs. Using structs can likely cut the space by 50 - 70%, however how this will affect the API and readability of the code is yet to be determined.

4.4.3 Multithreaded HashTree Access

A possible source of performance improvement is through parallel access of the tree. However there is much to be taken in consideration here, as this would require memory to store locks. Given that the tree is segmented across multiple memory modules, this could unnecessary cause slowdown due to cache coherence and end up not providing much performance improvements.

4.4.4 Improving parallel performance

There are many places where parameter tuning can be done to optimize for performance. In addition, there are further design issues to remedy between shared data structures across threads. False sharing is likely to be occurring in certain places and needs to be looked at.

5 Bidirectional Frequency Predictor

5.1 Motivation

The crux of the development of this research is through the concept of the predicting frequency. We want to be able to minimize queries to the HashTree for frequency of seeds and instead create a data structure which essentially provides a dimensionality reduction over the space of seeds.

We can view seeds (up to length 30) as a 31 dimensional vector in a space $\mathcal{S}$ of size $30 * 4^{30}$, where the first dimensional specifies length, and the remaining specify the seed. We have a function $f : \mathcal{S} \mapsto \mathbb{N}$ which maps a vector and its corresponding seed to the frequency of the seed in the reference. Clearly storing every possible seed frequency requires too much space. Instead, we want to store less information such that we can get an estimate of the frequency.

5.2 Description

Therefore, we create a data structure called a bidirectional frequency predictor. This is a table which given a base seed, left extension, and right extension, gives an estimate of the frequency of the seed if the base was extended to the left by a certain number of characters and to the right by a certain number of characters. Therefore if the base seed has length 10, we can support up to a left and right extension of 20, given the max length as 30, so the size of the table is $4^{10} * 20 * 20 * 4 = 1.6$GB, a much more manageable size. For a table with base seed length 12 and left and right extensions length of 18, the size is 20GB.

Note that it is very difficult for the prediction to be accurate. This is due to many factors. The prediction for a given base seed and left and right extension is made by averaging out all paths in the HashTree which are the desired length and have the base seed in the appropriate place. As the frequencies of seeds are heavily variable, ranging in 4 - 5 orders of magnitude even at the same depth, averages are skewed. Therefore the expectation here is that if the prediction is within a factor of 10x, maybe even 20-50x, it may be useful.

5.3 Implementation

The implementation is located in `database/seedselection/predictor`. The predictor is created by traversing the HashTree and updating the respective part of the table during the traversal. The frequency of a node is weighted by itself during the average calculation. This is because the frequency represents the number of positions in the reference which have that string occurring, and a highly frequent seed is more likely to appear, so it gets higher weight. Once the predictor is created, it can be stored to a file so that it does not have to be recreated each time.

5.4 Frequency Average Test

5.4.1 Mechanics

The first test used the `ERR240726_1.filt.fastq` read set composing of 4026809 reads of length 100. To get k-length seeds, the first k letters of each read were taken. The seeds were divided into 12 sets based on the actual frequency: [0], [1-5], [6 - 10], [11 - 50], [51 - 100], [101 - 500], [501 - 1000], [1001 - 5000], [5001 - 10000], [10001 - 100000], [100001 - 1000000], [1000001 - 1000000000]. For each set, the number of seeds in each set were counted, the average exact frequency of the seeds was calculated. In addition, for each possible sets of left and right extensions, the average predicted frequency was calculated. This was predicted through a bidirectional frequency predictor with base seedlength of 10. Therefore, for example, $(3, 4)$ in the tables below indicates that the prediction of a length $10 + 3 + 4 = 17$ seed was calculated such that positions $3 - 12$ were used for the base seed and searched in the table with left extension 3, right extension 4.

5.4.2 Selected Results

Freqs:	0	5	10	50	100	500
Count:	276	7852	15058	214606	155766	2045663
Avg Freq:	0.00	3.40	8.24	28.44	76.68	276.09
(0, 2):	15.93	36.43	60.24	140.76	297.01	564.25
(1, 1):	17.46	39.70	62.21	143.90	284.38	558.51
(2, 0):	16.09	30.70	57.89	145.43	290.71	566.10

Table 1: Seed length 12

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	688840	555620	108938	179399	52030	2761
Avg Freq:	693.54	2074.50	7095.21	36617.05	196559.76	2059168.57
(0, 2):	1234.97	3751.62	9835.89	39765.91	189361.26	1467818.82
(1, 1):	1207.22	3685.46	10587.19	38192.25	201147.23	1478942.37
(2, 0):	1242.16	3843.22	10190.89	42736.41	194746.92	1470746.04

Table 2: Seed length 12

Freqs:	0	5	10	50	100	500
Count:	119893	1296668	782215	1012545	169448	228469
Avg Freq:	0.00	2.90	7.71	21.54	70.75	231.81
(0, 5):	94.76	142.80	196.01	378.19	1050.82	2593.59
(1, 4):	88.25	137.32	196.19	365.75	1028.75	2575.29
(2, 3):	92.72	139.16	192.85	357.71	1051.37	2620.70
(3, 2):	93.05	139.42	187.77	364.28	1062.28	2669.95
(4, 1):	97.40	139.39	190.59	373.66	1074.14	2725.39
(5, 0):	92.20	143.02	198.27	391.13	1134.53	2930.94

Table 3: Seed length 15

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	67449	147865	68003	110002	22731	1521
Avg Freq:	708.27	2513.11	7125.24	38015.44	173055.18	1191151.41
(0, 5):	6500.26	9933.95	11493.09	33278.47	123185.21	499685.63
(1, 4):	5368.47	9709.77	12161.65	33446.83	131134.11	509271.39
(2, 3):	5750.41	9122.51	12469.85	34774.53	134129.87	514628.16
(3, 2):	5787.67	9534.10	12678.14	34910.12	137436.07	518084.84
(4, 1):	5965.05	10939.14	13353.92	33978.20	142319.18	519248.13
(5, 0):	7576.56	11455.24	13135.02	34583.92	138267.35	513356.68

Table 4: Seed length 15

Freqs:	0	5	10	50	100	500
Count:	708601	2509784	107476	190386	67327	137416
Avg Freq:	0.00	1.35	7.64	24.33	72.14	239.50
(0, 8):	146.54	185.98	905.40	1727.28	2540.82	3525.44
(1, 7):	140.95	181.57	826.45	1576.29	2497.08	3581.61
(2, 6):	141.73	181.21	820.99	1522.09	2392.90	3517.53
(3, 5):	143.74	177.98	836.01	1550.87	2286.16	3400.05
(4, 4):	139.46	177.38	815.23	1568.57	2444.74	3401.16
(5, 3):	143.27	178.04	835.76	1571.80	2377.97	3628.36
(6, 2):	143.49	183.79	868.36	1620.57	2579.40	4011.43
(7, 1):	136.92	192.24	930.45	1831.74	2920.05	4070.44
(8, 0):	136.26	204.77	1092.63	2148.54	3045.26	4079.51

Table 5: Seed length 18

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	45861	122776	49277	77158	10747	0
Avg Freq:	723.75	2557.84	6943.46	34782.67	211144.09	0.00
(0, 8):	5991.70	6760.99	8559.45	22857.12	99409.11	0.00
(1, 7):	6016.09	6928.29	8577.15	23204.92	102778.90	0.00
(2, 6):	6210.54	7092.48	8661.80	23809.86	104476.53	0.00
(3, 5):	6208.34	7406.93	8921.84	23769.40	107979.85	0.00
(4, 4):	6307.44	7736.48	9203.70	23643.40	111953.00	0.00
(5, 3):	6643.53	7744.20	9172.26	24029.35	111473.34	0.00
(6, 2):	6878.12	7708.36	9333.54	24259.54	110395.75	0.00
(7, 1):	6856.99	7757.74	9359.86	24430.85	108938.98	0.00
(8, 0):	7031.62	7812.77	9304.02	24638.03	107708.63	0.00

Table 6: Seed length 18

5.4.3 Analysis

For a particular seed length, we can see that the prediction is best for seeds with higher frequencies and gets progressively worse as the seed frequency decreases. This is due to the average being skewed toward large frequencies. The weights of the seeds with higher frequency is larger than the weights with lower frequency for the same total length, so seeds with lower frequencies will be overestimated. However the assumption is that since these seeds have low frequencies, we do not expect to see too many of them, so the prediction will be mostly accurate. Using a predictor with base length 10, we see that for seeds with length at most 15 with frequency at least 100, the prediction is at most 13x away. For seeds with length at most 30 with frequency at least 100, the prediction is at most 30x away.

5.5 Frequency Ratio Test

5.5.1 Mechanics

In this test, we compute the ratio of the predicted frequency to the exact frequency for seeds in the read set `ERR240726_1.filt.fastq`. The average ratio is displayed over different seed lengths and frequency classes. The classes are the same as the previous test.

5.5.2 Selected Results

Freqs:	0	1	5	10	50	100
Count:	276	889	6963	15058	214606	155766
Avg Freq:	0.00	1.00	3.70	8.24	28.44	76.68
(0, 2):	0.00	16.04	10.40	7.33	5.07	4.00
(1, 1):	0.00	24.43	11.36	7.63	5.20	3.81
(2, 0):	0.00	18.23	9.17	7.12	5.17	3.95

Table 7: Seed length 12

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	2045663	688840	555620	108938	179399	54791
Avg Freq:	276.09	693.54	2074.50	7095.21	36617.05	290419.39
(0, 2):	2.13	1.78	1.81	1.40	1.16	0.97
(1, 1):	2.13	1.74	1.78	1.50	1.20	1.04
(2, 0):	2.13	1.79	1.84	1.45	1.26	1.00

Table 8: Seed length 12

Freqs:	0	1	5	10	50	100
Count:	119893	287608	1009060	782215	1012545	169448
Avg Freq:	0.00	1.00	3.45	7.71	21.54	70.75
(0, 5):	0.00	104.66	48.59	25.94	18.37	14.84
(1, 4):	0.00	104.44	46.24	25.88	17.93	14.48
(2, 3):	0.00	107.86	46.88	25.49	17.33	14.86
(3, 2):	0.00	106.20	47.02	24.87	17.67	15.08
(4, 1):	0.00	106.30	46.82	25.13	18.16	15.23
(5, 0):	0.00	105.99	48.37	26.19	18.84	16.19

Table 9: Seed length 15

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	228469	67449	147865	68003	110002	24252
Avg Freq:	231.81	708.27	2513.11	7125.24	38015.44	236906.59
(0, 5):	11.64	9.26	4.70	1.64	1.01	0.70
(1, 4):	11.91	7.65	4.48	1.75	1.01	0.75
(2, 3):	12.03	8.19	4.31	1.78	1.06	0.77
(3, 2):	12.20	8.27	4.44	1.81	1.09	0.79
(4, 1):	12.54	8.52	5.05	1.92	1.07	0.82
(5, 0):	13.13	10.86	5.42	1.87	1.09	0.80

Table 10: Seed length 15

Freqs:	0	1	5	10	50	100
Count:	708601	1996228	513556	107476	190386	67327
Avg Freq:	0.00	1.00	2.70	7.64	24.33	72.14
(0, 8):	0.00	142.18	132.59	119.70	79.80	36.34
(1, 7):	0.00	140.53	126.94	109.25	70.90	35.57
(2, 6):	0.00	140.14	126.18	108.76	69.43	34.02
(3, 5):	0.00	137.29	123.64	110.86	70.14	32.67
(4, 4):	0.00	137.10	122.66	108.22	71.06	34.86
(5, 3):	0.00	137.23	123.01	110.78	71.11	33.97
(6, 2):	0.00	141.04	129.65	115.66	74.10	36.66
(7, 1):	0.00	146.91	136.27	124.14	82.55	41.80
(8, 0):	0.00	155.12	146.70	144.72	99.73	43.69

Table 11: Seed length 18

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	137416	45861	122776	49277	77158	10747
Avg Freq:	239.50	723.75	2557.84	6943.46	34782.67	211144.09
(0, 8):	16.79	8.51	3.22	1.25	0.72	0.54
(1, 7):	16.94	8.53	3.30	1.25	0.74	0.56
(2, 6):	16.58	8.80	3.38	1.27	0.76	0.57
(3, 5):	15.81	8.79	3.50	1.31	0.76	0.60
(4, 4):	15.88	8.93	3.68	1.34	0.75	0.63
(5, 3):	16.88	9.46	3.68	1.35	0.76	0.62
(6, 2):	19.05	9.80	3.69	1.37	0.77	0.62
(7, 1):	19.50	9.77	3.73	1.38	0.78	0.61
(8, 0):	19.69	10.01	3.77	1.37	0.78	0.60

Table 12: Seed length 18

5.5.3 Analysis

In this test, we have a clearer picture of by what factor the predicted frequencies differ from the actual frequencies. As this is just a reinterpretation of the previous data, the trends are the same. The prediction gets progressively worse as the frequency in a group of seeds of length m decreases. For seeds with length at most 15, we can predict seeds with large frequency (at least 10000) within a factor of 2. Expanding this to seeds with frequency at least 1000, we can predict this within a factor of 10. For seeds with length at most 20, the predictive ratios are slightly better: within a factor of 1.9 with frequency at least 10000, within a factor of 7.7 for seeds with frequency at least 1000, and within a factor of 30 for seeds with frequency at least 100. For seeds with length at most 25, the ratio is within 4.3 for seeds with frequency at least 1000. So the predictor works fairly well for high frequency seeds.

5.6 Frequency Comparative Test

5.6.1 Mechanics

This test is a comparative test between seeds. The test takes roughly 4 million pairs of seeds and calculates the percentage of the time the predictor predicts the correct sign, i.e. the percentage of time if the actual frequency of seed 1 is larger than seed 2 the predicted frequency of seed 1 is larger than seed 2, and vice versa.

5.6.2 Selected Results

(left, right)	% of correct comparisons
(0, 2):	0.838487
(1, 1):	0.834117
(2, 0):	0.838308

Table 13: Seed length 12

(left, right)	% of correct comparisons
(0, 5):	0.734238
(1, 4):	0.733823
(2, 3):	0.734016
(3, 2):	0.734292
(4, 1):	0.733973
(5, 0):	0.735255

Table 14: Seed length 15

(left, right)	% of correct comparisons
(0, 8):	0.644817
(1, 7):	0.648471
(2, 6):	0.647196
(3, 5):	0.647320
(4, 4):	0.649201
(5, 3):	0.649550
(6, 2):	0.649888
(7, 1):	0.649108
(8, 0):	0.649469

Table 15: Seed length 18

5.6.3 Analysis

As expected, the accuracy decreases with increasing seed size. This is due to, for larger length seeds, an average taken over a larger and more variable distribution, which will not be very accurate. The accuracy was consistent over the extension combinations for each seed length, differing at most by 0.01%. The predictor was 88% accurate for 11-length seeds, at least 73% accurate for seeds with length at most 15, and at least 60% accurate for seeds with length at most 25%. The predictor's high percentage in comparative accuracy makes it a useful tool in deciding which seeds to pick.

6 Seed Selectors

6.1 Baseline Methods

Three seed selectors using existing methods were implemented to compare the performance of the predictor: Uniform, Threshold, and Hobbes. Both Uniform and Threshold are used as a starting point for the seed selectors using the predictor, so they also provide a direct comparison in the percentage reduction of seed frequency. The seed selectors were compared in both frequency sum reduction and runtime. Comparison of runtime is subjective since there are always performance improvements which can be made to optimize the seed selection algorithm. Therefore absolute comparison of runtime is only somewhat meaningful. Since the predictive algorithms use Hobbes and Uniform seed selection schemes as a baseline, they can be compared reliably to their nonpredictive variant.

6.2 Uniform Seed Selector

6.2.1 Mechanics

The Uniform Seed Selector simply chooses equal length seeds. It does not have any parameters. The Uniform Seed Selector was tested with the `ERR240726_1.filt.fastq` read set with approximate 4 million reads of size 100 and 5 seeds.

6.2.2 Results

Initialization (s)	Runtime (s)	Average sum of frequencies
0.00	0.83	5247.36

Table 16: Uniform Seed Selector results (5 seeds)

6.2.3 Analysis

The average sum of frequencies of the seeds in a read was 5247.36 when they were chosen equally.

6.3 Threshold Seed Selector

6.3.1 Mechanics

The Threshold Seed Selector increases the seed length of a seed until it falls below a certain threshold. It takes in as a parameter the threshold at which to stop extending the seed. The Threshold Seed Selector was tested with the `ERR240726_1.filt.fastq` read set with approximate 4 million reads of size 100. The threshold parameter was varied to within reasonable bounds.

6.3.2 Results

Threshold	Initialization (s)	Runtime (s)	Average sum of frequencies
100	0.00	32.75	7804.55
200	0.00	30.34	6738.40
300	0.00	28.49	6281.90
400	0.00	27.33	6038.36
500	0.00	26.71	5914.05
600	0.00	25.65	5881.85
700	0.00	25.15	5894.80
800	0.00	24.72	5945.74
900	0.00	24.29	6022.96
1000	0.00	23.92	6129.13

Table 17: Threshold Seed Selector results (5 seeds)

6.3.3 Analysis

In all cases, the Threshold seed selector performs worse the the Uniform seed selector. It reaches its optimum at the threshold of 600 and increases on both sides. This is due to the fact that if the threshold is too small, then too many base pairs will be used initially, forcing the last seeds to have the minimal possible length, which have extremely high frequency. If the threshold is too large, valuable base pairs are essentially unused and wasted on the end seeds which do not seed them. Clearly the threshold predictor is not a good choice, especially if it cannot perform better than the naive method. It is also much slower than the Uniform seed selector, by a factor of 30x.

6.4 Hobbes Seed Selector

6.4.1 Mechanics

The Hobbes Seed Selector uses a dynamic programming algorithm to figure out the optimal set of k seeds where all the seeds have a common fixed length. This is a parameter the Hobbes Seed Selector takes in. Since the seeds given by Hobbes do not always use up the entire read, they are extended uniformly to fill up the read. If there were extra base pairs available between two seeds, the seeds were extended equally to use the extra base pairs. The Hobbes Seed Selector was tested with the `ERR240726_1.filt.fastq` read set with approximate 4 million reads of size 100 and 5 seeds with varying values of the base seed length parameter.

6.4.2 Results

Hobbes seed length	Initialization (s)	Runtime (s)	Average sum of frequencies
10	0.00	161.98	1957.68
11	0.00	328.19	1470.05
12	0.00	383.13	1293.36
13	0.00	468.93	1232.43
14	0.00	524.73	1254.66
15	0.00	574.62	1350.23
16	0.00	638.47	1499.83
17	0.00	695.79	1759.31
18	0.00	699.05	2261.20
19	0.00	683.73	3302.05

Table 18: Hobbes Seed Selector results (5 seeds)

6.4.3 Analysis

The frequency reduction of Hobbes increases with Hobbes seed length until seed length 13, at which it reduces at a much higher rate until seed length 19. The runtime increases with the Hobbes seed length. It jumps in the beginning, double at length 10 to 11, but becomes linear after that. Hobbes performs much better than the Uniform and Threshold Seed Selectors. The fastest Hobbes (seed length 10) is about 2.5x better than Uniform and Threshold, although it is 7x slower than Threshold, and considerably slower than Uniform. The best Hobbes selector (seed length 13) performs 4.5x better than Uniform and Threshold, but it is 21x slower. Since the maximum Hobbes seed length is 20, given read length of 100 and 5 seeds, as the seed length becomes close to the maximum, it is likely that many seeds use extra base pairs which they do not need. Then other seeds remain too frequent, causing the increase in frequency we see above.

6.5 Bidirectional + Hobbes Seed Selector

6.5.1 Bidirectional Predictive Seed Extension

The first of the predictive seed selectors is the Bidirectional Hobbes Seed Selector. Using a parameter describing the length of each seed k, it calls Hobbes to find the optimal set of seeds of length k in the read. Then instead of performing uniform seed extension in Hobbes, we perform a predictive seed extension. Hence during extension, no calls to HashTree are made. Suppose there are d base pairs between seeds i and $i+1$. Then there are $d+1$ possible ways to split the base pairs among seeds. For each division (a, b) where $a + b = d + 1$, the frequency is estimated using the predictor, where the Hobbes seed is used as the base seed for the predictor. The right extension for seed i is a, and the left extension for seed $i + 1$ is b. The division with the lowest predicted sum of seeds is picked. This is processes linearly from left to right among the seeds, first optimizing the divider between seed 1 and 2, and then 2 and 3, and so forth.

Note: refer to section 5 for reference on predictor terminology.

6.5.2 Mechanics

The Bidirectional Hobbes Seed Selector was tested with the `ERR240726_1.filt.fastq` read set with approximate 4 million reads of size 100 and 5 seeds. Since the Hobbes seeds are used as anchors and cannot move and the predictor uses the Hobbes seeds as the base seeds, the predictor base length must be the same as the Hobbes length. Since we are only testing with the base length 10 predictor, we can only test with the Hobbes length 10 as of now. We test using the predictor and using the HashTree for exact frequencies in order to provide a measurement of the effectiveness of the predictor.

6.5.3 Results

Hobbes Length	Initialization (s)	Runtime (s)	Average sum of frequencies
10	3.64	234.80	1817.16

Table 19: Bidirectional Hobbes Seed Selector results (5 seeds) (Predictive frequencies)

Hobbes Length	Initialization (s)	Runtime (s)	Average sum of frequencies
10	0.00	319.20	1788.83

Table 20: Bidirectional Hobbes Seed Selector results (5 seeds) (Exact Frequencies)

6.5.4 Analysis

Compared to the Hobbes seeds selector with Hobbes length 10, this has a 7% reduction in frequency sum of the seeds. The increase in runtime is about 45%. Without data for other Hobbes lengths, it is hard to determine how effective this method is over Hobbes. It clearly outperforms Threshold and Uniform as well. The predictive version is only 2% away from the optimal division using exact frequencies, as shown above. This shows that the predictor performs fairly well in choosing seed division close to the optimal for this method. As the theoretically lowest frequency sum using this method with a Hobbes seed length of 10 is only 2% lower, it seems as if this does not provide much potential for frequency reduction.

6.6 Bidirectional + Hobbes (Centralized prediction)

6.6.1 Centralized Prediction

This seed selector is a variant of the previous one. We again use Hobbes to find the initial set of seeds of length k. When we perform seed extension with the remaining base pairs, we use a centralized prediction scheme: if we are looking at the extension of the Hobbes seed by a base pairs to the left and b base pairs to the right, the instead of using the Hobbes seed as the predictor base seed and a as the left extension and b as the right extension, we choose the middle k of the $a+k+b$ base pairs as the predictor base seed, with left extension $(a+b)/2$ and right extension $(a+b)/2$. There are more possible divisions in this method since the predictor base seed is not restricted to the Hobbes "anchor" seeds as before. Once again, updates to the divisions are performed linearly from left to right.

Note: refer to section 5 for reference on predictor terminology.

6.6.2 Mechanics

The Centralized Bidirectional Hobbes Seed Selector was tested with the `ERR240726_1.filt.fastq` read set with approximate 4 million reads of size 100 and 5 seeds. Hobbes seeds are no longer anchors in this method, we can easily test over different Hobbes seed lengths with the same predictor. The results here are shortened due to large amounts of computational time required for the higher Hobbes seed lengths.

6.6.3 Results

Hobbes seed length	Initialization (s)	Runtime (s)	Average sum of frequencies
10	3.60	305.14	1475.86
11	3.60	447.17	1396.98
12	3.60	524.43	1371.63
13	3.60	588.36	1369.19
14	3.60	644.77	1388.12
15	3.60	703.14	1425.18
16	3.60	735.22	1472.08

Table 21: Centralized Bidirectional Hobbes Seed Selector results (5 seeds) (Predictive Frequencies)

Hobbes seed length	Initialization (s)	Runtime (s)	Average sum of frequencies
10	0.00	859.56	1008.44
11	0.00	1024.30	940.11

Table 22: Centralized Bidirectional Hobbes Seed Selector results (5 seeds) (Exact Frequencies)

6.6.4 Analysis

Centralized Bidirectional Hobbes performs the best out all the predictive seed selectors. Using a Hobbes length of 10, its average frequency sum is 1476, 10% better than Centralized Bidirectional Uniform and 19% better than Bidirectional Hobbes. It is 25% better than traditional Hobbes. Its runtime is about 33% more than Bidirectional Hobbes and 50% more than Centralized Bidirectional Uniform. Although surprisingly, it does not do better than Hobbes for Hobbes length of 12 - 15. This is an interesting result and cause is still undetermined. The theoretically optimal frequency using this method for Hobbes length 10 is 1008.44, which is lower than all of Hobbes. If the predictor is able to be improved, this method would perform really well.

6.7 Bidirectional + Uniform (Centralized prediction)

6.7.1 Uniform Centralized Prediction

In this seed selector, we use the Uniform seed selector as the baseline, so seeds are split equally in the read. Then for each pair of adjacent seeds, we view the conjunction of the base pairs as one large chunk of base pairs, and try all possible divisions of this chunk into two seeds. The optimal would be picked by choosing the one with the lowest predicted sum of frequencies, using centralized prediction, as detailed in the previous seed selector in section 6.6.1.

For example with a read length of 100 and 5 seeds, each seed would initially start out with 20 base pairs. In the first iteration, the sequence of 40 base pairs of the first two seeds would be looked at. If the minimum seed length was 10, then the following divisions would be considered: $(10, 30), (11, 29), \ldots, (29, 11), (30, 10)$.

6.7.2 Mechanics

The selector was tested with the `ERR240726_1.filt.fastq` read set with approximate 4 million reads of size 100 and 5 seeds. Results were calculated using both the predicted frequency from the predictor and the exact frequency from the HashTree to compare effective of the predictor and runtime.

6.7.3 Results

Initialization (s)	Runtime (s)	Average sum of frequencies
3.53	205.66	1607.09

Table 23: Centralized Bidirectional Uniform Seed Selector results (5 seeds) (Predictive frequencies)

Initialization (s)	Runtime (s)	Average sum of frequencies
0.00	796.38	1359.63

Table 24: Centralized Bidirectional Uniform Seed Selector results (5 seeds) (Exact frequencies)

6.7.4 Analysis

The predictive seed selector performed much better than its Uniform counterpart, reducing the frequency sum by a factor of 3.3x. This was at the cost of a 200s increase in computation time. It performed better than the Bidirectional Hobbes seed selector, which is surprising, since the expectation was that Hobbes seeds were the optimal k length seeds, so they should be part of the optimal overall seeds. It was about 11% better than Bidirectional Hobbes. It also ran 14% faster, as no Hobbes calculations needed to be done. Comparing this to the version with exact frequencies, the theoretically lowest possible frequency sum was 15.6%

lower. So in this case, using the predictor was not as accurate as in Bidirectional Hobbes, but reasonable accurate. Since there is a gap, it is likely that we can reduce the predictive version lower with improvements to the predictor and reach the lower bound.

7 Conclusions

The use of the Bidirectional Frequency predictor was able to reduce frequency sum of seeds in most cases. We investigated predictor specific tests in section 5, and found that the predictor did well in predicting seeds with high frequencies. Ultimately, we do not want there to be any high frequency seeds when the read in split into seeds, to it is much better for the predictor to perform well on high frequency seeds and not as well on low frequency seeds since low frequency seeds do not have the same amount of impact. Also the predictor functioned well in the comparative sense and was able to judge whether a seed had higher frequency than another with $60-88\%$ accuracy, depending on the seed length.

When we added the Bidirectional predictor to Hobbes and the Uniform seed selector, it showed marked improvements in most cases. The predictor was able to achieve a significant reduction in the sum of the frequencies without much overhead in runtime and initialization time.

There are a considerable number of possible improvements to the predictor which could increase its performance and accuracy. This method has shown to be a viable candidate in lowering the frequency sum of seeds. It has a lot of potential for future work.

8 Future Work

There are a wide variety of future avenues to consider in this research. Developing the frequency predictor was only the first step in determining in the effectiveness of this method. This first iteration showed that there is some promise to this method if it can be improved and optimized.

8.1 Iterative Prediction

In each of the predictive seed selectors, we run a linear pass of the seed division scheme which optimizes the divisions of two adjacent seeds. This is only done once. It is plausible that if this is repeated multiple times, the frequency would reduce further as the optimal divisions would get updated. However the reduction may not be by much, and this would not warrant the extra time needed to compute the divisions. Therefore, it could be useful to analyze how multiple iterations affect the frequency reduction.

8.2 Optimizing for Comparative Prediction

The earlier results showed that the predictor worked reasonably well in determining if one seed had a larger frequency than other. This was not originally the purpose of the predictor, and it could be possible to build a predictor specifically for this purpose and determine the performance benefits of using a comparative prediction versus a numerical prediction.

8.3 Extension Patterns

By examining the predictive frequency of a seed across the various extensions, there is a pattern that emerges. For seeds which have large variation in the predicted frequency, the variation is not randomly distributed, but appears in chunks. This is because the chunk corresponds to a certain substring of the seed which has high frequency. By analyzing the different frequencies across the extensions, it is likely that we can create an much more accurate aggregated guess instead of directly using the predictor.

8.4 Randomized Division

In the implementations of the predictors, the seed extension was performed from left to right, so that the division of the second seed was dependent on the first, and the division of the third seed was dependent on the first two, and so forth. This creates a bias in the range of possible divisions. A possible solution is randomizing which seed division we optimize.

8.5 Probabilistic Models

Considering second order statistic like standard deviation may be helpful instead of just considering the average. By using the mean and standard deviation and assuming a rough distribution, we would be able to construct a probabilistic distribution of the possible frequency of a read, and combine these distributions together to form the probabilistic model for the frequency of all the seeds of a read.

8.6 Machine Learning

Ultimately, the problem here which the predictor is solving is predicting the frequency of substrings in a long string. As mentioned before, we want to learn a model to estimate the function which maps seeds to frequency. It is possible that there may be structure in the seed frequencies, and they could be predicted from a machine learning model using a certain set of features. This is an interesting area which could show promise.

References

[1] Ahmadi A, Behm A, Honnalli N, Li C, Weng L, and Xie X. Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic Acids Research*, 40(6), 2012.

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

[3] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating read mapping with fasthash. *BMC Genomics*, 14.

[4] 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.*

9 Appendix A

9.1 Results of Frequency Average Test

Freqs:	0	5	10	50	100	500
Count:	3	370	1374	25497	79449	396593
Avg Freq:	0.00	3.83	8.21	34.83	76.60	296.36
(0, 1):	7.00	20.69	27.90	87.38	174.12	525.32
(1, 0):	7.33	18.72	27.47	87.28	174.09	528.46

Table 25: Seed length 11

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	924687	1989959	261184	273495	70693	3505
Avg Freq:	758.52	2063.03	6921.88	32482.25	204641.16	2457548.71
(0, 1):	1038.68	2524.14	8008.69	32842.72	208485.70	2074929.54
(1, 0):	1039.14	2542.68	8089.95	32705.69	216075.70	2074192.63

Table 26: Seed length 11

Freqs:	0	5	10	50	100	500
Count:	276	7852	15058	214606	155766	2045663
Avg Freq:	0.00	3.40	8.24	28.44	76.68	276.09
(0, 2):	15.93	36.43	60.24	140.76	297.01	564.25
(1, 1):	17.46	39.70	62.21	143.90	284.38	558.51
(2, 0):	16.09	30.70	57.89	145.43	290.71	566.10

Table 27: Seed length 12

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	688840	555620	108938	179399	52030	2761
Avg Freq:	693.54	2074.50	7095.21	36617.05	196559.76	2059168.57
(0, 2):	1234.97	3751.62	9835.89	39765.91	189361.26	1467818.82
(1, 1):	1207.22	3685.46	10587.19	38192.25	201147.23	1478942.37
(2, 0):	1242.16	3843.22	10190.89	42736.41	194746.92	1470746.04

Table 28: Seed length 12

Freqs:	0	5	10	50	100	500
Count:	3802	98737	98153	749457	1001262	1375999
Avg Freq:	0.00	3.16	7.87	33.10	73.31	206.49
(0, 3):	28.29	66.67	128.05	205.02	310.86	686.97
(1, 2):	33.65	64.81	109.04	206.42	309.03	669.90
(2, 1):	43.19	61.63	118.76	210.73	315.53	668.94
(3, 0):	32.25	63.77	129.10	207.58	313.69	700.56

Table 29: Seed length 13

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	192758	238362	80732	146139	39201	2207
Avg Freq:	697.59	2349.41	7300.64	38106.12	188165.50	1723067.42
(0, 3):	2355.34	6087.01	14133.42	41820.91	162365.18	1033792.12
(1, 2):	2217.11	6700.99	13895.22	40395.22	171603.17	1046385.28
(2, 1):	2223.30	6835.36	14281.67	42100.32	172929.41	1049067.41
(3, 0):	2397.17	6688.31	15609.27	44224.88	171740.53	1041373.26

Table 30: Seed length 13

Freqs:	0	5	10	50	100	500
Count:	119893	1296668	782215	1012545	169448	228469
Avg Freq:	0.00	2.90	7.71	21.54	70.75	231.81
(0, 5):	94.76	142.80	196.01	378.19	1050.82	2593.59
(1, 4):	88.25	137.32	196.19	365.75	1028.75	2575.29
(2, 3):	92.72	139.16	192.85	357.71	1051.37	2620.70
(3, 2):	93.05	139.42	187.77	364.28	1062.28	2669.95
(4, 1):	97.40	139.39	190.59	373.66	1074.14	2725.39
(5, 0):	92.20	143.02	198.27	391.13	1134.53	2930.94

Table 31: Seed length 15

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	67449	147865	68003	110002	22731	1521
Avg Freq:	708.27	2513.11	7125.24	38015.44	173055.18	1191151.41
(0, 5):	6500.26	9933.95	11493.09	33278.47	123185.21	499685.63
(1, 4):	5368.47	9709.77	12161.65	33446.83	131134.11	509271.39
(2, 3):	5750.41	9122.51	12469.85	34774.53	134129.87	514628.16
(3, 2):	5787.67	9534.10	12678.14	34910.12	137436.07	518084.84
(4, 1):	5965.05	10939.14	13353.92	33978.20	142319.18	519248.13
(5, 0):	7576.56	11455.24	13135.02	34583.92	138267.35	513356.68

Table 32: Seed length 15

Freqs:	0	5	10	50	100	500
Count:	708601	2509784	107476	190386	67327	137416
Avg Freq:	0.00	1.35	7.64	24.33	72.14	239.50
(0, 8):	146.54	185.98	905.40	1727.28	2540.82	3525.44
(1, 7):	140.95	181.57	826.45	1576.29	2497.08	3581.61
(2, 6):	141.73	181.21	820.99	1522.09	2392.90	3517.53
(3, 5):	143.74	177.98	836.01	1550.87	2286.16	3400.05
(4, 4):	139.46	177.38	815.23	1568.57	2444.74	3401.16
(5, 3):	143.27	178.04	835.76	1571.80	2377.97	3628.36
(6, 2):	143.49	183.79	868.36	1620.57	2579.40	4011.43
(7, 1):	136.92	192.24	930.45	1831.74	2920.05	4070.44
(8, 0):	136.26	204.77	1092.63	2148.54	3045.26	4079.51

Table 33: Seed length 18

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	45861	122776	49277	77158	10747	0
Avg Freq:	723.75	2557.84	6943.46	34782.67	211144.09	0.00
(0, 8):	5991.70	6760.99	8559.45	22857.12	99409.11	0.00
(1, 7):	6016.09	6928.29	8577.15	23204.92	102778.90	0.00
(2, 6):	6210.54	7092.48	8661.80	23809.86	104476.53	0.00
(3, 5):	6208.34	7406.93	8921.84	23769.40	107979.85	0.00
(4, 4):	6307.44	7736.48	9203.70	23643.40	111953.00	0.00
(5, 3):	6643.53	7744.20	9172.26	24029.35	111473.34	0.00
(6, 2):	6878.12	7708.36	9333.54	24259.54	110395.75	0.00
(7, 1):	6856.99	7757.74	9359.86	24430.85	108938.98	0.00
(8, 0):	7031.62	7812.77	9304.02	24638.03	107708.63	0.00

Table 34: Seed length 18

Freqs:	0	5	10	50	100	500
Count:	784079	2571704	80831	157567	57769	117097
Avg Freq:	0.00	1.18	7.71	24.65	72.27	235.91
(0, 10):	173.60	215.36	1055.95	1571.64	2173.87	3091.49
(1, 9):	169.24	204.65	1044.15	1574.85	2180.26	3130.75
(2, 8):	166.14	200.77	999.83	1568.93	2233.88	3167.77
(3, 7):	168.09	196.57	970.62	1549.92	2228.54	3193.01
(4, 6):	163.16	195.68	973.62	1571.02	2236.01	3252.70
(5, 5):	164.58	194.46	958.81	1592.64	2237.82	3312.36
(6, 4):	164.65	197.35	1009.97	1637.17	2406.66	3398.32
(7, 3):	162.80	201.84	1060.22	1739.83	2503.88	3447.56
(8, 2):	163.55	208.42	1140.96	1831.39	2557.90	3480.85
(9, 1):	162.41	223.73	1237.81	1855.60	2520.55	3498.96
(10, 0):	163.08	248.42	1301.65	1897.36	2534.19	3502.16

Table 35: Seed length 20

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	40776	111922	37171	61976	5917	0
Avg Freq:	724.44	2578.90	7029.43	32161.38	204915.80	0.00
(0, 10):	4765.08	5345.54	7178.56	18479.25	82432.48	0.00
(1, 9):	4809.48	5413.55	7392.81	18658.86	82975.53	0.00
(2, 8):	4950.51	5468.12	7659.98	18811.26	84940.53	0.00
(3, 7):	5082.51	5624.04	7712.46	18808.93	87318.63	0.00
(4, 6):	5226.03	5804.72	7698.03	18856.53	89900.76	0.00
(5, 5):	5331.00	5916.56	7627.76	19003.34	90801.45	0.00
(6, 4):	5387.70	6004.87	7705.25	19110.16	90860.76	0.00
(7, 3):	5350.29	5996.10	7765.89	19356.38	89369.76	0.00
(8, 2):	5435.58	6007.84	7787.24	19644.72	87522.08	0.00
(9, 1):	5441.54	6029.58	7819.80	19807.91	85931.09	0.00
(10, 0):	5556.77	6043.95	7656.41	20133.16	85702.30	0.00

Table 36: Seed length 20

Freqs:	0	5	10	50	100	500
Count:	816832	2625940	72592	140757	52022	100360
Avg Freq:	0.00	1.15	7.70	24.66	72.72	236.19
(0, 12):	173.10	199.19	905.74	1343.88	1830.05	2675.92
(1, 11):	168.84	191.83	914.68	1337.32	1832.98	2696.23
(2, 10):	165.86	188.07	916.37	1352.53	1843.06	2720.94
(3, 9):	165.77	183.51	922.56	1365.61	1850.92	2762.44
(4, 8):	160.96	182.03	912.40	1410.90	1884.97	2820.11
(5, 7):	161.34	181.23	899.09	1424.98	1930.33	2881.49
(6, 6):	161.47	183.77	919.37	1447.82	2007.35	2945.90
(7, 5):	159.70	185.59	958.09	1484.91	2022.90	2962.68
(8, 4):	159.58	189.25	1022.16	1520.40	2046.71	2972.05
(9, 3):	159.35	196.29	1039.84	1536.76	2029.38	2971.38
(10, 2):	157.17	206.61	1051.87	1545.73	2045.62	2964.31
(11, 1):	155.64	218.98	1057.49	1554.12	2065.97	2977.41
(12, 0):	159.86	234.11	1074.41	1583.41	2096.32	3010.47

Table 37: Seed length 22

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	36672	98988	32518	46541	3587	0
Avg Freq:	724.35	2563.87	7153.46	29390.73	181018.39	0.00
(0, 12):	3802.03	4268.56	5565.08	15264.58	62429.14	0.00
(1, 11):	3721.10	4315.08	5782.58	15227.76	62678.87	0.00
(2, 10):	3749.85	4371.60	5980.80	15209.59	64034.95	0.00
(3, 9):	3821.07	4435.22	6104.65	15112.36	65598.70	0.00
(4, 8):	3909.15	4482.56	6156.09	15060.06	67876.14	0.00
(5, 7):	4005.25	4486.61	6204.85	15042.99	69483.14	0.00
(6, 6):	4096.32	4528.20	6240.87	15019.70	70488.12	0.00
(7, 5):	4094.39	4559.67	6210.18	15162.97	70393.73	0.00
(8, 4):	4157.24	4596.58	6148.24	15358.89	69674.25	0.00
(9, 3):	4197.97	4618.91	6182.53	15536.38	68435.59	0.00
(10, 2):	4280.30	4617.48	6055.88	15849.58	66993.67	0.00
(11, 1):	4306.76	4611.93	5985.54	16047.22	65893.89	0.00
(12, 0):	4388.75	4625.24	5862.69	16356.85	65397.89	0.00

Table 38: Seed length 22

Freqs:	0	5	10	50	100	500
Count:	853607	2690186	63853	121559	45202	80411
Avg Freq:	0.00	1.13	7.69	24.45	72.22	237.72
(0, 15):	146.39	161.42	745.59	1117.90	1432.92	2175.54
(1, 14):	143.22	157.72	741.96	1113.32	1450.59	2191.55
(2, 13):	142.28	155.32	742.24	1113.50	1468.47	2211.74
(3, 12):	140.84	152.94	744.50	1113.80	1488.26	2231.81
(4, 11):	137.74	151.21	747.98	1136.73	1507.86	2264.47
(5, 10):	136.92	149.96	758.14	1156.87	1562.92	2278.56
(6, 9):	136.75	150.58	775.75	1175.56	1595.29	2293.14
(7, 8):	135.22	151.41	781.20	1191.83	1606.70	2292.46
(8, 7):	133.99	153.00	788.82	1207.48	1611.67	2320.52
(9, 6):	133.73	155.26	794.12	1205.93	1619.84	2336.90
(10, 5):	131.98	159.16	791.62	1202.04	1614.01	2339.32
(11, 4):	129.84	163.92	794.20	1202.98	1599.41	2342.90
(12, 3):	129.98	167.40	797.13	1208.91	1597.39	2346.62
(13, 2):	130.87	171.57	808.33	1206.11	1576.41	2343.47
(14, 1):	133.28	178.27	811.29	1216.69	1583.12	2350.08
(15, 0):	135.17	186.75	813.20	1236.31	1592.80	2373.69

Table 39: Seed length 25

Freqs:	1000	5000	10000	100000	1000000	1000000000
Count:	30319	84540	25750	29748	1634	0
Avg Freq:	720.23	2505.33	6938.21	26740.63	142364.21	0.00
(0, 15):	2909.74	2962.16	4382.68	11781.10	43068.11	0.00
(1, 14):	2881.89	3009.02	4435.95	11702.59	42475.91	0.00
(2, 13):	2898.62	3065.92	4492.42	11684.95	41972.42	0.00
(3, 12):	2938.71	3103.43	4507.86	11633.23	41922.99	0.00
(4, 11):	2942.50	3134.16	4548.19	11519.45	43714.50	0.00
(5, 10):	2933.02	3164.20	4530.14	11386.56	45431.60	0.00
(6, 9):	2950.71	3204.56	4504.49	11300.62	45858.71	0.00
(7, 8):	2963.75	3216.70	4504.29	11286.43	46313.43	0.00
(8, 7):	2976.91	3219.91	4467.72	11338.04	46442.20	0.00
(9, 6):	3021.66	3223.47	4460.72	11411.87	46276.79	0.00
(10, 5):	3054.27	3202.91	4472.30	11575.54	46084.15	0.00
(11, 4):	3059.21	3190.74	4472.23	11778.68	44664.61	0.00
(12, 3):	3095.02	3181.34	4460.39	11938.43	43212.25	0.00
(13, 2):	3125.38	3178.95	4446.57	12088.07	43382.08	0.00
(14, 1):	3151.53	3162.02	4414.87	12177.38	43944.70	0.00
(15, 0):	3183.09	3140.36	4422.06	12312.28	45009.59	0.00

Table 40: Seed length 25

9.2 Results of Frequency Ratio Test

Freqs:	0	1	5	10	50	100
Count:	3	21	349	1374	25497	79449
Avg Freq:	0.00	1.00	4.00	8.21	34.83	76.60
(0, 1):	0.00	11.10	5.72	3.41	2.55	2.28
(1, 0):	0.00	8.95	5.01	3.38	2.53	2.28

Table 41: Seed length 11

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	396593	924687	1989959	261184	273495	74198
Avg Freq:	296.36	758.52	2063.03	6921.88	32482.25	311065.07
(0, 1):	1.85	1.38	1.24	1.16	1.06	1.01
(1, 0):	1.87	1.38	1.24	1.17	1.06	1.06

Table 42: Seed length 11

Freqs:	0	1	5	10	50	100
Count:	276	889	6963	15058	214606	155766
Avg Freq:	0.00	1.00	3.70	8.24	28.44	76.68
(0, 2):	0.00	16.04	10.40	7.33	5.07	4.00
(1, 1):	0.00	24.43	11.36	7.63	5.20	3.81
(2, 0):	0.00	18.23	9.17	7.12	5.17	3.95

Table 43: Seed length 12

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	2045663	688840	555620	108938	179399	54791
Avg Freq:	276.09	693.54	2074.50	7095.21	36617.05	290419.39
(0, 2):	2.13	1.78	1.81	1.40	1.16	0.97
(1, 1):	2.13	1.74	1.78	1.50	1.20	1.04
(2, 0):	2.13	1.79	1.84	1.45	1.26	1.00

Table 44: Seed length 12

Freqs:	0	1	5	10	50	100
Count:	3802	10366	88371	98153	749457	1001262
Avg Freq:	0.00	1.00	3.42	7.87	33.10	73.31
(0, 3):	0.00	41.69	21.12	16.35	6.97	4.30
(1, 2):	0.00	56.91	19.05	14.05	6.99	4.28
(2, 1):	0.00	43.08	19.01	15.32	7.08	4.38
(3, 0):	0.00	58.21	18.63	16.44	7.05	4.34

Table 45: Seed length 13

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	1375999	192758	238362	80732	146139	41408
Avg Freq:	206.49	697.59	2349.41	7300.64	38106.12	269974.05
(0, 3):	3.43	3.37	2.82	1.96	1.28	0.86
(1, 2):	3.36	3.16	3.03	1.93	1.15	0.90
(2, 1):	3.33	3.18	3.09	1.98	1.23	0.90
(3, 0):	3.48	3.44	3.06	2.19	1.42	0.91

Table 46: Seed length 13

Freqs:	0	1	5	10	50	100
Count:	119893	287608	1009060	782215	1012545	169448
Avg Freq:	0.00	1.00	3.45	7.71	21.54	70.75
(0, 5):	0.00	104.66	48.59	25.94	18.37	14.84
(1, 4):	0.00	104.44	46.24	25.88	17.93	14.48
(2, 3):	0.00	107.86	46.88	25.49	17.33	14.86
(3, 2):	0.00	106.20	47.02	24.87	17.67	15.08
(4, 1):	0.00	106.30	46.82	25.13	18.16	15.23
(5, 0):	0.00	105.99	48.37	26.19	18.84	16.19

Table 47: Seed length 15

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	228469	67449	147865	68003	110002	24252
Avg Freq:	231.81	708.27	2513.11	7125.24	38015.44	236906.59
(0, 5):	11.64	9.26	4.70	1.64	1.01	0.70
(1, 4):	11.91	7.65	4.48	1.75	1.01	0.75
(2, 3):	12.03	8.19	4.31	1.78	1.06	0.77
(3, 2):	12.20	8.27	4.44	1.81	1.09	0.79
(4, 1):	12.54	8.52	5.05	1.92	1.07	0.82
(5, 0):	13.13	10.86	5.42	1.87	1.09	0.80

Table 48: Seed length 15

Freqs:	0	1	5	10	50	100
Count:	708601	1996228	513556	107476	190386	67327
Avg Freq:	0.00	1.00	2.70	7.64	24.33	72.14
(0, 8):	0.00	142.18	132.59	119.70	79.80	36.34
(1, 7):	0.00	140.53	126.94	109.25	70.90	35.57
(2, 6):	0.00	140.14	126.18	108.76	69.43	34.02
(3, 5):	0.00	137.29	123.64	110.86	70.14	32.67
(4, 4):	0.00	137.10	122.66	108.22	71.06	34.86
(5, 3):	0.00	137.23	123.01	110.78	71.11	33.97
(6, 2):	0.00	141.04	129.65	115.66	74.10	36.66
(7, 1):	0.00	146.91	136.27	124.14	82.55	41.80
(8, 0):	0.00	155.12	146.70	144.72	99.73	43.69

Table 49: Seed length 18

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	137416	45861	122776	49277	77158	10747
Avg Freq:	239.50	723.75	2557.84	6943.46	34782.67	211144.09
(0, 8):	16.79	8.51	3.22	1.25	0.72	0.54
(1, 7):	16.94	8.53	3.30	1.25	0.74	0.56
(2, 6):	16.58	8.80	3.38	1.27	0.76	0.57
(3, 5):	15.81	8.79	3.50	1.31	0.76	0.60
(4, 4):	15.88	8.93	3.68	1.34	0.75	0.63
(5, 3):	16.88	9.46	3.68	1.35	0.76	0.62
(6, 2):	19.05	9.80	3.69	1.37	0.77	0.62
(7, 1):	19.50	9.77	3.73	1.38	0.78	0.61
(8, 0):	19.69	10.01	3.77	1.37	0.78	0.60

Table 50: Seed length 18

Freqs:	0	1	5	10	50	100
Count:	784079	2314973	256731	80831	157567	57769
Avg Freq:	0.00	1.00	2.84	7.71	24.65	72.27
(0, 10):	0.00	164.21	244.45	140.79	72.51	30.96
(1, 9):	0.00	157.75	226.81	138.92	72.59	31.10
(2, 8):	0.00	157.05	215.55	132.67	72.40	31.83
(3, 7):	0.00	153.75	211.97	128.71	70.93	31.74
(4, 6):	0.00	152.96	211.41	128.88	71.63	31.87
(5, 5):	0.00	151.73	210.92	126.64	72.77	31.83
(6, 4):	0.00	154.96	211.52	134.00	74.82	34.06
(7, 3):	0.00	158.44	215.87	141.06	79.90	35.48
(8, 2):	0.00	162.43	225.58	151.04	85.22	36.43
(9, 1):	0.00	171.10	252.11	164.35	87.25	35.92
(10, 0):	0.00	188.48	287.06	173.37	89.71	35.99

Table 51: Seed length 20

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	117097	40776	111922	37171	61976	5917
Avg Freq:	235.91	724.44	2578.90	7029.43	32161.38	204915.80
(0, 10):	14.69	6.78	2.58	1.05	0.62	0.47
(1, 9):	14.86	6.87	2.59	1.08	0.63	0.47
(2, 8):	15.05	7.07	2.61	1.12	0.64	0.48
(3, 7):	15.17	7.26	2.69	1.13	0.64	0.49
(4, 6):	15.42	7.47	2.79	1.12	0.63	0.51
(5, 5):	15.59	7.66	2.85	1.11	0.64	0.51
(6, 4):	16.14	7.74	2.90	1.12	0.64	0.52
(7, 3):	16.39	7.70	2.89	1.13	0.65	0.51
(8, 2):	16.58	7.81	2.89	1.14	0.66	0.49
(9, 1):	16.72	7.81	2.91	1.14	0.66	0.48
(10, 0):	16.72	7.95	2.94	1.12	0.67	0.48

Table 52: Seed length 20

Freqs:	0	1	5	10	50	100
Count:	816832	2416456	209484	72592	140757	52022
Avg Freq:	0.00	1.00	2.91	7.70	24.66	72.72
(0, 12):	0.00	160.84	235.37	120.75	62.33	26.12
(1, 11):	0.00	154.26	228.54	122.13	62.00	26.24
(2, 10):	0.00	151.18	222.88	122.63	62.73	26.35
(3, 9):	0.00	146.82	220.31	123.34	63.50	26.40
(4, 8):	0.00	145.83	217.16	122.12	65.60	26.92
(5, 7):	0.00	145.44	215.59	119.89	66.17	27.54
(6, 6):	0.00	147.33	219.91	122.49	67.25	28.65
(7, 5):	0.00	148.44	223.33	127.27	69.23	28.86
(8, 4):	0.00	150.48	231.65	135.90	71.12	29.17
(9, 3):	0.00	155.93	240.53	138.10	72.08	28.87
(10, 2):	0.00	164.93	250.36	140.01	72.24	29.09
(11, 1):	0.00	176.55	259.85	140.91	72.69	29.38
(12, 0):	0.00	190.02	274.22	143.04	74.33	29.83

Table 53: Seed length 22

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	100360	36672	98988	32518	46541	3587
Avg Freq:	236.19	724.35	2563.87	7153.46	29390.73	181018.39
(0, 12):	12.65	5.44	2.09	0.80	0.55	0.39
(1, 11):	12.71	5.34	2.10	0.83	0.55	0.39
(2, 10):	12.89	5.37	2.12	0.86	0.55	0.40
(3, 9):	13.09	5.48	2.16	0.88	0.55	0.41
(4, 8):	13.32	5.59	2.17	0.89	0.55	0.42
(5, 7):	13.68	5.73	2.18	0.90	0.55	0.43
(6, 6):	14.01	5.87	2.20	0.90	0.55	0.44
(7, 5):	14.10	5.87	2.22	0.90	0.55	0.44
(8, 4):	14.16	5.95	2.24	0.89	0.56	0.43
(9, 3):	14.16	6.01	2.25	0.89	0.56	0.43
(10, 2):	14.17	6.12	2.25	0.87	0.57	0.42
(11, 1):	14.23	6.15	2.25	0.86	0.58	0.41
(12, 0):	14.36	6.26	2.28	0.84	0.59	0.40

Table 54: Seed length 22

Freqs:	0	1	5	10	50	100
Count:	853607	2504219	185967	63853	121559	45202
Avg Freq:	0.00	1.00	2.91	7.69	24.45	72.22
(0, 15):	0.00	136.21	183.55	98.93	52.06	20.38
(1, 14):	0.00	132.46	182.14	98.57	51.70	20.67
(2, 13):	0.00	130.11	181.36	98.50	51.95	20.93
(3, 12):	0.00	127.49	182.09	98.81	52.00	21.20
(4, 11):	0.00	125.69	181.43	99.24	53.29	21.50
(5, 10):	0.00	124.60	179.77	100.66	54.36	22.26
(6, 9):	0.00	125.12	179.72	103.08	55.19	22.74
(7, 8):	0.00	125.44	182.40	103.78	55.97	22.88
(8, 7):	0.00	126.19	187.11	105.28	56.91	22.96
(9, 6):	0.00	128.32	188.02	105.90	56.95	23.09
(10, 5):	0.00	132.06	190.64	105.68	56.58	23.00
(11, 4):	0.00	136.63	193.59	106.01	56.71	22.79
(12, 3):	0.00	140.36	194.40	106.25	56.91	22.75
(13, 2):	0.00	144.45	197.25	107.47	56.61	22.44
(14, 1):	0.00	151.15	199.61	107.78	57.16	22.58
(15, 0):	0.00	159.28	204.48	108.34	58.01	22.72

Table 55: Seed length 25

Freqs:	500	1000	5000	10000	100000	1000000000
Count:	80411	30319	84540	25750	29748	1634
Avg Freq:	237.72	720.23	2505.33	6938.21	26740.63	142364.21
(0, 15):	10.39	4.15	1.45	0.64	0.45	0.30
(1, 14):	10.48	4.12	1.46	0.65	0.45	0.30
(2, 13):	10.61	4.13	1.49	0.66	0.45	0.30
(3, 12):	10.72	4.20	1.51	0.67	0.45	0.30
(4, 11):	10.88	4.21	1.52	0.67	0.44	0.31
(5, 10):	10.98	4.20	1.53	0.67	0.44	0.33
(6, 9):	11.09	4.23	1.55	0.67	0.44	0.33
(7, 8):	11.10	4.25	1.55	0.67	0.44	0.33
(8, 7):	11.24	4.26	1.56	0.66	0.44	0.34
(9, 6):	11.32	4.33	1.56	0.66	0.44	0.33
(10, 5):	11.30	4.38	1.55	0.66	0.45	0.33
(11, 4):	11.30	4.39	1.55	0.66	0.46	0.32
(12, 3):	11.30	4.44	1.55	0.66	0.46	0.31
(13, 2):	11.25	4.47	1.55	0.65	0.47	0.31
(14, 1):	11.25	4.51	1.54	0.65	0.47	0.31
(15, 0):	11.34	4.55	1.54	0.65	0.48	0.32

Table 56: Seed length 25

9.3 Results of Frequency Comparison Test

(left, right)	% of correct comparisons
(0, 1):	0.885686
(1, 0):	0.885490

Table 57: Seed length 11

(left, right)	% of correct comparisons
(0, 2):	0.838487
(1, 1):	0.834117
(2, 0):	0.838308

Table 58: Seed length 12

(left, right)	% of correct comparisons
(0, 3):	0.802248
(1, 2):	0.799497
(2, 1):	0.799555
(3, 0):	0.801996

Table 59: Seed length 13

(left, right)	% of correct comparisons
(0, 5):	0.734238
(1, 4):	0.733823
(2, 3):	0.734016
(3, 2):	0.734292
(4, 1):	0.733973
(5, 0):	0.735255

Table 60: Seed length 15

(left, right)	% of correct comparisons
(0, 8):	0.644817
(1, 7):	0.648471
(2, 6):	0.647196
(3, 5):	0.647320
(4, 4):	0.649201
(5, 3):	0.649550
(6, 2):	0.649888
(7, 1):	0.649108
(8, 0):	0.649469

Table 61: Seed length 18

(left, right)	% of correct comparisons
(0, 10):	0.627535
(1, 9):	0.628668
(2, 8):	0.628174
(3, 7):	0.628991
(4, 6):	0.629764
(5, 5):	0.629845
(6, 4):	0.630774
(7, 3):	0.630830
(8, 2):	0.631497
(9, 1):	0.631189
(10, 0):	0.630728

Table 62: Seed length 20

(left, right)	% of correct comparisons
(0, 12):	0.616800
(1, 11):	0.617805
(2, 10):	0.617108
(3, 9):	0.617824
(4, 8):	0.618703
(5, 7):	0.618959
(6, 6):	0.619920
(7, 5):	0.619714
(8, 4):	0.620441
(9, 3):	0.621070
(10, 2):	0.621115
(11, 1):	0.619363
(12, 0):	0.620226

Table 63: Seed length 22

(left, right)	% of correct comparisons
(0, 15):	0.605822
(1, 14):	0.606799
(2, 13):	0.605634
(3, 12):	0.606399
(4, 11):	0.606932
(5, 10):	0.607885
(6, 9):	0.608510
(7, 8):	0.609034
(8, 7):	0.609307
(9, 6):	0.610289
(10, 5):	0.610546
(11, 4):	0.609536
(12, 3):	0.610421
(13, 2):	0.609668
(14, 1):	0.608081
(15, 0):	0.606699

Table 64: Seed length 25