FINDING APPROXIMATE REPEATS WITH MULTIPLE SPACED SEEDS

FINDING APPROXIMATE REPEATS IN DNA SEQUENCES USING MULTIPLE SPACED SEEDS

By SARAH BANYASSADY, B.S.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree Master of Science

McMaster University MASTER OF SCIENCE (2015) Hamilton, Ontario (Computational Science and Engineering)

TITLE: Finding Approximate Repeats in DNA Sequences Using Multiple Spaced Seeds AUTHOR: Sarah Banyassady, B.S. (Amirkabir University) SUPERVISOR: Professor William F. Smyth NUMBER OF PAGES: xv, 84

Abstract

In computational biology, genome sequences are represented as strings of characters defined over a small alphabet. These sequences contain many repeated subsequences, yet most of them are similarities, or approximate repeats. Sequence similarity search is a powerful way of analyzing genome sequences with many applications such as inferring genomic evolutionary events and relationships. The detection of approximate repeats between two sequences is not a trivial problem and solutions generally need large memory space and long processing time. Furthermore, the number of available genome sequences is growing fast along with the sequencing technologies. Hence, designing efficient methods for approximate repeat detection in large sequences is of great importance.

In this study, we propose a new method for finding approximate repeats in DNA sequences and develop the corresponding software. A common strategy is to index the locations of short substrings, or seeds, of one sequence and store them in an efficiently searchable structure. Then, scan the other sequence and look up the structure for matches with the stored seeds. A novel feature of our method is its efficient use of spaced seeds, substrings with gaps, to generate approximate repeats. We have designed a new space-efficient hash table for indexing sequences with multiple spaced seeds. The resulting seed-matches are then extended into longer approximate repeats using dynamic programming. Our results indicate that our hash table implementation requires less memory than previously proposed hash table methods, especially when higher similarities between approximate repeats are desired. Moreover, increasing the length of seeds does not significantly increase the space requirement of the hash table, while allowing the same similarities to be computed faster.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Bill Smyth, for his support during my studies, his useful comments, remarks, and commitment throughout this thesis. I would like to thank Professor Brian Golding for introducing me to the topic, his advice, assistance, and guidance on the way.

Table of Contents

List of Figures

List of Tables

List of Symbols

d	number of divided subsequences
D_{max}	maximum edit distance between repeat copies
h	hash function
h_{ip}	hit of seed s_i at position p of the reference sequence
h_{iq}	hit of seed s_i at position q of the query sequence
K	indexing step size
k	number of spaced seeds/a constant depending on the context
KEY_{ip}	key value of h_{ip}
KEY_{iq}	key value of h_{iq}
l	seed length
L_{min}	minimum length of approximate repeats
M	alignmnet matrix
m	length of the query sequence/size of the main hash table depending on the chapter
m_j	size of the secondary hash table of entry j
n	length of the reference sequence
n_1	length of the reference sequence
n_2	length of the query sequence

p	position in the reference sequence
p_l	left position of a repeat in the reference sequence
p_r	right position of a repeat in the reference sequence
$prev$	array of previous occurrences
Q	query sequence
q	position in the query sequence
q_l	left position of a repeat in the query sequence
q_r	right position of a repeat in the query sequence
R	reference sequence
s_i	seed number i
S_{gap}	gap score
S_{lext}	score of left extended segment
S_{match}	match score
S_{min}	minimum score of a repeat
$S_{mismatch}$	mismatch score
S_{rext}	score of right extended segment
S_{total}	total repeat score
T	main hash table
t	number of threads
T_j	secondary hash table of entry j
$T_{address}$	main hash table of addresses
T_{size}	main hash table of size counters

w	seed weight
w_i	weight of seed number i
X_{drop}	threshold score of X-drop method

Chapter 1

Introduction

1.1 Problem Statement

Locating various types of repeated segments in genome sequences is an important and well-studied problem which plays a major role in analyzing genomic data. The repeated segments are classified into different categories depending on their length, frequency, degree of similarity, etc. Exact repeats which are identically matching substrings form one of the interesting repeat categories. There exist numerous studies on finding exact repeats in genome sequences [1], [2]. Since exact repeats are not informative for many biological questions [3], researchers are also interested in approximate repeats. An approximate repeat is a repeating substring in which the instances are similar, but not identical. The similarity degree between string segments is measured by some metrics which define match, mismatch, and/or gap scores.

The problem of finding approximate repeats, in a general form, is as follows. We have two genome sequences, a reference and a query, and a similarity metric. Our goal is to find all pairs of approximate repeats between the reference and the query with at least a certain degree of similarity, according to the given metric.

1.2 Background

A considerable number of research studies have been done so far to find approximate repeats in genome sequences. The preliminary algorithm for this problem is the exhaustive search of all possible solutions proposed in 1981 [4]. It gives all pairs of approximate repeats with the desired similarity degree, but due to its quadratic time complexity it can only be used for small sequences.

Subsequent methods considered non-exhaustive search and employed a heuristic to restrict the search space and speed up the process. FASTA [5], [6] and BLAST [7] are two of the well-known early works of such methods. BLAST innovated a filtration technique known as the *hit and extend* approach. The key idea behind the hit and extend approach is to seek only the segment pairs of the reference and the query that fit some patterns, presumed to be a potential similarity [8]. These patterns are called *seeds* and the corresponding fitted substrings appear in both sequences are called *hits*. The neighbouring regions of the hits are then investigated in order to extend the hits into longer approximate repeats. Later, a family of tools was derived from BLAST [9]–[11]. The seed pattern of the BLAST family is an exact match of k continuous letters. This means that a hit in BLAST consists of k consecutive letter-matches with no mismatches in between.

In 2002, a new seed pattern, *spaced seed*, was introduced in PatternHunter [12]. Spaced seeds improved the sensitivity and speed by allowing mismatches to occur within the hits. In a spaced seed, some gaps are arranged between the consecutive letter-matches to capture mismatches. It has been shown later that more spaced seeds can raise the sensitivity even further with a small cost [13]–[16]. Therefore, the spaced seed model was extended into *multiple spaced seeds* as used in PatternHunter II [13].

Most satisfactory methods for this problem rely on the hit and extend approach [17].

The common way of implementing such a method comprises three steps: 1) indexing the locations of the reference sequence hits in an efficiently searchable structure; 2) scanning the query sequence and looking up the structure for the hits of the query sequence; 3) extending the matching hit pairs from left and right, in both reference and query sequences, to get longer approximate repeats.

1.3 Motivation and Objectives

Despite decades of research on this problem, improving similarity search efficiency still remains of great importance. Genome sequences are usually millions of letters or larger. Therefore, indexed based approaches require substantial memory resources only for indexing the reference sequence. Moreover, the amount of sequenced genome data is continuously growing. Considering the above point and the fact that some genome similarities are not yet found by the existing tools, we see a need to find more space-efficient, sensitive, and faster tools for similarity searching in genome sequences.

In this project, we present a new software program for computing pairs of approximate repeats between two DNA sequences. We use the source code of another repeat finding program, E-MEM [18], as a framework. E-MEM is a software for computing maximal exact matches in very large genomes based on the hit and extend approach. It exploits a BLAST-like continuous seed to detect the hits and a hash table as the lookup structure. Since we aim to find approximate repeats, the E-MEM seed in our program is replaced by multiple spaced seeds. The key advantage of spaced seed over continuous seed is that it provides greater sensitivity, i.e., increases the chance of finding approximate repeats. On the other hand, for technical reasons explained in Chapter 4, use of spaced seed forces us to fully index the reference sequence, while E-MEM's use of continuous seed does not require a full-indexed reference sequence.

This increases the amount of data we store in the hash table. Hence, we redesign the E-MEM hash table, and make a new flexible and space-efficient hash table with greater capacity to handle numerous hits produced by multiple spaced seeds.

1.4 Thesis Organization

The rest of this document is organized as follows. Definitions and background information are given in Chapter 2. In Chapter 3, we review the literature on the problem of finding approximate repeats. Chapter 4 describes the methodology. In this chapter, the software design and implementation details are presented. The hash table structure and memory issues are also discussed. We also provide the program usage instructions, and explain its available options and their corresponding parameters. In Chapter 5, we conduct experiments on some human and mouse chromosomes in order to compare the execution time and memory requirement of our software with two similar tools, PatternHunter II [13] and YASS [19]. We test the software with various spaced seeds and in different parameter settings to study the impact of these changes on the software performance. We also show the trade-off between the execution time and the number of output repeats for different settings. Finally, conclusions are drawn in Chapter 6.

Chapter 2

Definitions and Background Information

In this chapter, we define the concepts and terms used throughout this thesis. Section 2.1 describes the biological sequences we are dealing with. The precise definition of exact and approximate repeats are given in Section 2.2. In Section 2.3, we discuss how sequence similarity is usually measured in this context and define different similarity metrics. Section 2.4 is to clarify one of our frequently used terms, the hit and extend approach. The concepts of hits and seeds which arise from the hit and extend approach are also defined in this section.

2.1 Biological Sequences

A biological sequence is a molecule of nucleic acid or protein. It can be organized into classes based on the underlying molecule type: DNA, RNA, or protein. This study focuses on DNA sequences.

2.1.1 DNA Sequences

DNA is a usually very long and unbroken nucleic acid molecule. The building blocks of this molecule are nucleotides. Nucleotides consist of four types of bases: **A**denine,

Cytosine, **G**uanine, and **T**hymine, denoted respectively by letters A, C, G, and T. The process of obtaining the order of the four nucleotide bases in a DNA molecule is called DNA sequencing. Using sequencing technologies, a DNA sequence can be represented as a string over the alphabet $\Sigma = \{A, C, G, T\}$.

2.2 Repeats

Let $x = x[1 \ldots n]$ denote a string of length n, where $x[i \ldots j]$ is a substring from position i to position j. A ***repeat*** in string x is a collection of identical or similar substrings of x which occur more than once. In Figure 2.1, the repeating substring CAC makes a repeat which occurs 3 times at positions 1, 3 and 10.

$$\begin{array}{rcccccccccccc} \text{index:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ x = & \mathrm{C} & \mathrm{A} & \mathrm{C} & \mathrm{A} & \mathrm{C} & \mathrm{T} & \mathrm{G} & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{A} & \mathrm{C} \end{array}$$

Figure 2.1: An example of a repeat.

Suppose u is a repeating substring of length l in x which occurs at starting positions i_1, i_2 and i_3. Each instance of the repeating substring u, including $x[i_1 \ldots i_1 + l]$, $x[i_2 \ldots i_2 + l]$, and $x[i_3 \ldots i_3 + l]$, is called a ***copy*** or an ***occurrence*** of the repeat. In the example of Figure 2.1, $x[1 \ldots 3]$, $x[3 \ldots 5]$, and $x[10 \ldots 12]$ are the repeat copies.

A ***repeat pair*** is any two copies of a repeating substring within a sequence or between two sequences.

2.2.1 Exact Repeats

An ***exact repeat*** is a repeat in which the copies are exactly identical substrings. For example, in Figure 2.2, the repeating substring $u =$ACCT makes an exact repeat pair with two copies at positions 1 and 7.

$$\begin{array}{rcccccccccccc} \text{index:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ x = & \mathrm{A} & \mathrm{C} & \mathrm{C} & \mathrm{T} & \mathrm{G} & \mathrm{T} & \mathrm{A} & \mathrm{C} & \mathrm{C} & \mathrm{T} & \mathrm{T} & \mathrm{T} \end{array}$$

Figure 2.2: Example of exact and approximate repeats in x.

2.2.2 Approximate Repeats

An ***approximate repeat*** is a repeat in which the copies are "similar", but not necessarily identical. Again in Figure 2.2, the two substrings $x[1 \dots 6]$ = ACCTGT and $x[7 \dots 12]$ = ACCTTT form an approximate repeat pair.

2.3 Similarity Metrics

The similarity between the copies of an approximate repeat can be quantified by distance metrics or scoring systems, both are defined below.

2.3.1 Distance

The ***distance*** between two strings is the minimum number of operations required to transform one string into the other. Typical transformation operations include:

- ***substitution*** of one letter for another, for example substituting T for G transforms ACCTGT to ACCTTT.
- ***insertion*** of a single letter, for example inserting one T transforms ACCGT to ACCTGT.
- ***deletion*** of a single letter, for example deleting one G transforms ACCTGT to ACCTT.

Among different kinds of distances, the two following metrics are more commonly used in biological applications:

2.3.1.1 Hamming Distance

The ***Hamming distance*** is defined only between two strings of equal length and is the minimum number of substitutions that can transform one string into the other [20]. For example, the Hamming distance between x_1 = AACAGTT and x_2 = ACCTTTT equals 3, because the 3 substitutions A→C, A→T, and G→T are necessary to change x_1 to x_2.

2.3.1.2 Edit Distance

The ***edit distance*** is the minimum number of insertions, deletions, and substitutions that can transform one string into the other [20]. For example, in order to transform x_1 = ACCTTGT to x_2 = ACTTGG at least one deletion of C and one substitution of G for T are required. Therefore, the edit distance between x_1 and x_2 is 2.

If the distance between two substrings is less than a given threshold, D_{max}, we say that the two substrings are similar enough to be copies of an approximate repeat.

2.3.2 Scoring Systems

Another way of quantifying the similarity between two strings is to align the two strings, assign a score to each aligned pair and sum up the scores of all pairs. In an alignment, one of the following states applies to each letter of one string:

- ***match*** if it is aligned with an identical letter of the other string.
- ***mismatch*** if it is aligned with a non-identical letter of the other string.
- ***gap*** if it is aligned with nothing.

	A	**C**	**G**	**T**	−
A	+5	-1	-2	-1	-3
C	-1	+5	-3	-2	-4
G	-2	-3	+5	-2	-2
T	-1	-2	-2	+5	-1
−	-3	-4	-2	-1	

Figure 2.3: A sample scoring matrix for DNA sequence alignment. The diagonal elements are the match scores. The rightmost column and the bottom row contain the gap scores. All the other elements are the scores of different mismatches.

The scoring systems are to specify the score of these states. A scoring scheme usually rewards a match with a positive number and penalizes a mismatch or a gap with negative numbers.

To provide a comparison between distance and score, a mismatch is interpreted as a substituted letter and a gap as an inserted/deleted letter in one of the strings. Therefore, when transforming one string into the other, the substitution operation is used to correct mismatches and insertion/deletion operation is used to correct gaps. The above-proposed distance definitions assign unit cost to every transformation operation, while in scoring schemes various scores are assigned to various alignment states. A scoring scheme may also assign different scores to different mismatch states, in case the substitution of one letter for another is more probable than some other substitution. In this case, a *scoring matrix* or *substitution matrix* is created to define a score for each specific pair of letters [20]. Figure 2.3 shows an example of a scoring matrix for DNA sequence alignment.

2.3.2.1 Optimal Alignment

Any two strings can be aligned in different ways. Among all the possible alignments, the one with the maximum score is chosen as the ***optimal alignment***. The optimal alignment score is the metric of similarity level. For example, using the matrix defined in Figure 2.3 for scoring the alignments of x_1 = AGTGATG and x_2 = GTTAG, the optimal alignment is the following with the score of 14:

$$\begin{array}{ccccccc} \text{A} & \text{G} & \text{T} & \text{G} & \text{A} & \text{T} & \text{G} \\ - & \text{G} & \text{T} & \text{T} & \text{A} & - & \text{G} \\ \hline -3 & +5 & +5 & -2 & +5 & -1 & +5 = 14 \end{array}$$

The optimal alignment score of two substrings must be above a given threshold, S_{min}, in order for them to be similar enough to be identified as two copies of an approximate repeat

2.4 The Hit and Extend Approach

The hit and extend approach, initiated by Altschul et al. [7], is a standard heuristic method for finding repeat pairs within a sequence or between two sequences. It involves two steps:

1. Hit identification: in this first step, we search for short identical or similar substrings, called ***hits***. The hits can be found quickly using some patterns defined below.

2. Hit extension: in this second step, we look for similarities around the hits and extend both copies of a hit from left and right to generate longer repeats.

2.4.1 Seeds

Seeds are short string patterns used for identifying hits in the first step of the hit and extend approach. Here, we introduce some various seed models:

2.4.1.1 Consecutive Seed

The simplest kind of seed captures k consecutive matches between two strings. A consecutive seed is represented by k successive 1's each 1 denotes a required match. For example, $s = 1111111111$ is a consecutive seed of length $l = 10$ which identifies any 10 consecutive matches between two strings, as shown in Figure 2.4. The substrings captured by the seed are the hits.

$$\begin{array}{rccccccccccccc}
s = & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
x_1 = & \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A} & \mathrm{G} & \mathrm{A} & \mathrm{A} & \mathrm{C} & \mathrm{T} & \mathrm{A} \\
x_2 = & & \mathrm{C} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A} & \mathrm{G} & \mathrm{A} & \mathrm{A} & \mathrm{C} & \mathrm{T} & \mathrm{C}
\end{array}$$

Figure 2.4: A consecutive seed identifies 10 consecutive matches between x_1 and x_2. Here, $x_1[3 \ldots 12] = x_2[2 \ldots 11] =$ CGTTAGAACT are the matching hits.

2.4.1.2 Spaced Seed

A spaced seed captures k matches while some fixed positions in between are allowed to mismatch. In other words, the spaced seed allows some mismatches in certain positions of the hits. The spaced seed is a string over the alphabet $\Sigma = \{1, *\}$; where 1 denotes a required match and $*$ denotes a free position. A free position can have either a match or a mismatch. The number of 1's in a seed pattern is called the ***weight*** of seed. For example, $s = 111 * 111 * *1$ is a spaced seed of length $l = 10$ and weight $w = 7$. It seeks hits of length 10 with all positions matching except the 4^{th}, 8^{th}, and 9^{th} which are allowed to mismatch. This is shown in Figure 2.5.

$$\begin{array}{rccccccccccccc}
s = & & & 1 & 1 & 1 & * & 1 & 1 & 1 & * & * & 1 & \\
x_1 = & A & G & C & G & T & \mathbf{T} & A & G & A & \mathbf{C} & A & T & A \\
x_2 = & & C & C & G & T & \mathbf{A} & A & G & A & \mathbf{A} & A & T & C
\end{array}$$

Figure 2.5: A spaced seed identifies non-consecutive matches between x_1 and x_2. Mismatched letters are in bold type. $x_1[3\ldots12]$ = CGTTAGACAT and $x_2[2\ldots11]$ = CGTAAGAAAT are the matching hits.

2.4.1.3 Multiple Spaced Seeds

This model is a set of spaced seeds as shown in Figure 2.6a. Multiple spaced seeds identify a pair of hits between two strings if there is at least one seed in the set which identifies the hits. Figure 2.6b shows an example.

$$\begin{array}{l}
s_1 = 1\,1\,1*1\,1\,1**1 \\
s_2 = 1\,1\,1*1\,1*1\,1*1 \\
s_3 = 1\,1*1*1\,1\,1*1\,1\,1
\end{array}$$

(a)

$$\begin{array}{rcccccccccccccccccc}
s_1 = & & & 1 & 1 & 1 & * & 1 & 1 & 1 & * & * & 1 & & & & & \\
x_1 = & A & G & C & G & T & T & A & G & A & C & A & T & A & C & G & C & C \\
x_2 = & & C & C & G & T & A & A & G & A & A & A & T & C & C & G & T & C \\
s_2 = & & & & & & & 1 & 1 & 1 & * & 1 & 1 & * & 1 & 1 & * & 1
\end{array}$$

(b)

Figure 2.6: Hit identification using multiple spaced seeds. (a) is an example of multiple spaced seeds. (b) shows two pairs of matching hits between x_1 and x_2. One pair is $x_1[3\ldots12]$ and $x_2[2\ldots11]$ identified by s_1. The other pair is $x_1[7\ldots17]$ and $x_2[6\ldots16]$ identified by s_2.

2.4.1.4 Other Kinds

More complicated seed models include transition-constrained seeds [8], vector seeds [14], and neighbour seeds [17], to name a few. The readers are referred to [21], [22] for more information on seeds. Here, we only give a brief description of transition-constrained seeds.

Transition-constrained Seed:

DNA substitutions are of two types. ***Transitions*** are interchanges of A↔G or C↔T.

Transversions are all the other substitutions. Transitions are more frequent in real genomic data than transversions. For this reason, the transition-constrained seed model has been invented [8], [11]. The transition-constrained seed is a string over the alphabet $\Sigma = \{1, *, T\}$. Like the spaced seeds, 1 stands for a match and $*$ denotes a free position. Positions with T's can have either transition substitutions or matches, but not transversions. The ***weight*** of a transition-constrained seed is the sum of the number of 1's and half the number of T's. Figure 2.7 shows an example.

$$\begin{array}{r@{\,=\,}ccccccccccccc}
s & & & 1 & 1 & * & 1 & T & * & 1 & 1 & T & 1 & \\
x_1 & \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{G} & \underline{\mathrm{T}} & \mathrm{G} & \mathbf{G} & \mathbf{T} & \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{A} \\
x_2 & & \mathrm{C} & \mathrm{C} & \mathrm{G} & \underline{\mathrm{A}} & \mathrm{G} & \mathbf{A} & \mathbf{C} & \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{C}
\end{array}$$

Figure 2.7: A transition-constrained seed of weight $w = 7$ identifies hits between x_1 and x_2. Transitions are in bold type. The underlined letters indicate a transversion. $x_1[3 \ldots 12]$ = CGTGGTAGCT and $x_2[2 \ldots 11]$ = CGAGACAGCT are the matching hits.

Chapter 3

Literature Review

A large body of research has provided tools for comparing and aligning protein and nucleic acid sequences. In this chapter, we review some of the most influential or recent works. The focus of classical studies in this area has been on relatively short sequences such as a single protein or a single gene [23]. These studies give rigorous solutions by exhaustive search procedures.

3.1 Exhaustive Search Methods

Exhaustive search methods compare each letter of one sequence with all the letters in the other sequence. With these algorithms, comparison of a sequence of length n with a sequence of length m requires nm comparisons. The Smith and Waterman algorithm, proposed in the early 1980s [4], is one of the well-known exhaustive search methods.

3.1.1 Smith and Waterman Algorithm

The Smith and Waterman algorithm gives an exact solution to the problem of finding optimal local alignments between two sequences. It uses dynamic programming to find all the local alignments scoring above a given threshold. The algorithm works as described below.

A reference sequence R of length n, a query sequence Q of length m, and a scoring scheme are given. The scoring scheme includes a score $s(x, y)$ for each pair of letters and a *cost function* C_k for gaps of length k. To find the pairs of segments with high similarity between R and Q, the algorithm first sets a matrix M with $n + 1$ rows and $m + 1$ columns. We call M the dynamic programming matrix or the *alignment matrix*. The value of $M(i, j)$ represents the maximum similarity score of the two subsequences ending at $R[i]$ and $Q[j]$. M is filled according to the following formulas:

$$M(i,0) = M(0,j) = 0, \quad 0 \leq i \leq n \text{ and } 0 \leq j \leq m \tag{3.1}$$

$$M(i,j) = \max \left\{ \begin{array}{c} M(i-1,j-1) + s(R[i],Q[j]) \\ \max\limits_{k\geq 1}\{M(i-k,j) - C_k\} \\ \max\limits_{l\geq 1}\{M(i,j-l) - C_l\} \\ 0 \end{array} \right\}, \quad 1 \leq i \leq n \text{ and } 1 \leq j \leq m \tag{3.2}$$

In Equation (3.2), case 1 is interpreted as "$R[i]$ and $Q[j]$ are aligned". Case 2 and 3 respectively show the cases when $R[i]$ is at the end of a length k gap and when $Q[j]$ is at the end of a length l gap. Finally, case 4 indicates there is no similarity up to $R[i]$ and $Q[j]$. To find the pair of segments with the maximum score the maximum element of M is found. Then, the matrix elements leading to this maximum element are traversed using a backtracking procedure. Backtracking stops when it reaches a zero element. This process gives the optimal local alignment. Figure 3.1 shows a simple example. The next best alignment is found by backtracking from the second largest element of M that is not traversed in the first backtrack [4].

	–	**G**	**T**	**G**	**A**	**A**	**T**	**T**	**C**	**A**
–	0	0	0	0	0	0	0	0	0	0
G	0	2	1	2	1	0	0	0	0	0
A	0	1	1	1	4	3	2	1	0	2
C	0	0	0	0	3	3	2	1	3	2
T	0	0	2	1	2	2	5	4	3	2
T	0	0	2	1	1	1	4	7	6	5
A	0	0	1	1	3	3	3	6	6	8

(a)

	–	**G**	**T**	**G**	**A**	**A**	**T**	**T**	**C**	**A**
–	0	0	0	0	0	0	0	0	0	0
G	0	↖	←	↖	←	0	0	0	0	0
A	0	↑	↖	↑	↖	↖	←	←	0	↖
C	0	0	0	0	↑	↖	↖	↖	↖	←
T	0	0	↖	←	↑	↖	↖	↖	←	↖
T	0	0	↖	↖	↑	↖	↖	↖	←	←
A	0	0	↑	↖	↖	↖	↑	↑	↖	↖

(b)

```
G A A T T C A
| |   | |   |
G A C T T – A
```

(c)

Figure 3.1: The Smith and Waterman algorithm on the sequences R = GTGAATTCA and Q = GACTTA, assuming $s(x, y) = +2$ for matches, $s(x, y) = -1$ for mismatches, and $C_k = -k$ as the gap cost. (a) the alignment matrix M is filled using Equations (3.1) and (3.2). The gray cell is the maximum element of M. (b) shows how to trace back each element of M. The gray cells indicate the backtracking path from the maximum element 8. (c) the first optimal local alignment or the alignment with the maximum score 8.

The Smith and Waterman algorithm guarantees to find the optimal local alignment between the reference and query sequences. The time complexity of the described algorithm is $O(nm^2)$ and its space complexity is $O(nm)$. Using a linear cost function for gaps (C_k), the algorithm can be implemented so to run in $O(nm)$ time with no change in the space complexity.

The exhaustive algorithms work very well on short sequences; however, they are very time-consuming and impractical for large problems. Therefore, other solutions are needed which focus only on some regions of similarities between the two sequences and search with fewer comparisons. With this motivation, heuristic search methods have been proposed. Heuristic search methods may lose accuracy by missing some similarities, but are fast in practice.

3.2 Heuristic Search Methods

3.2.1 FASTA

FASTA [6] is a program for DNA and protein similarity searching first proposed as FASTP in 1985 [5]. In comparison with exhaustive search methods, FASTP improved the speed of searching a 500,000-letter protein sequence, by more than two orders of magnitude [6]. FASTA finds regions of similarity between two sequences using heuristic search with a short consecutive seed. The search algorithm of FASTA proceeds through five main steps:

1. A lookup table of positions is constructed for short substrings, or hits, in the first sequence. Then, the hits of the second sequence are looked up in the table of the first sequence. The seed length is a user-defined parameter which can have values from 1 to 6 for DNA sequences and 1 or 2 for protein sequences.

2. For each matching pair of hits found with the lookup table, the offset between the starting positions of the hits in the two sequences is calculated. The pairs with equal offset values are located on the same diagonal of the alignment matrix M (see Section 3.1.1) and can form diagonal regions. The diagonal regions are scored according to the seed length and offset value, and the 10 highest scoring regions are detected.

3. These 10 regions are rescored using a substitution matrix (see Section 2.3.2) and for each one of them, a subregion with maximal score is identified. The subregions are called initial regions.

4. The algorithm proceeds by checking whether multiple initial regions can be joined to increase scores. However, only those initial regions whose scores are

above a threshold are considered. Given a gap cost, FASTA joins compatible initial regions with maximal scores.

5. In the final step, a modification of the alignment method described by Needleman and Wunsch [24] and Smith and Waterman [4] is performed on the joined regions. The algorithm assumes that the final optimal alignment is adjacent to or includes the highest scoring initial region. Therefore, it performs a banded dynamic programming procedure around that region [6].

As mentioned above, FASTA is faster than its preceding methods. It achieves much of its speed by the constant lookup time of the positions table [25]. In spite of speed improvement, FASTA is not suitable for large scale alignment. It is still slow and needs a large amount of memory for very long sequences [12], [23]. Also in the lookup table, FASTA uses a strict one-to-one mapping function which limits the maximum seed length to the logarithm of the table size [25]. Another deficiency is that FASTA loses accuracy by restricting the alignment to only a strip centred around a highly similar region, while the optimal alignment may extend beyond this strip. Increasing the strip width can reduce the possibility of misaligning, but also reduces the algorithm's speed [9].

3.2.2 BLAST Family

3.2.2.1 BLAST

The Basic Local Alignment Search Tool (BLAST) was proposed in 1990 [7]. BLAST finds regions of local similarity between protein and nucleic acid sequences. It introduced the hit and extend approach, which can be implemented using lookup tables. BLAST is practically faster than FASTA since it seeks only the more significant patterns in the sequences, without sacrificing much accuracy.

The main strategy of BLAST is to search only for the segment pairs with a score above some cutoff. An overview of the BLAST algorithm for DNA sequences is as follows:

1. Identify hits in the reference sequence according to a consecutive seed of length l, with default $l = 11$.
2. Enter the positions of the hits into a hash table with a one-to-one mapping function.
3. Scan the query sequence for hits of length l. Search for the hits in the hash table where the matching hits can be found in constant time.
4. Extend every pair of the matching hits to the left and right without allowing gaps. At the same time, compute the score of the alignment with match and mismatch scores. Stop extending once the score falls a certain amount below the best score yet found.
5. Report the final alignment, if it scores more than a threshold, S_{min} [7].

A simple example of the above steps is given in Figure 3.2. The BLAST time complexity is proportional to the product of the lengths of the reference and query sequences, the same as the preceding algorithms. However, discarding the hits shorter than l significantly reduces the number of hits need to be extended. This feature makes the algorithm run faster than its preceding methods.

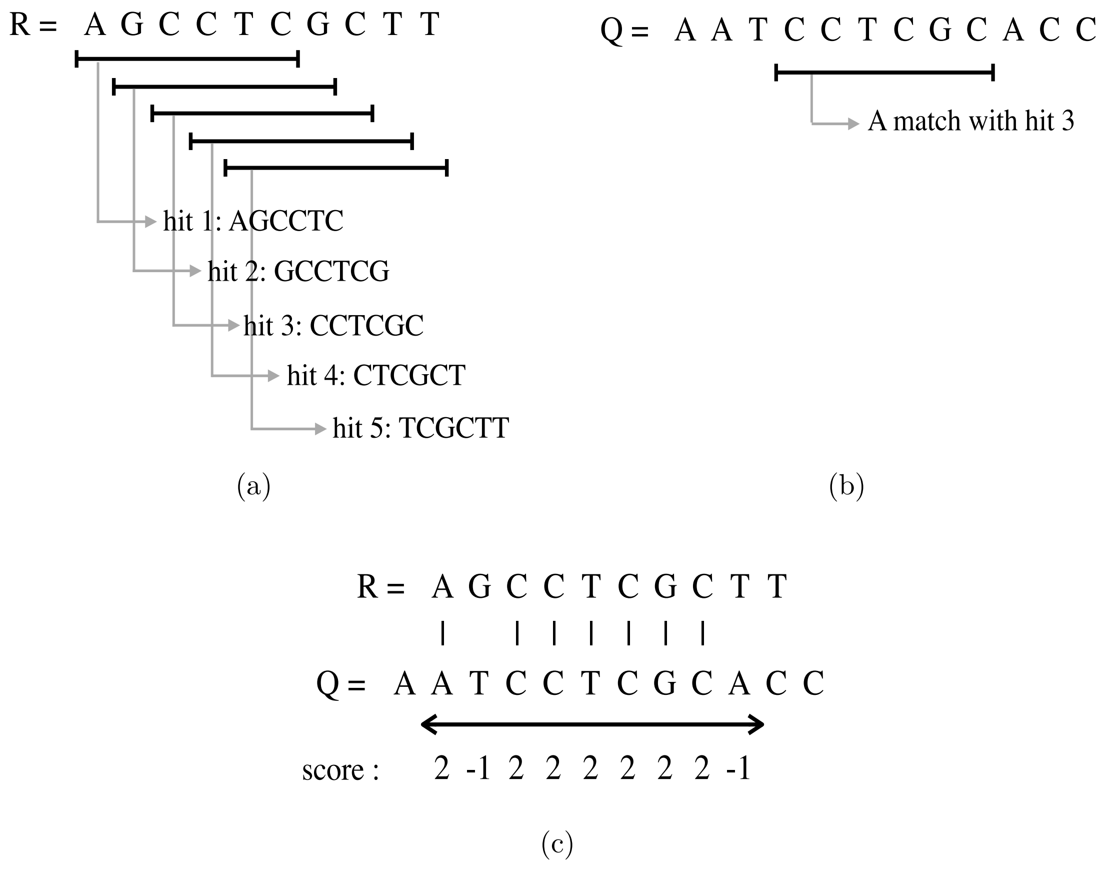


Figure 3.2: The BLAST algorithm on the sequences $R =$ AGCCTCGCTT and $Q =$ AATCCTCGCACC, assuming $l = 6$, $s(x, y) = +2$ for matches, and $s(x, y) = -1$ for mismatches. (a) shows the 1st step of the algorithm in which the reference sequence hits are identified. (b) a matching hit pair is found in the query sequence during the 3rd step of the algorithm. (c) indicates the extension process in the step 4. The extension stops in both directions when the score drops by more than 1 unit. The score of the final local alignment equals 12.

3.2.2.2 Gapped BLAST

The original BLAST only generates gapless alignments. Since the more biologically meaningful similarities may contain insertions and deletions, a modification to BLAST, Gapped BLAST, was introduced in 1997 [9]. Gapped BLAST includes two major refinements:

1. The criterion for extending the hits has been modified such that instead of one

pair of hits, two pairs of hits are required for extension. Also, the two pairs of hits must satisfy the following conditions to be extended: i) the two pairs are not overlapping and ii) they are on the same diagonal of the alignment matrix and within a fixed distance of one another.

2. The extension phase has been developed such that it can produce gapped alignments. To reduce the computational cost imposed by the gapped extension, a moderate score, S_g, was introduced. The eligible hits are first extended without gaps. If the resulting alignment score exceeds S_g, then a gapped extension is applied on the hits using a dynamic programming procedure.

These refinements not only give the ability to generate gapped alignments, but also speed up the algorithm 3 times over the original BLAST [9].

3.2.2.3 BLASTZ

BLASTZ [11] is a modified version of the Gapped BLAST designed for comparing two large genome sequences. It also provides additional search options as user-defined parameters. One option is to force the program to report only the matching regions that locate in the same order and orientation in both sequences. Another difference between BLASTZ and Gapped BLAST is that BLASTZ uses a scoring scheme which includes a substitution matrix and a linear cost function for gaps of length k [11].

3.2.2.4 Other versions

There are several other tools derived from BLAST, some designed for specific purposes. For example, MegaBLAST [10], with a consecutive seed of length $l = 28$, is designed for comparing highly similar sequences. It makes the BLAST algorithm faster for closely related sequences, but does not provide proper output for distant similarities and is not efficient for huge sequences [12].

In general, performing large scale alignment on modern genomic data with members of the BLAST family is both slow and space-consuming [13]. Moreover, many alignments are missed by these tools, because of their ungapped and inflexible seed patterns. A short seed generates many random hits and increases the computational cost; on the other hand, a long consecutive seed misses distant similarities and decreases the sensitivity of the program.

3.2.3 PatternHunter

PatternHunter [12] made an advance in similarity searching by introducing the spaced seed model. It designs a sensitive spaced seed for weight $w = 11$ with length $l = 18$. It also supports user-defined or randomly generated seed patterns. In addition to the improved sensitivity, PatternHunter increases the speed with a new hit-processing technique. PatternHunter is 5 to 100 times faster than Blastn[1] on various size genome sequences, considering the same sensitivity degree in both tools [13].

The PatternHunter algorithm follows the hit and extend approach of the BLAST family. The generated hits are entered into a hash table consisting of two arrays. One array has 4^w entries (for explanation, see Section 4.4.1), each entry for a single possible hit. This array stores the position of the first occurrence of every detected hit. The second array with n entries keeps the positions of all the next occurrences.

Similar to Gapped BLAST, PatternHunter first extends the hit pairs without allowing gaps and calculates the scores. Then, the gapped extension is applied only to the segment pairs whose score are above some threshold. For the gapped extension, it uses a different method in which neighbouring high scoring segments and local short

[1]BLAST for nucleotide sequences.

hits are joined together.

The amount of memory used by PatternHunter has been estimated in the article [12]. For two sequences of length n and m and a spaced seed of weight w, PatternHunter requires approximately $(4^{w+1} + m + (5 + \epsilon)n)$ bytes. From this amount, $(4^{w+1} + 4n)$ bytes are for the hash table and $(n + m)$ bytes are for storing the input sequences. The remaining ϵn bytes indicates a variable amount used by other data structures.

According to an experiment conducted on two bacterial genome sequences [12], PatternHunter runs 20 times faster than Blastn, uses $\frac{1}{10}$ of the Blastn memory and also yields better output quality according to the alignments score.

3.2.3.1 PatternHunter II

The idea of spaced seed pattern has been extended into multiple spaced seeds in PatternHunter II [13]. The authors also described a greedy algorithm for optimizing multiple spaced seeds to maximize sensitivity. In fact, the problem of computing multiple optimal spaced seeds is NP-hard [13], but the proposed greedy method gives a near optimal solution. Various studies have focused on design and evaluation of spaced seeds [16], [26]–[33] including several software programs for computation of sensitive sets [15], [34]–[37].

PatternHunter II designs a set of 16 spaced seeds of weight $w = 11$ with length $l \leq 21$. A separate hash table is build for each one of the seeds. Therefore, the memory consumption for the hash tables is $(4^{w+1} + 4n)k$ bytes, in which k denotes the number of seeds. Since the memory consumption in PatternHunter tools is exponentially dependent on the seed weight, the use of longer and heavier seeds is very expensive and inefficient.

PatternHunter II with $k = 2$, 4 and 8 runs over 1000 times faster than the Smith and Waterman algorithm, while it approaches the same sensitivity[2]. It has also been shown that PatternHunter II with $k = 2$ improves Blastn sensitivity about 3%, while running twice as fast [13].

3.2.4 YASS

Yet Another Similarity Searcher (YASS) was proposed in 2005 [8], [19]. YASS is a DNA local alignment tool based on the hit and extend approach. Two improvements were introduced by this tool: i) the transition-constrained seed model which increases the sensitivity to biologically meaningful similarities and ii) a new hit criterion for detecting groups of hits that are potential high similarities.

The first step of YASS algorithm is similar to the other hit and extend approaches. In its default mode, YASS exploits a single transition-constrained seed of weight $w = 9$ to detect hits and constructs a lookup table to store them. Assume $R[i_1 \dots i_1 + k_1 - 1]$ and $Q[j_1 \dots j_1 + k_1 - 1]$ form a hit pair detected between the reference and query sequences, and $R[i_2 \dots i_2 + k_2 - 1]$ and $Q[j_2 \dots j_2 + k_2 - 1]$ form another pair. To determine whether these two pairs are eligible for the next step or not, the two following conditions must be verified:

1. The inter-hit distance is less than some predefined value, that is, the two hits are close to each other in both sequences. This condition can be formulated as follows:
$$\max(|i_2 - i_1|, |j_2 - j_1|) < \rho \tag{3.3}$$
2. The diagonal distance of the two pairs is less than a predefined bound. In other words, the two hit pairs occur at neighbouring diagonals of the alignment matrix.

[2]The Smith and Waterman algorithm finds all the solutions and has 100% sensitivity.

This condition can be expressed as follows[3]:

$$|(i_2 - i_1) - (j_2 - j_1)| < \delta \tag{3.4}$$

Any two pairs of hits satisfying the above conditions are grouped together and ready to be extended. To avoid groups of strongly overlapping hits or groups with just a single hit, a filtering criterion, called group size, is defined. *Group size* is the minimum number of matching individual letters in all the hits of a group. With the group size parameter, YASS can make a balance between the sensitivity and the number of useless extensions, since the extension is invoked only for groups of sufficient size [8], [19].

In an experiment [8], YASS performance has been compared with a BLAST tool[4] according to the execution time and the number of exclusive similarities[5]. It indicates that YASS runs relatively faster than BLAST, while computing more exclusive similarities, and so is more sensitive.

YASS provides a set of transition-constrained seeds and also supports user-defined seed patterns. Therefore, the user can either select the seeds from the predefined set or import arbitrary seed patterns designed for specific purposes. However, designing transition-constrained seeds with good performance remains an expensive and challenging problem.

[3]YASS performs some statistical analysis on the input sequences for computing the ρ and δ parameters.

[4]NCBI BLAST 2.2.6

[5]The similarities that have been detected exclusively by one of the programs.

Chapter 4

Methodology

In this chapter, the program we developed for finding approximate repeats in DNA sequences is described. This program takes as input two DNA sequences, a reference and a query, and a set of spaced seeds, and outputs a list of approximate repeat pairs between the two sequences along with their length and score. This list includes all the pairs of approximate repeats that share a pattern according to the given seeds, with a minimum length and an edit distance no more than some threshold.

As mentioned before, our program follows the hit and extend approach. The hits are detected according to the given set of spaced seeds and stored in a memory-efficient hash table. Then, the matching pairs of hits between the reference and query are extended into longer repeat pairs. We used the source code of the E-MEM software [18] as a framework to implement our program in C++. The source code of our program is freely available at `http://www.cas.mcmaster.ca/~bill/approxrepeats/`. In Section 4.1, we mention the major modifications made to the E-MEM program. Then, we provide an outline for the algorithm of our method in Section 4.2. We give a more detailed description of each step in subsequent sections.

4.1 E-MEM Modifications

E-MEM computes maximal exact matches[1] rather than approximate repeats, so it works with a single consecutive seed. This allows E-MEM to not index all the positions of the reference sequence into the hash table. In fact, it is sufficient for E-MEM to index only every l positions of the reference, assuming l to be the seed length. The reason behind this reduction is as follows. For any maximal exact match of length L_{min} or more between the reference and query sequences, there must be a hit of length l in the reference sequence that starts at a position that is a multiple of $L_{min} - l + 1$ and is completely contained in the maximal exact match. Therefore, any maximal exact match can be detected by one of the hits starting at multiples of $L_{min} - l + 1$ in the reference sequence and it is sufficient if only such hits are indexed into the hash table [18].

The same argument cannot be applied to the case of spaced seeds and approximate repeats. For any approximate repeat, there must be a hit obtained from a spaced seed that is completely contained in the repeat, but the spaced seed may fit at any position along the repeat. In other words, the hit may start at any position of the reference sequence. Therefore, all the positions of the reference sequence need to be indexed into the hash table. This leads to a major difference in the hash table size and the amount of required memory for storing the hash table. Let n denote the reference sequence length. E-MEM indexes n/l positions of the reference sequence into the hash table, while in our program, all the n positions are indexed into the table. Therefore, we redesigned the E-MEM hash table to make it space-efficient for this increase in the amount of data. Moreover, our program indexes each position of the reference

[1]A maximal exact match between two sequences is a match, or an exact repeat pair, that cannot be extended on either side [18].

sequence k times, assuming k to be the number of spaced seeds. For this reason, we arranged k hash tables, one dedicated to each seed.

Another major modification we made to the E-MEM program is in the extension phase. The extension process in E-MEM stops whenever a mismatch between the two sequences is visited. However, in extending an approximate repeat pair, both mismatches and gaps are allowed. This makes the extension phase of our program more complicated and expensive than the E-MEM extension process. Next, the algorithm outline of the program is presented. In the rest of this document, n_1 denotes the reference sequence length and n_2 denotes the query sequence length.

4.2 The Algorithm Outline

// Initialization

1. Load the input sequences, R and Q, and a set of k spaced seeds.

   // Index the reference sequence and build k hash tables.

2. Scan the reference sequence and align every position, $p = 1 \dots n_1$, with all the seeds, s_i $(i = 1 \dots k)$, to make k hits. For each hit h_{ip}:

   2.1. Compute a key value KEY_{ip}.

   2.2. Store p in an entry of the i^{th} hash table according to the key value KEY_{ip} (p is the h_{ip} starting position in the reference sequence).

   // Scan the query sequence and extend the eligible hit pairs.

3. Scan the query sequence and align every position, $q = 1 \dots n_2$, with all the seeds, s_i $(i = 1 \dots k)$, to make k hits. For each hit h_{iq}:

   3.1. Compute a key value KEY_{iq}.

```
    3.2. Lookup the key value $KEY_{iq}$ in the $i^{th}$ hash table.
    3.3. If $KEY_{iq}$ is found, for all the positions recorded for this key in the hash table,
         $\{p \mid p$ is stored in the location of $KEY_{iq}\}$ report $(h_{ip}, h_{iq})$ as a matching
         pair. For each matching pair of hits:
        3.3.1. Extend the hit pair without allowing gaps until the score drops by a
               certain amount, $X_{drop}$.
        3.3.2. If the score of the resulting repeat pair is above some threshold $S_{min}$,
               perform gapped extension on the initial hit pair $(h_{ip}, h_{iq})$ using dynamic
               programming.
        3.3.3. Report both the left and right positions of the extended pair in the
               referenc sequence $(p_l, p_r)$ and in the query sequence $(q_l, q_r)$.
        3.3.4. If the extended repeat pair is shorter than $L_{min}$, ignore the pair.
        3.3.5. If the edit distance between the repeat copies is more than $D_{max}$, ignore
               the pair.
        3.3.6. If the repeat pair is completely contained in some other recently detected
               repeat, ignore the pair.
        3.3.7. Store the repeat pair positions, $(p_l, p_r)$ and $(q_l, q_r)$, in some temporary
               memory.
    // Remove redundant pairs and report the output.
4. Sort the repeat pairs based on the starting position in the query sequence.
5. Remove successive pairs which are completely contained in each other.
6. Output the remaining repeat pairs.
```

```
>gi|685508232|ref|NC_007121.6| Danio rerio strain Tuebingen chromosome 10, GRCz10
TCAGGATGATGATGCGGCGACACACACAGCAGGACTGGAGACAGAGCGCTCTGATGAACATGACAGTGAA
GGTGAACATCTCCCATGGCCTTCACAGTACCCAGATCTAAACATTATTGAGCACTGTGGGCTGTTTTAGG
GGAGCGAGTCAGGAGAGGTTTTCCTCCAGCAGCATCAGCAGTGACCTGAACACTATTCTGAAGAAGAACA
GCTCACAACCCCTCTGACCACTGTGCAGGACTCCTGTCTGTCAATCACTACACTGAGGAGGAGGAGCTAC
AGCACTGACGCCCTATTCACACGGGGCGTCAGCGTCAACGCTTCCCATTCACTTTGAATGGGT
```

Figure 4.1: A small chunk of zebrafish chromosome 10 in Fasta format.

4.3 Initialization

The input sequences should be provided in two *Fasta* format files. Fasta format is a text file format which represents biological sequences with single letters. A Fasta file may contain a single or multiple sequences. Each sequence in a Fasta file begins with a description line, followed by lines of sequence data. The description line is started with a ">" character to be distinguished from the data lines. The data lines usually do not exceed 80 characters. Figure 4.1 shows an example of a sequence in Fasta format[2].

For simplicity in this document, we refer to all sequences in the reference input file as the reference sequence, R, and all sequences in the query input file as the query sequence, Q. The k spaced seeds must be given in a regular text file in which the spaced seeds are separated with newline. As mentioned in Section 2.1.1 a DNA sequence is represented as a string over the 4-letter alphabet of $\Sigma = \{A, C, G, T\}$. In addition to these 4 letters, a sequenced DNA molecule may contain unknown bases which are denoted by N in Fasta format. These unknown bases must be either disregarded from computation (as explained in Section 4.8.1) or replaced with known bases. In the latter case, since an N symbol stands for any of the 4 alphabet letters, it is replaced with a randomly selected one. By this replacement, a DNA sequence consists of only 4

[2]Obtained from `http://www.ncbi.nlm.nih.gov/`

different letters and each letter is encoded using 2 bits, as shown below.

$$A \rightarrow 00, \quad C \rightarrow 01, \quad G \rightarrow 10, \quad T \rightarrow 11$$

In order to store an input sequence, we create an array of 64-bit integers and encode every 32 letters of the sequence into a 64-bit integer. The whole sequence is stored as an array of integers by packing every 4 letters into 1 byte. As a result, $n/4$ bytes of memory is required for storing a sequence of length n. Figure 4.2 shows a small example.

$$\text{GCTA} \rightarrow (10{,}01{,}11{,}00)_{binary} = (156)_{decimal}$$

Figure 4.2: 4 letters are encoded into 1 byte.

4.4 Hit Identification

Hit identification has two steps: 1) indexing the reference sequence 2) scanning the query sequence. In this section, we explain how the matching hit pairs are identified between the reference and query sequences in these two steps.

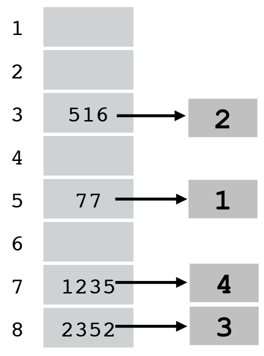
4.4.1 Indexing the Reference Sequence

In this step, all the positions of the reference sequence are indexed into the hash table. We start from the first position and move along the reference sequence until the end. For each position, $p = 1 \ldots n_1$, all the k seeds, s_i $(i = 1 \ldots k)$, are aligned one by one with the substring starting at p to obtain a hit. A key value is computed for the hit of seed s_i at position p, denoted by KEY_{ip}. In order to store the hit position in a hash table, there must be a hash function h which maps a key value into a hash table entry (see Section 4.5). Using this hash function, we compute a hash index for the key value,

p	:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
R	:	A	A	G	C	A	A	T	C	A	A	T	C	A	A	T	C	A	G	T	C
s_1	:	1	1	$*$	1	1	$*$	1	1												

p	hit_{1p}	KEY_{1p}	$h(KEY_{1p})$
1	AAGCAATC	$(00{,}00{,}01{,}00{,}11{,}01)_b = 77$	5
2	AGCAATCA	$(00{,}10{,}00{,}00{,}01{,}00)_b = 516$	3
3	GCAATCAA	$(10{,}01{,}00{,}11{,}00{,}00)_b = 2352$	8
4	CAATCAAT	$(01{,}00{,}11{,}01{,}00{,}11)_b = 1235$	7

(a)



(b)

Figure 4.3: Indexing the reference sequence R using the spaced seed s_1 and building the corresponding hash table. (a) s_1 is aligned with the reference sequence R at the position $p = 1$. The table provides the hit, the key value, and the hash index computed for $p = 1 \dots 4$. (b) The positions $p = 1 \dots 4$ are stored in the hash table according to the hash indices given in the last column of the left table. For example, $p = 1$ is stored in the entry indexed by $h(KEY_{11}) = h(77) = 5$.

$h(KEY_{ip})$, and store p into the entry indexed by $h(KEY_{ip})$ in the i^{th} hash table. An example is provided in Figure 4.3 to show this process.

To compute a key value for the hit of seed s_i at position p, first the letters of the hit that are aligning with $*$'s of the seed s_i are removed. Then, the 2-bit code of the remaining letters is taken as the key value. Figure 4.4 gives an example of computing key values. Assuming w_i to be the weight of seed s_i, the key values obtained from s_i are integers of $2w_i$ bits and within the interval $[0, 2^{2w_i}) = [0, 4^{w_i})$. Thus, the number of all possible key values is 4^{w_i} for the i^{th} hash table[3]. The key values are stored in 64-bit integers. In order for 4^{w_i} to fit in a 64-bit integer, the seed weight w_i must not exceed 32 ($w_i \leq 32$).

Following this approach, if the seed s_i is aligned with a different position p' and produces the same key value as it does for p ($KEY_{ip} = KEY_{ip'}$), then it is concluded

[3]If there is only a single spaced seed of weight w, the number of all possible key values is 4^w. Using a one-to-one hash function (e.g. $h(x) = x$), the hash table must provide 4^w different entries, each for a different key value. This explains the 4^w factor in the memory consumption of PatternHunter [12], [13] in Section 3.2.3

$$\begin{array}{lcccccccccc}
R & : & \mathrm{A} & \mathrm{A} & \mathbf{G} & \mathrm{C} & \mathrm{A} & \mathbf{A} & \mathrm{T} & \mathrm{C} \\
s_1 & : & 1 & 1 & * & 1 & 1 & * & 1 & 1 \\
 & & \mathrm{A} & \mathrm{A} & & \mathrm{C} & \mathrm{A} & & \mathrm{T} & \mathrm{C} \\
\text{2-bit} & : & 00 & 00 & & 01 & 00 & & 11 & 01 \\
\hline
\end{array}$$

$$KEY_{11} : \quad 00\ 00\ 01\ 00\ 11\ 01 \quad = (77)_{decimal}$$

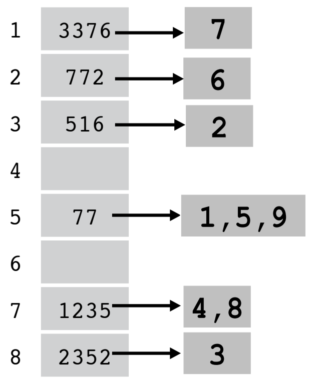
Figure 4.4: Computing the key value for the hit h_{11} of the Figure 4.3a.

that the two hits starting at positions p and p' are matching hits. Obviously, for the function h, if $KEY = KEY'$, then $h(KEY) = h(KEY')$. Since the key values of two matching hits are equal, the corresponding hash indexes are also equal. As a result, p and p' are stored in the same entry of the hash table in the form of a list. See Figure 4.5 for an example. As we move forward, some other positions may also be added to this list (e.g. $p = 9$ in Figure 4.5). These positions, which are stored in a same entry of the hash table, represent matching hits in the reference sequence, i.e, the occurrences of a same pattern in the reference sequence.

p	:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
R	:	A	A	G	C	A	A	T	C	A	A	T	C	A	A	T	C	A	G	T	C
s_1	:					1	1	$*$	1	1	$*$	1	1								

p	hit_{1p}	KEY_{1p}	$h(KEY_{1p})$
5	AATCAATC	$(00{,}00{,}01{,}00{,}11{,}01)_b = 77$	5
6	ATCAATCA	$(00{,}11{,}00{,}00{,}01{,}00)_b = 772$	2
7	TCAATCAA	$(11{,}01{,}00{,}11{,}00{,}00)_b = 3376$	1
8	CAATCAAT	$(01{,}00{,}11{,}01{,}00{,}11)_b = 1235$	7
9	AATCAATC	$(00{,}00{,}01{,}00{,}11{,}01)_b = 77$	5

(a)



(b)

Figure 4.5: Continue indexing the reference sequence R using the spaced seed s_1. (a) The table provides the hit, the key value, and the hash index computed for $p = 5 \ldots 9$. (b) The positions $p = 5 \ldots 9$ are added to the hash table according to the hash indices. Since $KEY_{11} = KEY_{15} = KEY_{19} = 77$, the positions $p = 5$, and $p = 9$ are stored in the same entry of the hash table as $p = 1$ stored. Similarly, $p = 4$ and $p = 8$ are hashed to the same entry as the positions of two matching hits.

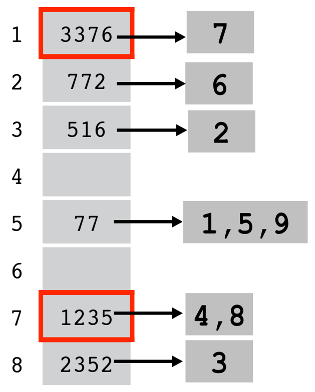
4.4.2 Scanning the Query Sequence

This step begins after indexing the whole reference sequence has finished. Like the previous step, we start from the first position of the query sequence and move along it until the end. For each position, $q = 1 \ldots n_2$, all the k seeds, s_i ($i = 1 \ldots k$), are aligned one by one with the substring starting at q to obtain a hit. Then, a key value is computed for every hit using the same approach as used for the reference sequence. This way, the matching hits between the reference and query sequences will have equal key values. For the hit of seed s_i at position q, $h(KEY_{iq})$ is computed to find the positions of the reference sequence in which the matching hits occur. Then, the entry of the i^{th} hash table indexed by $h(KEY_{iq})$ is looked up for the matches. In that location, there is a list containing the positions of all the reference hits whose key values equal KEY_{iq}. For every p in the list, (p, q) is reported as a matching hit pair between the reference and query sequences as shown in Figure 4.6.

q :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Q :	T	C	A	G	T	C	C	A	T	T	A	A	G	G	G	A	C	T	G	T
s_1 :		1	1	$*$	1	1	$*$	1	1											

q	hit_{1q}	KEY_{1q}	$h(KEY_{1q})$
1	TCAGTCCA	$(11,01,10,11,01,00)_b = 3508$	not found
2	CAGTCCAT	$(01,00,11,01,00,11)_b = 1235$	7
3	AGTCCATT	$(00,10,01,01,11,11)_b = 607$	not found
4	GTCCATTA	$(10,11,01,00,11,00)_b = 2892$	not found
5	TCCATTAA	$(11,01,00,11,00,00)_b = 3376$	1

(a)



(b)

Figure 4.6: Scanning the query sequence Q using the spaced seed s_1 and looking up the hash table of Figure 4.5b. (a) s_1 is aligned with the query sequence Q at the position $q = 2$. The table provides the hit, the key value, and the hash index computed for $q = 1 \dots 5$. (b) KEY_{12} is found at index 7 of the hash table colored in red. Therefore, the position $q = 2$ is reported with all the positions stored at index 7 in the form of $(4, 2)$ and $(8, 2)$. Similarly for $q = 5$, the pair $(7, 5)$ is reported as a matching hit pair between R and Q.

4.5 Hash Tables

In this section, we first give a brief description of hash tables and address the issues related to hashing techniques. We proceed by proposing our new hash table design and discussing how hashing issues are handled in our program.

A hash table is a data structure that allows the storage and retrieval of data in an average constant time. This data structure consists of a table T where the data to be searched is stored, and a hash function h which maps each element of data into a table index. The element is then stored in the table entry indicated by the assigned index. Every element of data must be represented with a unique key value. This key value is the input of the hash function to assign an index to the element. As a result, the hash function is a function on the set of all possible key values to the set of all table indices. In the next two sections, we discuss the hashing issues.

4.5.1 The Hash Table Size

When the number of all possible key values is small, the hash table size can be large enough to accommodate all possible keys. In this case, the hash function can simply be a one-to-one mapping function. However, when the set of all possible key values is large, it is not space-efficient to set the hash table size as large as this set. In this case, a non-injective function is used as the hash function. As discussed before, in our program the number of all possible key values is 4^w for a single hash table related to a seed of weight w. The exponential dependence of this number on the seed weight, makes the use of a one-to-one hash function (as in PatternHunter [12], [13]) inefficient, specially for long and heavy seeds.

4.5.2 Hashing Collisions

When using a non-injective hash function, there is a situation in which the two key values are not equal, but the hash indices computed by the hash function are equal as shown below.

$$KEY \neq KEY' \text{ , } while \ h(KEY) = h(KEY')$$

This situation is called a hashing collision. Figure 4.7 provides an example. Since two elements with different key values cannot be stored in one entry of the table, a hash table needs a strategy for resolving collisions. A well-designed hash function can reduce the number of collisions, but according to the pigeon hole principle, there may still be some collisions for which a resolution technique is required.

Two of the most commonly used collision resolution techniques are *chaining* and *open addressing*. In chaining, a chain is created out of all the colliding elements at one entry. A pointer to the chain is stored in the table entry. In open addressing, when a

$$h(KEY) = KEY \bmod 12;$$
$$KEY = 13 \rightarrow h(KEY) = (13 \bmod 12) = 1$$
$$KEY' = 25 \rightarrow h(KEY') = (25 \bmod 12) = 1$$

Figure 4.7: An example of hash collision.

new element has to be inserted into an occupied location of the table, a series of table entries are examined in some order defined by the hash function, until an unoccupied one is found. Then, the new element is stored in the first found unoccupied entry of the table [38].

4.5.3 The Program Hash Table Structure

The hash table structure of our program is described as follows. As mentioned before, k hash tables are designed for the k seeds. In each hash table, we use two levels of hashing. The first level is essentially the same as hashing with chaining, but instead of making a chain out of the elements hashed to an entry j, a secondary hash table T_j is used. This is shown in Figure 4.8. The first-level hash table is called the main table and denoted by T. m denotes the main table size. The real data is stored in the secondary tables, while the main table entries keep only the address of the secondary tables. Notice that the key values are not stored in the hash table, because at any time they can be retrieved given the seed number, the hit position and the reference sequence. This design gives a saving in memory consumption. Once an element is hashed as the first element to the entry j of the main table, the secondary table T_j is created in the entry j with an initial size $m_j > 1$. The collisions at the first level are resolved by the secondary tables. The secondary tables employ open addressing (see Section 4.5.2) as the collision resolution technique. The purpose of the main table is to divide the elements into smaller groups which are stored as the secondary tables. Then, the collisions are resolved easier and faster in these small size tables.

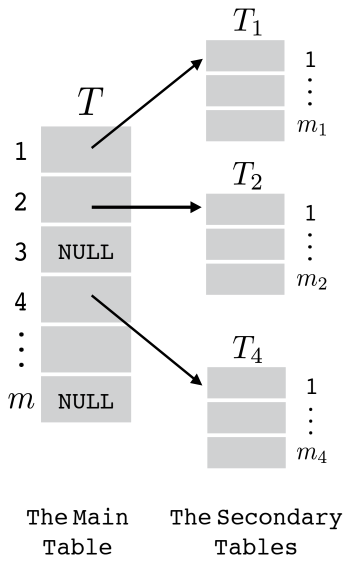


Figure 4.8: An example of the hash table structure of our program. The entries $j = 1, 2$, and 4 of the main table keep pointer to the secondary tables. No element has been hashed so far to the entries with a $NULL$ value.

4.5.3.1 The Hash Functions

The two levels of hashing use different hash functions. For the main table, we choose a division method hash function. A division hash function is in the following form:

$$h(key) = key \bmod m \tag{4.1}$$

It takes the reminder of the key value divided by the table size, m, to map the key value into a table index. To reduce the number of collisions for a division hash function, certain values of m that provide a fair distribution of the elements among the table entries are desired. A good choice for m is a prime number not too close to an exact power of 2 [38].

In the secondary tables, a hash function relevant to open addressing must be used. Double hashing is a method of producing a series of hash indices for open addressing and has the following general form:

$$h(key, c) = [h_1(key) + c\ h_2(key)] \bmod m \tag{4.2}$$

in which h_1 and h_2 are two auxiliary hash functions, c is a step counter starting from 1, and m is the hash table size. If the hash index generated for $c = 1$ indicates an occupied entry of the hash table, the c value is incremented by 1 and a new hash index is computed using $c = 2$. This process continues until an unoccupied entry is visited. Therefore, this hash function produces a series of hash indices to be examined sequentially in the case of a collision. In the above formula, m denotes the hash table size which equals m_j for the secondary tables.

To ensure that the entire hash table is searched by the generated series of hash indices in double hashing, the value $h_2(key)$ must be relatively prime to the hash table size m_j. One approach to satisfy this condition is to set m_j to be a power of 2 and to design h_2 such that it always produces an odd number [38]. We choose h_1 to be a simple division hash function, and h_2 to be a division hash function combined with a bit-OR operation as follows:

$$h_1(key) = key \bmod m_j \tag{4.3}$$

$$h_2(key) = (key \bmod m'_j) \mid 1 \tag{4.4}$$

where m'_j is the largest power of 2 less than m_j. The bit-OR operation in Equation (4.4), is an OR operation between 1 and the least significant bit of (key mod m'_j) value. It does not change odd values of (key mod m'_j), but converts even values of (key mod m'_j) to odd numbers.

4.5.3.2 The Hash Table Size

As discussed in the beginning of this chapter, n_1 elements are hashed into one hash table[4]. Here, we define the size of the main and secondary tables such that one hash table can accommodate n_1 elements. We set m to be the smallest prime number greater than n_1/k and m_j to be the closest power of 2 to the number k. Therefore, a hash table is capable of having n_1/k secondary tables, each table with initially about k entries.

Since the number of elements hashing to each secondary table is unknown, there is always a chance for secondary tables to overflow. We say that a secondary table T_j of size m_j has overflowed, if for a new element hashed to this table, an empty location is not found after running $c = m_j$ steps in the hash function. In case a secondary table T_j overflows, the table size m_j is doubled. Then, all the elements in T_j are rehashed considering the new table size as shown in Figure 4.9.

In order to keep track of the secondary table sizes, a column of one-byte entries is added to the main table. This column is initially filled with zeros. An entry of the column, called the size counter, is incremented by 1 each time its corresponding secondary table size is doubled. Figure 4.10 shows an example. A one-byte memory slot can represent the numbers from 0 to 255. This means that a size counter can indicate up to 255 times doubling of a table size which is more than adequate for our needs.

[4]Notice that the key values of these n_1 elements are not necessarily distinct, so some of them will be stored together in one entry of a secondary table as a list. These elements indicate the positions of matching hits (see Section 4.4.1).

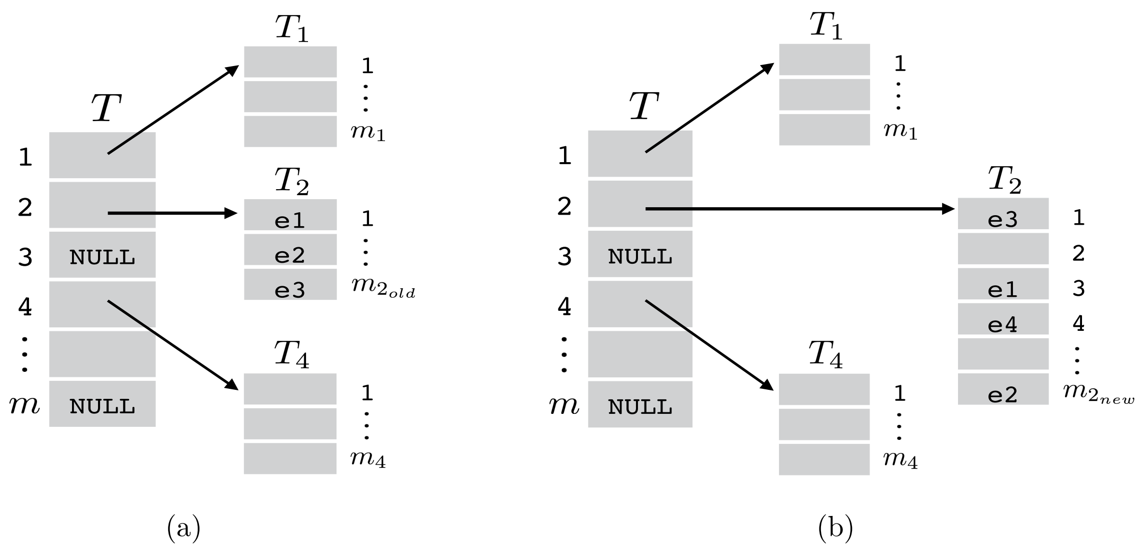


Figure 4.9: Resizing of the secondary tables (a) All the entries of T_2 are occupied (b) To add another element to T_2, m_2 is doubled ($m_{2_{new}} = 2 \times m_{2_{old}}$). Then, all the element of T_2 are rehashed using the new size $m_{2_{new}}$ in the hash function

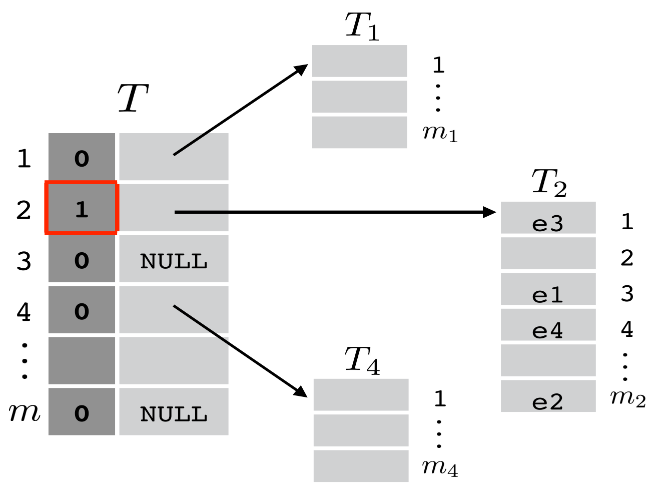


Figure 4.10: The darker column of the main table contains the size counters. The size counter corresponding to the table T_2 is 1. This means that the table size has been doubled once, so its current size is $2 \times m_{2_{initial}}$.

4.5.3.3 The Main Table Entries

This section answers the question of how big an entry of the main table is in bytes. In a 64-bit processor, the size of an address is 8 bytes. Therefore, we expect each T entry to take 9 bytes, 1 byte for the size counter and 8 bytes for the address of a secondary table. However, in practice, the number of bytes depends on the table implementation. One way is to implement a T entry as a structure or a class object containing two data members and make an array out of the structure or the class instances. Using this approach, a T entry would take 16 bytes of memory. The reason for the extra 7 bytes is *data structure padding* applied to such a structure to avoid decrease in the system's performance. A computer reads from or writes to memory in word-sized chunks. With a 8-byte computer word, the data to be read should be at a memory offset equal to some multiple of 8. Since this is not the case in the structure of a T entry, 7 padding bytes are inserted between the two pieces of data to align the next data to the 8^{th} byte. This is shown in Figure 4.11.

The solution to avoid the padding bytes is to implement the main table T as two separate arrays, T_{size} and $T_{address}$, each with only a single data member as in Figure 4.12. This way, a main table entry takes 9 bytes as it was expected.

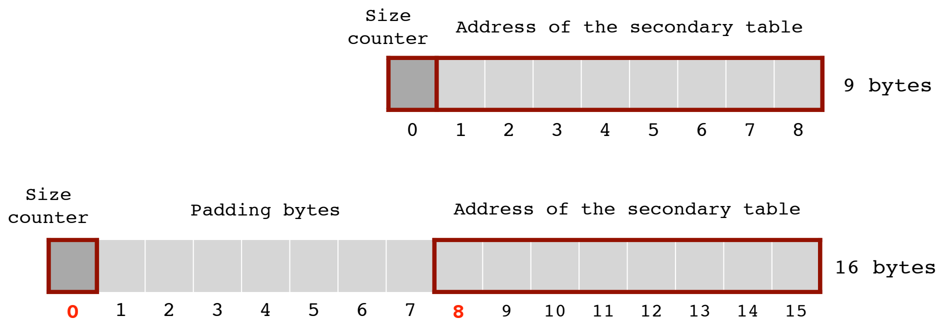


Figure 4.11: One entry of the main table. The top figure shows the two data members without considering padding bytes. In the bottom figure, 7 padding bytes are inserted after the size counter to make the address start at position 8.

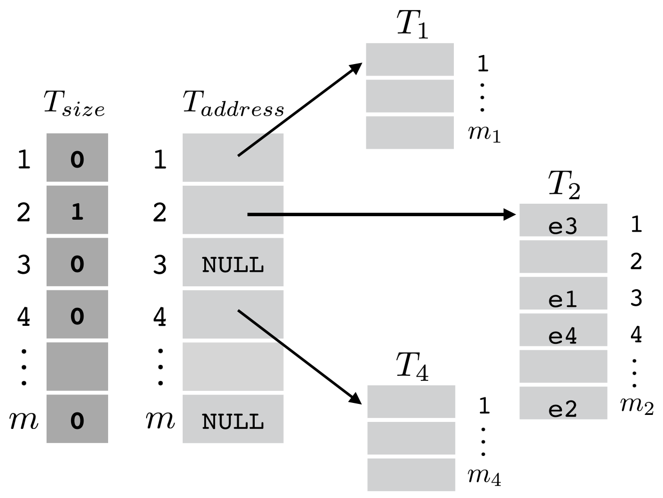


Figure 4.12: The main table T is split into two tables to avoid padding bytes. T_{size} is for the size counters and $T_{address}$ keeps the address of the secondary tables.

4.5.3.4 The Secondary Table Entries

Entries in the secondary tables contain the real data which means the lists of the matching hits positions. Different data structures can be used for implementing these lists, such as linked lists, vectors, and arrays. When indexing the reference sequence and constructing the hash table, the lists extend as we move forward on the reference. Although the size of a list is not known in advance, it is known that if all the lists of a hash table are merged together, we get a list containing all the integers of the range $[1, n_1]$. This is because all the positions of the reference sequence from 1 to n_1 are indexed once for every seed and stored in the corresponding hash table. This property leads to a space-efficient implementation which is also used in PatternHunter [12], [17]. Below, we describe an approach similar to the one of PatternHunter.

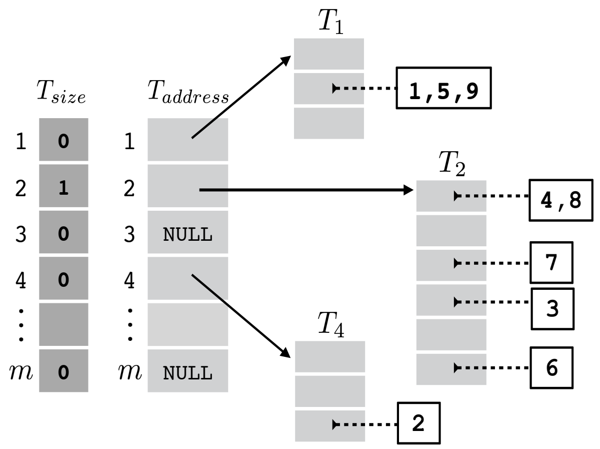
For each hash table, we create an array, namely *prev*, of length $n_1 + 1$. The tail of each list in stored in the secondary table entry. All the previous element of a list are stored in *prev* using the following rules:

1. *prev*$[i]$ is initially set to zero for $0 \leq i \leq n_1$.

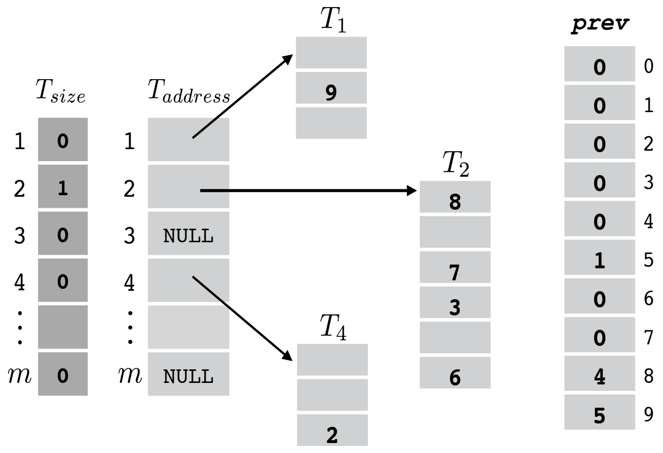
2. *prev*$[i]$ keeps the previous element of the value i for $1 \leq i \leq n_1$.

Since the value i appears only once among all the lists, there is no conflict between the elements of the *prev* array. An example of this implementation is given in Figure 4.13. Finally, having the tail of each list, we can retrieve all the other elements from tail to head using the second rule.

By this implementation, every hash table has a supplementary array of length $n_1 + 1$. Each element of this array and each entry of a secondary table is an integer of the range $[1, n_1]$.



(a) The given lists in white rectangles are supposed to be stored into the secondary tables. Each list is connected to its intended destination by a dashed line.



(b) During the process, at each time, the tail of a list is stored in the secondary table entry. For example, at first the list 1, 5, 9 has only one element, 1, which is stored in the secondary table entry. Then, 5 is added to the list. Since 5 is the list tail, it is stored in the secondary table entry and 1 is stored in *prev*[5]. Next, 9 is added as the tail and takes the place of 5. 5 is stored in *prev*[9].

Figure 4.13: Storing lists into the secondary table entries.

4.5.3.5 Memory Requirement

In this section, we give an estimate for the amount of memory required by a single hash table and k hash tables, considering a 64-bit processor. We use 32-bit integers to represent the numbers of the range $[1, n_1]$. Therefore, an entry of the secondary tables or the *prev* array takes 4 bytes of memory. In a single hash table, we have approximately n_1/k secondary tables, each with about k entries. Therefore, the amount of memory required for the secondary hash tables is approximately $4 \times k \times n_1/k = 4n_1$ bytes. The supplementary array *prev* uses exactly $4(n_1+1)$ bytes which can be rounded to $4n_1$ bytes. The main table has about n_1/k entries of size 9 bytes as discussed in Section 4.5.3.3. In total, a hash table requires about $4n_1 + 4n_1 + 9n_1/k = 8n_1 + 9n_1/k$ bytes. Therefore, the amount of required memory for the k hash tables is approximated by $8n_1k + 9n_1 = (8k + 9)n_1$ bytes.

4.6 Hit Extension

Referring back to the algorithm outline in Section 4.2, so far we have explained the steps 1 to 3.3 which are mainly about the hit identification phase. In the following sections, we continue the outline from step 3.3.1 and discuss the extension phase of the algorithm.

4.6.1 Gapless Extension

Once a pair of matching hits (h_{ip}, h_{iq}) is detected during the scan of the query sequence, it is reported to the extension phase. The extension phase in our program is similar to that in Gapped BLAST [9]. Extending a hit pair without considering gaps is much faster and less expensive than a gapped extension process. Hence, first, a gapless

```
Input: $R$, $Q$; hits starting positions $(p_0, q_0)$; hit length $l$; allowed drop $X_{drop}$;
match score $S_{match}$; mismatch score $S_{mismatch}$; minimum score $S_{min}$.
// Evaluate the score of the hit pair
1: set $p \leftarrow p_0$; $q \leftarrow q_0$; $S_{hit} \leftarrow 0$
2: for $k = 1$ to $l$ do
3:     if $R[p] = Q[q]$ then
4:         $S_{hit} \leftarrow S_{hit} + S_{match}$
5:     else
6:         $S_{hit} \leftarrow S_{hit} + S_{mismatch}$
7:     end if
8:     $p \leftarrow p + 1$; $q \leftarrow q + 1$
9: end for
```

Figure 4.14: The gapless extension algorithm (part 1).

extension procedure is applied to the pair. If the score of the resulting extended pair is not less than S_{min},[5] the original hit pair is passed to the next step for a gapped extension process. For both the gapped and gapless extensions, we follow the so-called *X-drop* method, used in BLAST and many other programs [7], [9], [10]. In this approach, we extend the hit pair in each of the two directions and stop once the score drops by X_{drop} below the so far maximum score. Figures 4.14 and 4.15 provide the gapless extension procedure in pseudocode. First, in Figure 4.14 we evaluate the score of the given hit pair as the center of the later-extended repeat. Next, in Figure 4.15 the hit pair is extended to the right (lines 10 to 19) and then to the left (lines 20 to 30). Finally, the scores of the central segment, the right and the left extensions are added together to be compared with S_{min} (lines 31 and 32).

[5]The minimum score, S_{min}, in our program is equivalent to the moderate score, S_g, of Gapped BLAST. We have employed the method described in the Gapped BLAST paper to compute a reliable value for S_{min}. In this method, the letter frequency of the input sequences is used to choose S_g such that the expected number of invoked extensions is limited [9].

```
    // X-drop algorithm for extending to the right
10: set $S_{rext} \leftarrow 0$; $max \leftarrow 0$
11: while $S_{rext} > max - X_{drop}$ and $p \leq n_1$ and $q \leq n_2$ do
12:     if $R[p] = Q[q]$ then
13:         $S_{rext} \leftarrow S_{rext} + S_{match}$
14:     else
15:         $S_{rext} \leftarrow S_{rext} + S_{mismatch}$
16:     end if
17:     if $S_{rext} > max$ then $max \leftarrow S_{rext}$
18:     $p \leftarrow p + 1$; $q \leftarrow q + 1$
19: end while

    // X-drop algorithm for extending to the left
20: set $p \leftarrow p_0 - 1$; $q \leftarrow q_0 - 1$          ▷ reset the positions to the left of the hits
21: set $S_{lext} \leftarrow 0$; $max \leftarrow 0$
22: while $S_{lext} > max - X_{drop}$ and p $> 0$ and q $> 0$ do
23:     if $R[p] = Q[q]$ then
24:         $S_{lext} \leftarrow S_{lext} + S_{match}$
25:     else
26:         $S_{lext} \leftarrow S_{lext} + S_{mismatch}$
27:     end if
28:     if $S_{lext} > max$ then $max \leftarrow S_{lext}$
29:     $p \leftarrow p - 1$; $q \leftarrow q - 1$
30: end while

    // Report the pair if it scores no less than $S_{min}$
31: set $S_{total} \leftarrow S_{hit} + S_{rext} + S_{lext}$
32: if $S_{total} \geq S_{min}$ then report $(p_0, q_0)$ for the gapped extension
```

Figure 4.15: The gapless extension algorithm (part 2).

4.6.2 Gapped Extension

For the gapped extension, an approach proposed in MegaBLAST [10] is used. This approach is a banded dynamic programming procedure which restricts the search space with X-drop method. The dynamic programming matrix, M, is filled using the following recursion:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + S_{match}, & \text{if } i > 0, j > 0, \text{and } R[i] = Q[j] \\ M(i-1,j-1) + S_{mismatch}, & \text{if } i > 0, j > 0, \text{and } R[i] \neq Q[j] \\ M(i,j-1) + S_{gap}, & \text{if } j > 0 \\ M(i-1,j) + S_{gap}, & \text{if } i > 0 \end{cases} \tag{4.5}$$

The algorithm is implemented such that only three antidiagonals of the matrix need to be kept. Antidiagonal k is the set of all matrix entires with $i + j = k$. Therefore, three lists are sufficient for storing the matrix and finding the optimal extension, i.e., the extension with maximum score. For a given hit pair of length l with start positions (p_0, q_0), we first find the middle of the two hits as follows:

$$p_{mid} = p_0 + \lfloor l/2 \rfloor$$
$$q_{mid} = q_0 + \lfloor l/2 \rfloor$$

Figure 4.16 shows a hit pair between two sample sequences and its middle positions. We construct the matrix M and set (p_{mid}, q_{mid}) as the origin. Then, we extend the hit pair from the middle to the right and fill M along antidiagonals as shown in Figure 4.17. The matrix entries with a score below the X-drop criterion are filled with negative infinity to not be considered in computation further. The extension terminates if either

p :	1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	15	16	17	18	19	20		
R :	A	A	G	C	A	A	T	**C**	A	A	T	C	A	A	T	C	A	G	T	C		
Q :			T	C	A	G	T	**C**	C	A	T	T	A	A	G	G	G	A	C	T	G	T
q :			1	2	3	4	5	**6**	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Figure 4.16: A hit pair at positions $(p_0, q_0) = (4, 2)$ between R and Q. The middle positions (p_{mid}, q_{mid}) are in red.

the right end of one sequence is reached or the last two antidiagonals are all negative infinities. Then, the maximum value of the last non-infinity antidigonal is picked as the optimal score of the right extension, S_r, and its coordinates are marked as (p_r, q_r). The output of this process is the right positions of the extended pair, p_r and q_r, the score S_r, and edit distance of the right extension D_r.

Again, we extend the hit pair in a similar way from the middle to the left and get the left positions, p_l and q_l, the left optimal score S_l, and the left edit distance D_l. See Figure 4.18 for the final extended repeat pair. At the end, the length and the edit distance of the obtained repeat is compared with thresholds as shown in Figure 4.19. The repeat pair is reported if it satisfies the threshold conditions.

	p	8	9
q	0	-3	-i
6	-3	2	
7	-i		

(a) Initially $max = 0$. After filling the 2nd antidiagonal $max = 2$.

	p	8	9	10
q		-3	-i	-i
6	-3	2	-1	
7	-i	-1		
8	-i			

(b) $max = 2$

	p	8	9	10
q			-i	-i
6		2	-1	-i
7	-i	-1	0	
8	-i	-i		

(c) $max = 2$

	p	8	9	10
q				-i
6			-1	-i
7		-1	0	-i
8	-i	-i	1	
9		-i		

(d) $max = 2$

	p	8	9	10	11
q					
6				-i	
7			0	-i	-i
8		-i	1	2	
9		-i	-2		
10		-i			

(e) $max = 2$

	p	8	9	10	11	12
q						
6						
7				-i	-i	-i
8			1	2	-1	
9		-i	-2	-1		
10		-i	-i			

(f) $max = 2$

Figure 4.17: The first steps of the right extension process for the hit pair of Figure 4.16. The matrix origin is $(p_{mid}, q_{mid}) = (8, 6)$. In this example, we set $X_{drop} = 4$, $S_{match} = 2$, $S_{mismatch} = -2$, and $S_{gap} = -3$. The max value is the so far maximum score and is updated each time a new antidiagonal is filled. $-i$ stands for negative infinity and indicates the entries with a value less than $max - X_{drop}$.

```
p :  1  2 |3  4  5  6  7  8  9 10 11 12 13 14 15 16| 17 18 19 20
R :  A  A |G  C  A  A  T  C  A  A  T  C  A  A  T  C|  A  G  T  C
Q :       |T  C  A  G  T  C  C  A  T  T  A  A  G  G|  G  A  C  T  G  T
q :       |1  2  3  4  5  6  7  8  9 10 11 12 13 14| 15 16 17 18 19 20
```

Figure 4.18: The extended repeat pair obtained from applying right and left gapped extensions on the hit pair of Figure 4.16. The output left and right positions are $(p_l, p_r) = (3, 16)$ and $(q_l, q_r) = (1, 14)$.

```
1: if $(D_l + D_r) \leq D_{max}$ and $(p_r - p_l + 1) \geq L_{min}$ and $(q_r - q_l + 1) \geq L_{min}$ then
2:     report $(p_l, p_r)$ and $(q_l, q_r)$ as a repeat pair
3: end if
```

Figure 4.19: The distance and length conditions to determine if an extended repeat pair is eligible to be reported.

4.7 Report the Output

The program outputs a list of approximate repeat pairs between the reference and query sequences. It also reports the score and length of the repeat pairs. Each line of the output represents a single repeat by giving the following information from left to right: the left and right positions in the reference sequence (p_l, p_r), the left and right positions in the query sequence (q_l, q_r), the repeat score, and the repeat size, or length. Since the length of a repeat copy in the reference may differ from its copy in the query, two lengths are reported. The first one refers to the reference copy and the second one is the length of the query copy.

Since there can be multiple query sequences, the repeats are displayed in groups according to their query sequence. Each group starts by a header line which contains a short description of the corresponding query sequence. Inside a group, the repeat are sorted based to the staring positions in the query (q_l). In case there are multiple reference sequences, an identifier is displayed on the left of each line to specify which reference sequence this repeat belongs to. Figure 4.20 shows a sample output of the program.

```
> gi|684237074|gb|CM002897.1|
       (14071     , 14105     )    (40       , 75       )     score: 51        size:35, 36
       (40266     , 40303     )    (40       , 76       )     score: 55        size:38, 37
       (77928     , 77966     )    (40       , 76       )     score: 56        size:39, 37
       (26397     , 26427     )    (41       , 73       )     score: 44        size:31, 33
       (40270     , 40304     )    (41       , 77       )     score: 52        size:35, 37
       (40453     , 40488     )    (41       , 76       )     score: 52        size:36, 36
       (40453     , 40490     )    (41       , 75       )     score: 53        size:38, 35
       (73048     , 73078     )    (41       , 71       )     score: 42        size:31, 31
       (84757     , 84791     )    (41       , 75       )     score: 50        size:35, 35
       (73047     , 73079     )    (43       , 75       )     score: 50        size:33, 33
       (73125     , 73154     )    (43       , 73       )     score: 45        size:30, 31
       (14072     , 14103     )    (44       , 76       )     score: 49        size:32, 33
       (39642     , 39673     )    (44       , 76       )     score: 45        size:32, 33
       (39742     , 39772     )    (44       , 76       )     score: 44        size:31, 33
       (72980     , 73011     )    (44       , 76       )     score: 45        size:32, 33
       (85669     , 85698     )    (44       , 74       )     score: 41        size:30, 31
       (87141     , 87173     )    (44       , 77       )     score: 51        size:33, 34
       (87390     , 87421     )    (44       , 75       )     score: 44        size:32, 32
       (28876     , 28907     )    (45       , 77       )     score: 49        size:32, 33
```

Figure 4.20: A sample output of the program with only one reference and one query sequence.

4.8 Usage

To run the program, three input files are required. The input files must contain the reference sequence, the query sequence, and the set of spaced seeds, respectively. Moreover, the user is provided with some options to adjust the desired settings or to reset default values of the program parameters. Below, we describe the available options.

4.8.1 Available Options

-t Option

The program is parallelized using OpenMP directives. With -t option, the user can set the number of threads in parallel mode. The default is 1 thread, i.e., the sequential execution.

-K Option

As we discussed in Section 4.1, unlike E-MEM, all the positions of the reference sequence are required to be indexed in our program. However, we provide the option of indexing every K positions for the users who prefer a faster and less space-consuming mode in sacrifice of accuracy. The K value, called the indexing step size, is set through -K option. The program scans both the reference and query sequences in every K positions. The default value for K is 1.

-d Option

To reduce the memory consumption, the input sequences can be split into d subsequences using -d option. The d subsequences are overlapped to avoid missing the repeats which locate at marginal regions. The reference subsequences are considered one at a time and a hash table is built for each. This way, the hash table memory requirement decreases by a factor of d. However, each reference subsequence must be compared with d query subsequences one by one. This increases the execution time, since the query sequence is scanned d times. The default value for d is 1. This procedure is implemented as a part of the E-MEM program. More details can be found in the E-MEM paper [18].

-D Option

This option is used to set D_{max}, the maximum edit distance (as defined in Section 2.3.1.2) between the repeat copies. By default D_{max} equals 5, which means that edit distance only up to 5 is tolerated between approximate repeat copies.

-s Option

The scoring system is set by -s option. It includes a match, a mismatch, and a gap score. Therefore, this option takes three numbers which can be positive or negative. The default scores are 2, -2, and -3, respectively for matches, mismatches and gaps.

-x Option

This option is to set the X_{drop} score. The default value of X_{drop} is 5.

-L Option

The user can set the minimum length of repeats, L_{min}, by this option to exclude shorter repeats from the output. The default value of L_{min} is the length of the shortest given seed.

-n Option

When choosing this option, the unknown bases (N's) of the input sequences are ignored. This is implemented as part of the E-MEM program [18] and works as follows. First of all, since the unknown bases usually appear as blocks of N's, the start and end positions of theses blocks are stored while reading the input sequences. Later in the hit detection process, the hits that share a region with these blocks are excluded from the rest of the computation. Finally in the extension process, repeat pairs are not extended beyond the borders of the N's blocks.

-r Option

In order to compute repeats between the reference and the reverse complement[6] of the query sequence, this option must be chosen. When choosing -r option, the program only computes the reserve complement repeats.

-b Option

-b is used to compute both forward and reverse complement repeats.

[6]The reverse complement of a DNA sequence is the sequence in reverse direction in which A and T letters are interchanged as are C and G letters.

-c Option

-c is set to report the query position of a reverse complement repeat relative to the original query sequence. This option must be used with either -b or -r.

-l Option

-l shows the length of query sequences on the header lines in the output.

-o Option

By default, the output is printed on the terminal. The -o option is used to print the output in a file, namely *Output*.

-h Option

-h displays the help menu, containing the list of all possible options.

Chapter 5

Results

This chapter presents the experimental results. We have tested our program under different parameter settings and studied the impact of these parameters on the program performance. In the first section, we briefly describe the test genomic sequences, the spaced seed sets, and the metrics we used for performance measurement. The effect of seed weight is studied in the next section, followed by the results obtained from varying the values of the -t, -K, -d, and -D parameters. We also examine the percentage of the running time spent on various parts of the program. In the two last sections, we compare the memory requirement of our program with PatternHunter II [13], and propose a comparison of run time, resource usage and output quality with YASS [19].

5.1 Preliminaries

The experiments are performed on some human, mouse and chimpanzee chromosomes[1]. For the spaced seeds, we use different sets of seeds. Some sets are directly computed by the algorithm of Ilie and Ilie [31], while others are combinations of the former sets. Tables 5.1 and 5.2 provide the information of the test sequences and the seed sets.

[1]Obtained from `http://www.ncbi.nlm.nih.gov/`

Table 5.1: The genomic sequences used in the experiments.

Id	**Desceription**	**Length (Base No.)**
Hchr19	Homo sapiens (human) chromosome 19	58,617,616
HchrY	Homo sapiens chromosome Y	57,227,415
HchrX	Homo sapiens chromosome X	156,040,895
Mchr19	Mus musculus (mouse) chromosome 19	61,431,566
MchrX	Mus musculus chromosome X	171,031,299
Pchr18	Pan troglodytes (chimpanzee) chromosome 18	76,611,499
PchrY	Pan troglodytes chromosome Y	26,342,871

Table 5.2: The spaced seed sets used in the experiments.

Set Id	Pattern	Length	Weight
2w11	111*1*1**1*11*111	17	11
	1111***1**1***1*1*111	21	
2w16	111*1*1**1*11*111**11*1*11	26	16
	1111***1**1***1*1*111**1*1111	29	
2w22	111*111**1*1111111***1***1****1*11**1*11	40	22
	11*11*1****1*1**111111*11***1*****1***1*1*111	45	
2w27	111*111**1*1111*111***1***1****1*11**1*11***11*1*11	51	27
	11*11*1****1*1**1111*11*11***1*****1***1*1*111***1*1111	55	
2w32	111*11*11*1*111111***1***1*1***11*11111*1*1****1**1**1****111	61	32
	11*1*1**11**1*111111**1***1***11****1*1*1111*11**1*1******1*1***111	67	
4w11	111*111**1*1111	15	11
	11*11*1****1*1**1111	20	
	111***1***1****1*11**1*11	25	
	11*11***1*****1***1*1*111	25	

All the experiments of this chapter are conducted on the same computers with 4 cores Intel i7-2600 processor at 8×3400 MHz and 16 GB of RAM, running Linux Mageia 1. Our program is used with the default parameters, unless otherwise stated.

5.1.1 Time Measurement

During the program execution, three times are recorded. The first is the time spent on constructing the hash tables, called the hash time. The hash time is actually the time of indexing the reference sequence, or step 2 of the program outline in Section 4.2. The second is the time spent on scanning the query sequence and extending the hits. This is the running time of step 3 in the outline which is shown to be the most time-consuming step of the program and is called the extension time. The third is the total execution time of the program.

5.1.2 Space Measurement

For space consumption, we report three numbers:

1. The hash tables size.
2. The physical memory (RAM) used by the current process.
3. The virtual memory used by the current process.

The hash tables size is the amount of memory used by the hash tables. This number can be compared with the estimate we made in Section 4.5.3.5. The amount of physical and virtual memory are measured by reading the *proc* filesystem during execution, when memory usage is at its maximum level. In the rest of this document, the total memory consumption refers to the sum of physical and virtual memory used in the program.

5.2 Seed Weight

In this section, we study the impact of seed weight on the run time, the space consumption, and the number of output repeats. The seed sets used in this experiment are $2w11$, $2w16$, $2w22$, $2w27$ and $2w32$. Figure 5.1 illustrates both the space consumption and the total execution time for these seed sets.

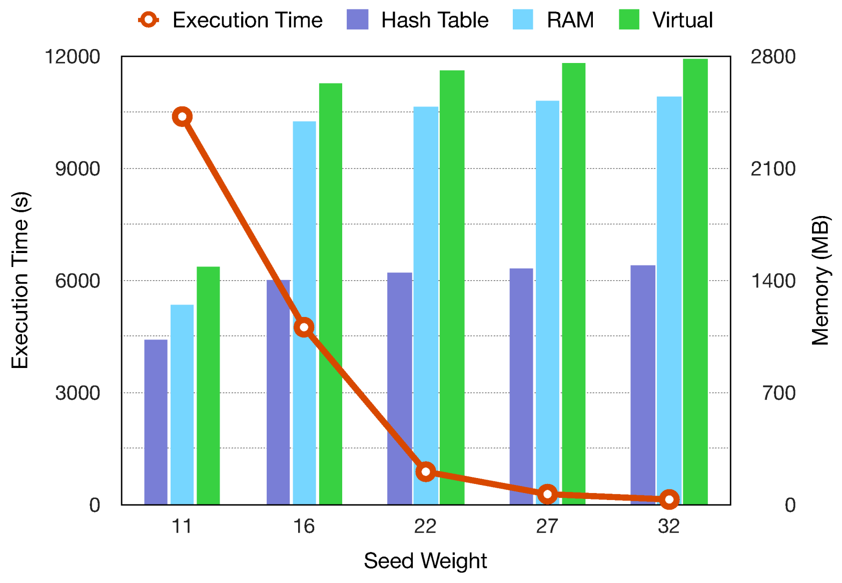


Figure 5.1: Hchr19 vs. Mchr19; Running the program with L_{min} set to be 50 and in parallel with 4 threads. The graph shows the space consumption and the running time for different seed weights.

The hash table size, RAM and virtual memory usage increases as the seeds gets heavier and longer. This can be interpreted as follows. As the seed weight increases, the number of possible key values increases as well. The more diverse key values require more entries in the hash table, and therefore, make the hash table bigger. On the other hand, the total execution time decreases significantly with increasing weight. Heavier seeds cause fewer matches between the hits of the reference and query because they impose a more strict matching pattern. This reduces the number of matching hit

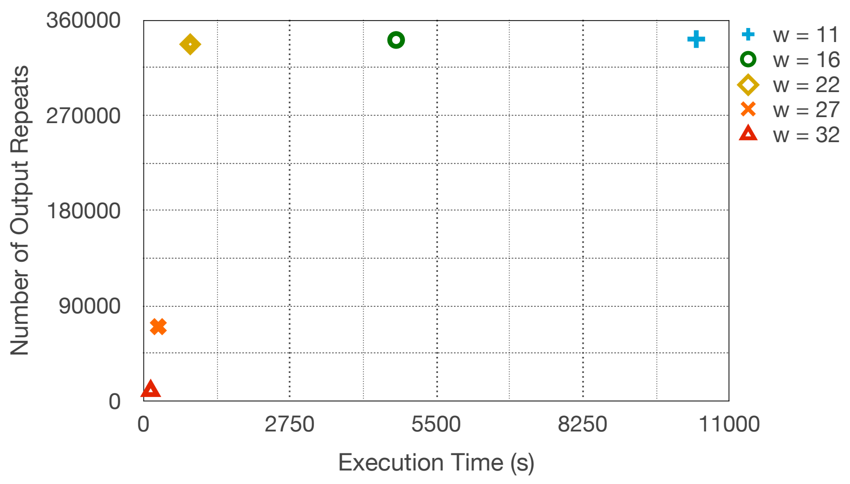
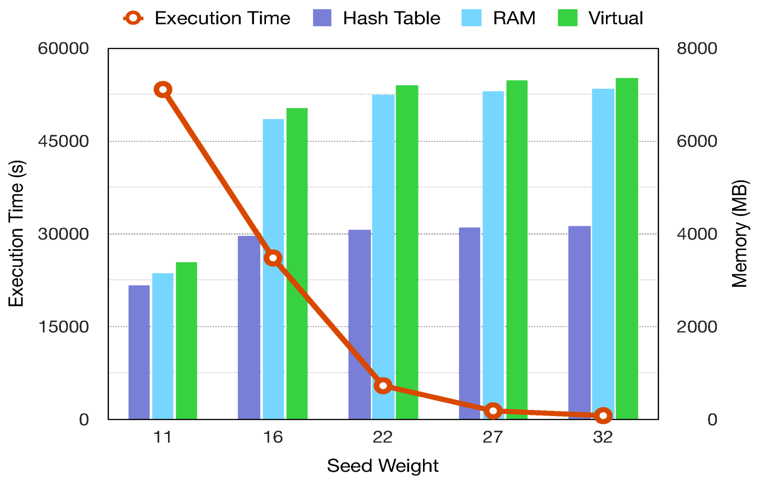


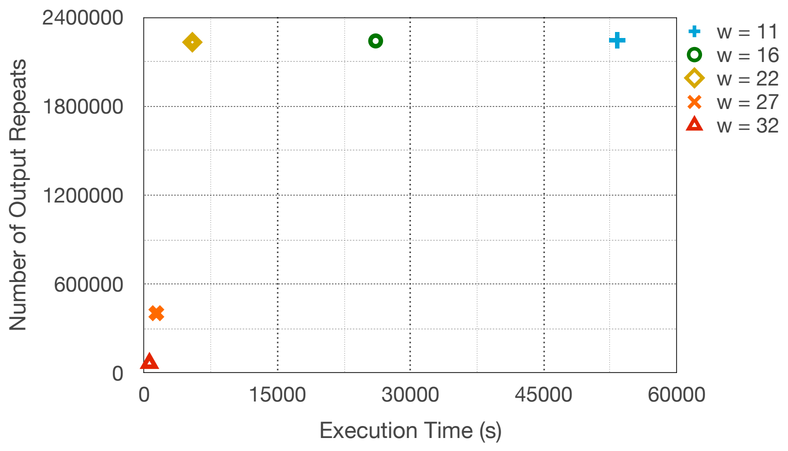
Figure 5.2: Hchr19 vs. Mchr19; Running the program with L_{min} set to be 50 and in parallel with 4 threads. The graph shows the number of output repeats and the running time for different seed weights.

pairs which are passed to the extension phase and yields a faster speed.

Figure 5.2 shows the trade-off between the number of output repeats and the execution time for the above-mentioned seed sets. According to this figure, the seed set $2w22$ gives almost the same accuracy as $2w16$ and $2w11$, while running at much faster speed. As anticipated, the faster speed is achieved at the cost of memory. Looking back at Figure 5.1, we see the memory consumption of $2w22$ is twice of $2w11$, but its execution time is about $\frac{1}{10}$ of $2w11$. This shows the superiority of $2w22$ against $2w11$ and $2w16$. The two other sets, $2w27$ and $2w32$, do not have a major difference with $2w22$ in terms of the space consumption and time, but they respectively miss about 75% and 97% of the repeats reported by $2w22$. The same experiment is conducted on MchrX vs. HchrX and the results are provided in Figure 5.3. The behaviour of the two graphs is similar to Figures 5.1 and 5.2. In this case, $2w27$ and $2w32$ fail to detect about, respectively, 81% and 94% of the repeats detect by $2w22$. To clarify the reason for this great loss, the average length of output repeats for each of these cases is given in Table 5.3. The average length of repeats reported by $2w11$, $2w16$, and $2w22$ is about 54. On the other hand, the seed length of the two other sets, $2w27$ and $2w32$, is above 54, except for one seed (see Table 5.2). These long seeds miss the shorter repeats which in this case constitute a major portion of the output repeats.



(a)



(b)

Figure 5.3: MchrX vs. HchrX; Running the program with L_{min} set to be 50 and in parallel with 4 threads; (a) shows the space consumption and the running time for different seed weights; (b) shows the number of output repeats and the running time for different seed weights.

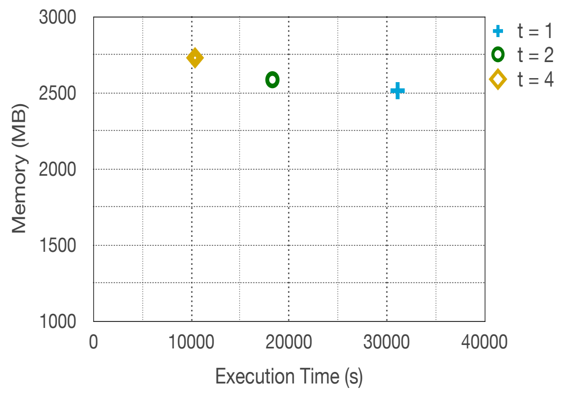
According to these experiments, it is preferred to use longer and heavier seeds with our program especially to improve the speed. However, it should be considered that very long seeds miss the short repeats.

Table 5.3: The average repeat length for different seed weights; L_{min} set to be 50.

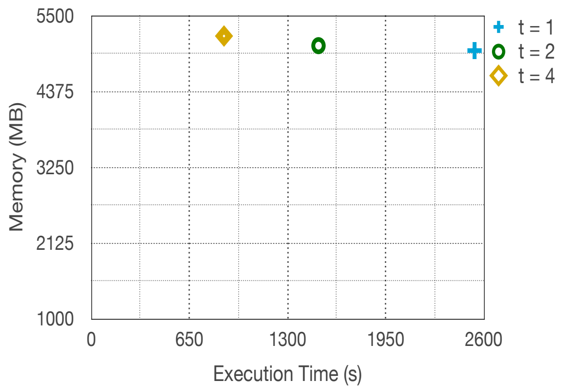
Seed Set	**2w11**	**2w16**	**2w22**	**2w27**	**2w32**
Hchr19 vs. Mchr19	53.98	53.97	53.98	59.99	68.94
MchrX vs. HchrX	53.82	53.82	53.82	60.31	68.44

5.3 Number of Threads: -t parameter

We run the program in serial and parallel with 2 and 4 threads. As shown in Figure 5.4, increasing the number of threads, t, from 1 to 2 decreases the total execution time by about half, and increasing it to 4 reduces the execution time to about $\frac{1}{3}$ of the serial time. Notice that even in the parallel mode the hash tables are constructed in serial. Parallelization is applied only to the process of scanning the query and extension, so the hash time remains unchanged. Figure 5.4 also shows that a small overhead in space is caused by parallelization. This overhead does not include the hash table size, since running the program in parallel does not change the number of stored hits. Also, as expected, parallelization does not affect the output repeats in any way.



(a)



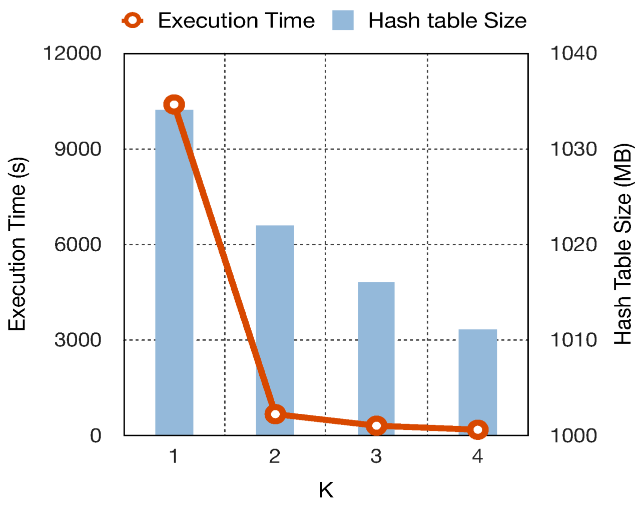
(b)

Figure 5.4: Hchr19 vs. Mchr19; Running the program with L_{min} set to be 50 and (a) $2w11$ (b) $2w22$ as the spaced seed set. The graphs show the total memory consumption and the execution time in serial and parallel modes.

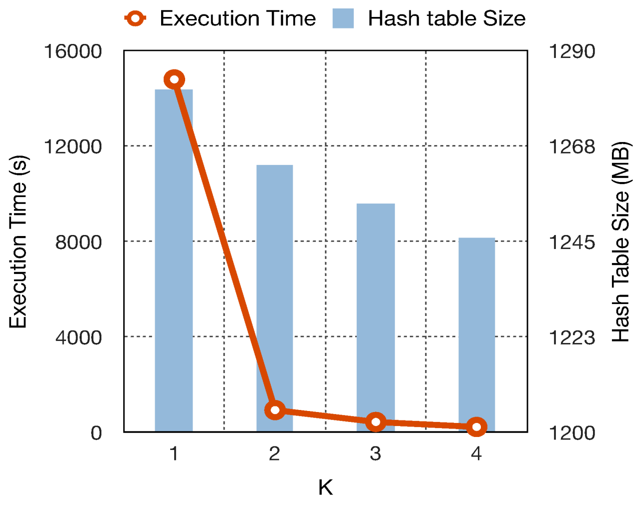
5.4 Indexing Step Size: -K parameter

In this section, the program is tested for different values of -K option which defines the indexing step size (see Section 4.8.1). The variation of running time, space usage and number of output repeats is studied here. Figure 5.5 demonstrates a decrease in the total execution time and the hash table size for higher values of K. It is seen that, the reduction in the execution time is much faster than the reduction in the hash table size as K increases. This indicates that in addition to the hits that are not indexed into the hash table due to $K = 2$, numerous hits that were matching before are also lost. These latter lost matches have a great impact on the execution time.

We expect a loss of accuracy for K values greater than 1 as discussed in Section 4.8.1. Figure 5.6 illustrates the trade-off between the running time and the number of output repeats for different K values. According to this figure, a large number of repeats (about 72%) is missed only by increasing K from 1 to 2. However, the running time also decreases by about 93%. In this experiment, we used $2w11$ and $4w11$ as the



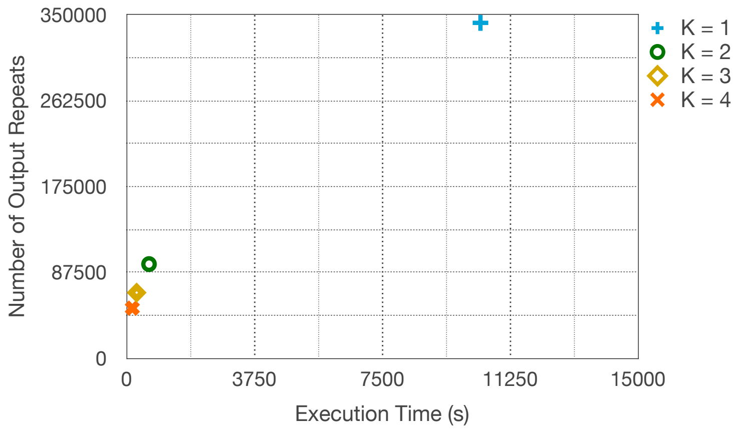
(a)



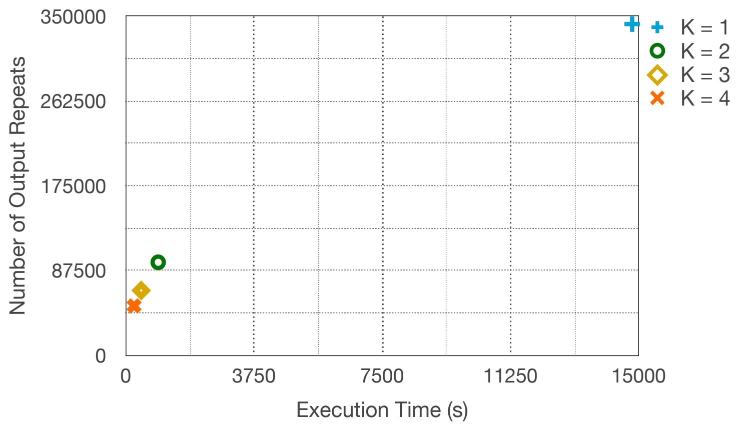
(b)

Figure 5.5: Hchr19 vs. Mchr19; Running the program in parallel with 4 threads; L_{min} set to be 50 and (a) $2w11$ (b) $4w11$ is used as the spaced seed set. These graphs show the hash table size and the running time for different K values.

spaced seed sets to see whether employing more spaced seeds can compensate the loss of accuracy caused by increasing K. However, for different K values, $4w11$ generated almost the same number of repeats as $2w11$.



(a)

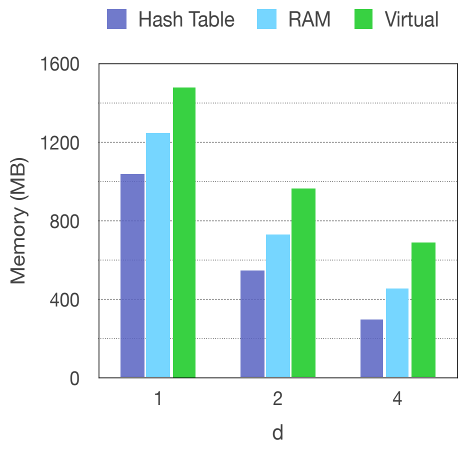


(b)

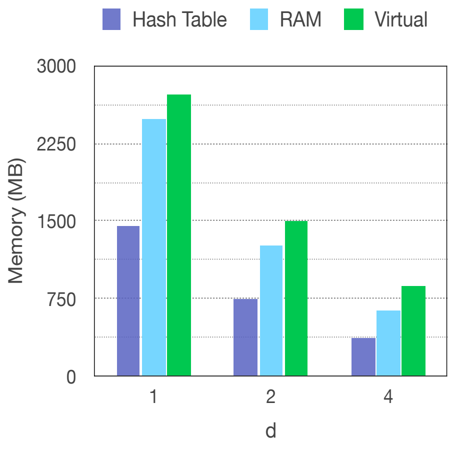
Figure 5.6: Hchr19 vs. Mchr19; Running the program in parallel with 4 threads; L_{min} set to be 50 and (a) $2w11$ (b) $4w11$ is used as the spaced seed set. These graphs show the number of output repeats and the running time for different K values.

5.5 Number of Subsequences: -d parameter

The parameter d, the number of divided subsequences, is expected to affect both the space usage and the execution time as discussed in Section 4.8.1. The hash table size, RAM and virtual memory usage for various d values are graphed in Figure 5.7. Notice that since the reference sequence is divided into d subsequences, the hash tables are constructed and destroyed d times. We have chosen the largest of these d values to represent the hash table size of the program. In Figure 5.7, raising d by a factor of 2 decreases the hash table size, RAM, and virtual memory by a factor between 1.4 and 2. Figure 5.8 shows the total execution time which experiences only a small increase due to scanning the query sequence multiple times.



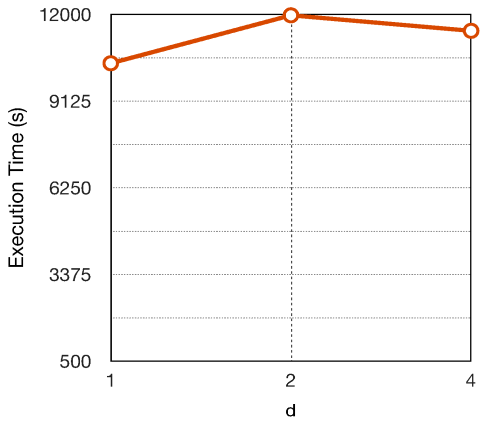
(a)



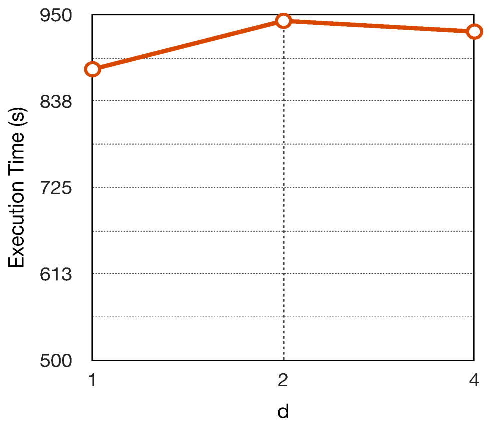
(b)

Figure 5.7: Hchr19 vs. Mchr19; Running the program in parallel with 4 threads; L_{min} set to be 50 and (a) $2w11$ (b) $2w22$ is used as the spaced seed set. These graphs show the space consumption for different d values.

This parameter does not have any effect on the accuracy. It has been designed only to save more memory, if it is necessary.



(a)



(b)

Figure 5.8: Hchr19 vs. Mchr19; Running the program in parallel with 4 threads; L_{min} set to be 50 and (a) $2w11$ (b) $2w22$ is used as the spaced seed set. These graphs show the running time for different d values.

5.6 Maximum Edit Distance: -D parameter

The program can produce more and longer repeats if one allows higher edit distances between copies of a repeat with -D option. In Figure 5.9, we show the effect of increasing D_{max} on the number of output repeats as well as the running time. This parameter does not affect the space consumption.

It is shown that increasing D_{max} from 5 to 15 moderately slows down the program, while multiplying the number of output repeats by about 16. One can reduce this number by choosing a larger L_{min} which excludes the shorter repeats, but this does not make a considerable difference to the running time or the space usage. The reason is that the shorter repeats are excluded after the extension phase, prior to the last step of the algorithm[2]. Notice that further increase of D_{max} does not provide gains

[2]Notice that a repeat length is determined only after it has been extended, therefore the short repeats cannot be identified earlier. However, the program has a mechanism to avoid extending the hit pairs that are not potentially long enough: If a hit pair is located in a region which is either surrounded by blocks of N's or close to the sequence endpoints and cannot be extended at least as much as L_{min}, it is excluded before the extension is applied.

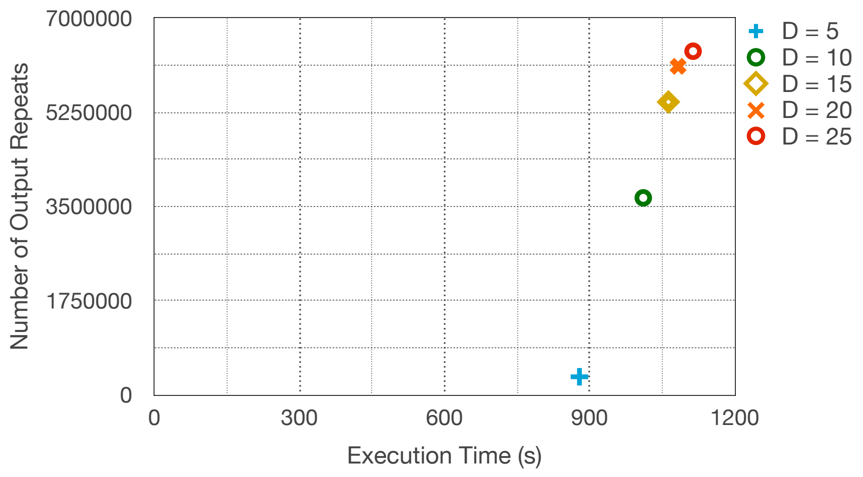


Figure 5.9: Hchr19 vs. Mchr19; Running the program in parallel with 4 threads; L_{min} set to be 50 and $2w22$ is used as the spaced seed set.

as much as it did before and the graph seems to be convergent. One reason for this phenomenon is the fixed amount of X_{drop} score during the experiment which avoids the extension to be continued past the X_{drop} threshold. Table 5.4 shows a similar increasing behaviour for the average and the maximum repeat length as higher values of D_{max} are applied. Although, the average repeat length grows slowly, the maximum length is almost doubled by increasing the maximum distance from 5 to 10. The third row of this table contains the values of D_{max}/L_{min} which indicate the maximum value of edit distance in unit length for the repeats of these experiments. This number is increasing from 0.1 to 0.5, which means that the repeat copies in $D_{max} = 5$ are at most 10% divergent from one another, while they are up to 50% divergent in case of $D_{max} = 25$.

Table 5.4: Repeat length information for Hchr19 vs. Mchr19; Running the program in parallel with 4 threads; L_{min} set to be 50 and $2w22$ is used as the spaced seed set.

D_{max}	**5**	**10**	**15**	**20**	**25**
Average Repeat Length	53.98	57.89	62.68	66.18	68.63
Maximum Repeat Length	120	202	225	267	288
D_{max}/L_{min}	0.1	0.2	0.3	0.4	0.5

5.7 Hash Time vs. Extension Time

This section justifies our claim regarding the extension phase being the most time-consuming step of the program. Table 5.5 gives the hash time and the extension time for several runs with different parameter settings. The last two columns represent the percentage of the total running time spent on each of these two steps. We see that the hash time is often less than 6% and always less than 50% of the total execution time. On the other hand, in most cases, the extension time occupies more than 90% of the total execution time. There is a non-linear increase in the hash percentage for the the first two sets of test (Hchr19 vs. Mchr19 and MchrX vs. HchrX) as the seed weight increases. This non-linear behaviour comes from the slow increase in the hash time and the fast decrease in the total execution time, which is also seen in Figures 5.1 and 5.3a. As explained in Section 5.2, the fast decrease of the total execution time is the result of reduction in the number of matching hits and fewer number of invoked extensions.

Table 5.5: Execution time in seconds for various steps of the program; Running the program in parallel with 4 threads; L_{min} set to be 50.

	Seed Set	Hash Time (s)	Extension Time (s)	Total Running Time (s)	Hash Percentage	Extension Percentage
Hchr19 vs. Mchr19 score: (+2,-2,-3)	$2w11$	32	10,339	10,381	0.30%	99.6%
	$2w16$	42	4,682	4,745	0.88%	98.7%
	$2w22$	49	810	879	5.57%	92.1%
	$2w27$	54	206	279	19.35%	73.8%
	$2w32$	55	61	136	40.44%	44.8%
MchrX vs. HchrX score: (+2,-2,-3)	$2w11$	94	53,147	53,285	0.18%	99.7%
	$2w16$	140	25,834	26,067	0.54%	99.1%
	$2w22$	142	5,204	5,418	2.62%	96%
	$2w27$	153	1,145	1,359	11.29%	84.2%
	$2w32$	163	357	581	28.05%	61.4%
HchrY vs. PchrY score: (+1,-3,-4)	$2w11$	14	3,650	3,677	0.38%	99.3%
	$2w16$	18	2,748	2,783	0.65%	98.7%
	$2w22$	18	2,052	2,093	0.86%	98%
	$2w27$	20	1,689	1,725	1.16%	97.9%
	$2w32$	21	1,462	1,499	1.40%	97.5%

5.8 PatternHunter II

Here, the hash table size of our program is compared with the one of PatternHunrter II [13]. The PatternHunter II hash table has a fixed size of $(4^{w+1} + 4n_1)k$ bytes, in which k is the number of seeds, w is the seed weight, and n_1 is the reference sequence length (see Section 3.2.3.1). In Section 4.5.3.5, we provided an estimate of the hash table size in our program which is $(8k+9)n_1$ bytes. Here, we also compare this estimate with the real size. Tables 5.6 to 5.9 contain the information about the hash table size for 4 different sequences. Our program has been executed with default parameter settings in all these tests.

Table 5.6: The hash table size (in GB) for Hchr19 as the reference sequence; n_1 equals $58,617,616$ and k is 2.

Seed Set	**2w11**	**2w16**	**2w22**
PatternHunter II	0.47	32.44	131,072.44
Our Estimate	1.36	1.36	1.36
Real Hash Table Size	1.02	1.38	1.43

Table 5.7: The hash table size (in GB) for Pchr18 as the reference sequence; n_1 equals $76,611,499$ and k is 2.

Seed Set	**2w11**	**2w16**	**2w22**
PatternHunter II	0.6	32.57	131,072.57
Our Estimate	1.78	1.78	1.78
Real Hash Table Size	1.30	1.87	1.91

The PatternHunter II hash table is smaller than our hash table (about half-size) for a light seed like $2w11$. However, it requires an extreme amount of memory with seeds

Table 5.8: The hash table size (in GB) for HchrX as the reference sequence; n_1 equals $156,040,895$ and k is 2.

Seed Set	**2w11**	**2w16**	**2w22**
PatternHunter II	1.19	33.16	131,073.16
Our Estimate	3.63	3.63	3.63
Real Hash Table Size	2.56	3.65	3.80

Table 5.9: The hash table size (in GB) for MchrX as the reference sequence; n_1 equals $171,031,299$ and k is 2.

Seed Set	**2w11**	**2w16**	**2w22**
PatternHunter II	1.30	33.27	131,073.27
Our Estimate	3.98	3.98	3.98
Real Hash Table Size	2.81	3.85	4

of weight 16 or higher which makes the use of these seeds impractical. We see that our estimate for the table size is not dependent on the seed weight, and similarly, the real size of hash tables is not considerably affected by the seed weight. There is a small increase in our hash table size for higher weights which is the result of more diverse key values generated by these weights. As it is seen and explained in Section 5.2, diversity of the key values increases the number of occupied entries in the hash table.

5.9 YASS

Here, we compare the running time, the space usage and the output of our program with YASS [19] which was introduced in Section 3.2.4. Table 5.10 provides this information for Hchr19 and Mchr19. In this experiment, the space consumption of the two programs are almost equal. Our program runs about 6 to 7 times faster than

YASS, but the output repeats of YASS are significantly longer. One reason for the longer repeats of YASS is the criterion it uses for merging the neighbouring hits. It groups the nearby hits together to generate large repeats.

Table 5.10: Hchr19 vs. Mchr19; Running both programs in parallel with 4 threads; $2w22$ is used as the spaced seed set in our program.

	YASS	$D_{max} = 5$	$D_{max} = 20$	$D_{max} = 30$
		$L_{min} = 50$	$L_{min} = 100$	$L_{min} = 120$
No. of Output Repeats	302,678	337,093	179,279	232,135
Average Length	484.901	53.9848	112.989	138.758
Maximum Length	3,799	120	267	295
Execution time (s)	6,661	879	1,042	1,052
RAM (GB)	2.52	2.42	2.42	2.42
Virtual Memory (GB)	2.75	2.65	2.65	2.65

YASS does not provide an option to control the distance between repeat copies. Therefore, we tested our program with different D_{max} values and tried to approach YASS lengths, but we are still far from those lengths. We also conducted another comparison experiment for HchrX and MchrX. The results are given in Table 5.11. This time our program is executed 10 times faster and consumes less memory, but again YASS reports much longer repeats. The number of output repeats of our program with $D_{max} = 5$ is high as compared to YASS. Therefore, L_{min} and D_{max} were increased in the next two tests to exclude some shorter repeats and include more longer ones.

Considering the large difference in repeat length of the two programs, we expect a considerable difference in distance of repeat copies as well. Our program guarantees not to report repeats with an edit distance more than D_{max}, but as mentioned above, there is no control over the maximum allowable distance in YASS. We evaluated the edit

Table 5.11: HchrX vs. MchrX; Running both programs in parallel with 4 threads; $2w22$ is used as the spaced seed set in our program.

	YASS	$D_{max} = 5$	$D_{max} = 15$	$D_{max} = 20$
		$L_{min} = 50$	$L_{min} = 100$	$L_{min} = 150$
No. of Output Repeats	70,046	2,227,364	266,177	22,704
Average Length	2,444.11	53.8301	110.816	165.85
Maximum Length	7,984	275	526	833
Execution time (s)	67,479	5,801	6,980	7,183
RAM (GB)	9.64	6.49	6.49	6.49
Virtual Memory (GB)	9.86	6.71	6.71	6.71

distance (as defined in Section 2.3.1.2) of 50 randomly selected output repeats of YASS and compared them with similar data from our program output. The results are given in Tables 5.12 and 5.13. The output repeats in YASS have higher edit distances in comparison with our program output, where edit distances does not exceed D_{max}. The average ratio of edit distance to repeat length is also computed to scale the distances to unit length. This ratio is higher for YASS which indicates lower similarity between the repeat copies of its output.

Table 5.12: Hchr19 vs. Mchr19; Comparing the average edit distance of 50 randomly selected output repeats of YASS with our program in different settings.

	YASS	$D_{max} = 5$	$D_{max} = 20$	$D_{max} = 30$
		$L_{min} = 50$	$L_{min} = 100$	$L_{min} = 120$
Avg. Edit Distance	127.2	4.7	16.7	25.3
Avg. Length	421.5	54.7	111.1	141.3
Avg. Edit Distance/Length	0.3	0.08	0.15	0.18

Table 5.13: HchrX vs. MchrX; Comparing the average edit distance of 50 randomly selected output repeats of YASS with our program in different settings.

	YASS	$D_{max} = 5$	$D_{max} = 15$	$D_{max} = 20$
		$L_{min} = 50$	$L_{min} = 100$	$L_{min} = 150$
Avg. Edit Distance	706.3	4.5	12.9	17.2
Avg. Length	2359.5	53.4	112.6	166.3
Avg. Edit Distance/Length	0.3	0.08	0.11	0.1

Chapter 6

Conclusions

6.1 Summary and Conclusions

We proposed a new software program for detecting approximate repeats between two DNA sequences. The program builds an efficient hash table for indexing sequences with multiple spaced seeds. The hash table combines the speed advantage of open addressing and space saving of dynamic resizing. The hash table size is not sensitive to the seed weight and enables us to use spaced seeds of any length with weights up to 32. Employing large spaced seeds of such weights is almost impractical in previously proposed similar software programs. It was shown by experiment that we can achieve faster speed using longer seeds of higher weights without losing significant accuracy.

There is always a trade-off between the accuracy and resource usage. We showed this trade-off by plotting the number of output repeats against execution time for different parameter settings. The program provides the user with a control over the similarity degree of output repeats in the form of maximum edit distance. With this parameter, the user can increase the number of output repeats and get more distant ones, but at the cost of speed. The efficient use of heavy spaced seeds, along with the bound on the maximum allowable distance between repeat copies, makes the program an appropriate tool for detecting repeats of higher similarity.

Similar to all the hit and extend methods, the initial similarities, or hits, detected by the hash table are extended into longer repeats. Our program first runs a simple and fast gapless extension to exclude regions of low similarity. Then, it applies a second extension which allows gaps using dynamic programming. This step was shown to be the most time-consuming part of the program and is a barrier to getting very long and distant repeats.

6.2 Future Work

In this project, we mainly focused on the hit detection process and the hash table optimization. The hit extension process is still an expensive procedure and needs to be improved. To make the output repeats of this program longer, a possible solution would be merging of the overlapping or nearby repeats together. Implementing this solution will need some computations and definition of metrics, like the ones defined in YASS, to specify which repeats are suitable to be grouped together. This way not only can the program detect longer similarities, but the time-consuming extension process can be less frequently invoked or replaced by faster procedures.

Bibliography

[1] E. Rivals, L. Salmela, and J. Tarhio, “Exact search algorithms for biological sequences”, in *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*, M. Elloumi and A. Y. Zomaya, Eds. Hoboken, NJ: John Wiley & Sons, 2011, ch. 5, pp. 91–111. DOI: 10.1002/9780470892107.ch5.

[2] S. Faro and T. Lecroq, “The exact online string matching problem: a review of the most recent results”, *ACM Comput. Surv.*, vol. 45, no. 2, 13–13:42, Mar. 2013. DOI: 10.1145/2431211.2431212.

[3] G. Achaz, F. Boyer, E. P. Rocha, A. Viari, and E. Coissac, “Repseek, a tool to retrieve approximate repeats from large DNA sequences”, *Bioinformatics*, vol. 23, no. 1, pp. 119–121, Jan. 2007. DOI: 10.1093/bioinformatics/btl519.

[4] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences”, *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, Mar. 1981. DOI: 10.1016/0022-2836(81)90087-5.

[5] D. J. Lipman and W. R. Pearson, “Rapid and sensitive protein similarity searches”, *Science*, vol. 227, no. 4693, pp. 1435–1441, Mar. 1985. DOI: 10.1126/science.2983426.

[6] W. R. Pearson and D. J. Lipman, “Improved tools for biological sequence comparison”, *Proc. Natl. Acad. Sci. U. S. A.*, vol. 85, no. 8, pp. 2444–2448, Apr. 1988.

[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool”, *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct. 1990. DOI: 10.1016/S0022-2836(05)80360-2.

[8] L. Noé and G. Kucherov, “Improved hit criteria for DNA local alignment”, *BMC Bioinf.*, vol. 05, no. 1, p. 149, Oct. 2004. DOI: 10.1186/1471-2105-5-149.

[9] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs”, *Nucleic Acids Res.*, vol. 25, no. 17, pp. 3389–3402, 1997. DOI: 10.1093/nar/25.17.3389.

[10] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, “A greedy algorithm for aligning DNA sequences”, *J. Comput. Biol.*, vol. 7, no. 1-2, pp. 203–214, Feb. 2000. DOI: 10.1089/10665270050081478.

[11] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller, “Human–mouse alignments with BLASTZ”, *Genome Res.*, vol. 13, no. 1, pp. 103–107, Jan. 2003. DOI: 10.1101/gr.809403.

[12] B. Ma, J. Tromp, and M. Li, “PatternHunter: faster and more sensitive homology search”, *Bioinformatics*, vol. 18, no. 3, pp. 440–445, Mar. 2002. DOI: 10.1093/bioinformatics/18.3.440.

[13] M. Li, B. Ma, D. Kisman, and J. Tromp, “PatternHunter II: highly sensitive and fast homology search”, *J. Bioinf. Comput. Biol.*, vol. 2, no. 3, pp. 417–439, 2004. DOI: 10.1142/S0219720004000661.

[14] B. Brejová, D. G. Brown, and T. Vinař, “Vector seeds: an extension to spaced seeds”, *J. Comput. Syst. Sci.*, vol. 70, no. 3, pp. 364–380, May 2005. DOI: `10.1016/j.jcss.2004.12.008`.

[15] J. Buhler, U. Keich, and Y. Sun, “Designing seeds for similarity search in genomic DNA”, *J. Comput. Syst. Sci.*, vol. 70, no. 3, pp. 342–363, May 2005. DOI: `10.1016/j.jcss.2004.12.003`.

[16] Y. Sun and J. Buhler, “Designing multiple simultaneous seeds for DNA similarity search”, *J. Comput. Biol.*, vol. 12, no. 6, pp. 847–861, Jul. 2005. DOI: `10.1089/cmb.2005.12.847`.

[17] M. Csuros and B. Ma, “Rapid homology search with neighbor seeds”, *Algorithmica*, vol. 48, no. 2, pp. 187–202, May 2007. DOI: `10.1007/s00453-007-0062-y`.

[18] N. Khiste and L. Ilie, “E-MEM: efficient computation of maximal exact matches for very large genomes”, *Bioinformatics*, vol. 31, no. 4, pp. 509–514, Feb. 2015. DOI: `10.1093/bioinformatics/btu687`.

[19] L. Noé and G. Kucherov, “YASS: enhancing the sensitivity of DNA similarity search”, *Nucleic Acids Res.*, vol. 33, no. suppl 2, W540–W543, Jul. 2005. DOI: `10.1093/nar/gki478`.

[20] W. F. Smyth, *Computing Patterns in Strings.* Harlow, England: Pearson Addison-Wesley, 2003.

[21] M. C. Frith and L. Noé, “Improved search heuristics find 20 000 new alignments between human and mouse genomes”, *Nucleic Acids Res.*, vol. 42, no. 7, e59, Apr. 2014. DOI: `10.1093/nar/gku104`.

[22] D. G. Brown, "A survey of seeding for sequence alignment", in *Bioinformatics Algorithms*, I. I. Mandoiy and A. Zelikovsky, Eds. Hoboken, NJ: John Wiley & Sons, 2008, ch. 6, pp. 117–142. DOI: `10.1002/9780470253441.ch6`.

[23] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes", *Nucleic Acids Res.*, vol. 27, no. 11, pp. 2369–2376, Jun. 1999. DOI: `10.1093/nar/27.11.2369`.

[24] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970. DOI: `10.1016/0022-2836(70)90057-4`.

[25] K. Popendorf, H. Tsuyoshi, Y. Osana, and Y. Sakakibara, "Murasaki: a fast, parallelizable algorithm to find anchors from multiple genomes", *PLoS One*, vol. 5, no. 9, e12651, 2010. DOI: `10.1371/journal.pone.0012651`.

[26] K. P. Choi and L. Zhang, "Sensitivity analysis and efficient method for identifying optimal spaced seeds", *J. Comput. Syst. Sci.*, vol. 68, no. 1, pp. 22–40, Feb. 2004. DOI: `10.1016/j.jcss.2003.04.002`.

[27] K. P. Choi, F. Zeng, and L. Zhang, "Good spaced seeds for homology search", *Bioinformatics*, vol. 20, no. 7, pp. 1053–1059, Feb. 2004. DOI: `10.1093/bioinformatics/bth037`.

[28] I.-H. Yang, S.-H. Wang, Y.-H. Chen, P.-H. Huang, L. Ye, X. Huang, and K.-M. Chao, "Efficient methods for generating optimal single and multiple spaced seeds", in *Proc. 4th IEEE Symp. Bioinformatics and Bioengineering (BIBE)*, 2004, pp. 411–416. DOI: `10.1109/BIBE.2004.1317372`.

[29] F. P. Preparata, L. Zhang, and K. P. Choi, "Quick, practical selection of effective seeds for homology search", *J. Comput. Biol.*, vol. 12, no. 9, pp. 1137–1152, Nov. 2005. DOI: `10.1089/cmb.2005.12.1137`.

[30] Y. Kong, "Generalized correlation functions and their applications in selection of optimal multiple spaced seeds for homology search", *J. Comput. Biol.*, vol. 14, no. 2, pp. 238–254, Mar. 2007. DOI: `10.1089/cmb.2006.0008`.

[31] L. Ilie and S. Ilie, "Multiple spaced seeds for homology search", *Bioinformatics*, vol. 23, no. 22, pp. 2969–2977, 2007. DOI: `10.1093/bioinformatics/btm422`.

[32] L. Egidi and G. Manzini, "Spaced seeds design using perfect rulers", in *Proc. 18th Int. Symp. String Processing and Information Retrieval (SPIRE)*, 2011, pp. 32–43. DOI: `10.1007/978-3-642-24583-1_5`.

[33] ——, "Better spaced seeds using quadratic residues", *J. Comput. Syst. Sci.*, vol. 79, no. 7, pp. 1144–1155, Nov. 2013. DOI: `10.1016/j.jcss.2013.03.002`.

[34] G. Kucherov, L. Noe, and M. Roytberg, "A unifying framework for seed sensitivity and its application to subset seeds", *J. Bioinf. Comput. Biol.*, vol. 4, no. 2, pp. 553–569, Apr. 2006. DOI: `10.1142/S0219720006001977`.

[35] L. Ilie, S. Ilie, and A. Mansouri Bigvand, "SpEED: fast computation of sensitive spaced seeds", *Bioinformatics*, vol. 27, no. 17, pp. 2433–2434, 2011. DOI: `10.1093/bioinformatics/btr368`.

[36] S. Ilie, "Efficient computation of spaced seeds", *BMC Res. Notes*, vol. 5, no. 1, 123, Feb. 2012. DOI: `10.1186/1756-0500-5-123`.

[37] D. Do Duc, H. Q. Dinh, T. H. Dang, K. Laukens, and X. H. Hoang, "AcoSeeD: an ant colony optimization for finding optimal spaced seeds in biological sequence

search", in *Proc. 8th Int. Conf. Swarm Intelligence (ANTS)*, 2012, pp. 204–211. DOI: `10.1007/978-3-642-32650-9_19`.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd. ed. Cambridge, Massachusetts: MIT Press, 2009.