

---

Electronic Thesis and Dissertation Repository

---

4-16-2020 10:30 AM

## Computation of Sensitive Multiple Spaced Seeds

Arnab Mallik  
*The University of Western Ontario*

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

© Arnab Mallik 2020

Follow this and additional works at: <https://ir.lib.uwo.ca/etd>

---

### Recommended Citation

Mallik, Arnab, "Computation of Sensitive Multiple Spaced Seeds" (2020). *Electronic Thesis and Dissertation Repository*. 6977.  
<https://ir.lib.uwo.ca/etd/6977>

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact [wlsadmin@uwo.ca](mailto:wlsadmin@uwo.ca).

---

Supervisor

Ilie, Lucian

*The University of Western Ontario*

# Abstract

Similarity search is one of the most important problem in bioinformatics, with application in read mapping, homology search, oligonucleotide design, etc. Similarity search is time and memory intensive, hence heuristic methods using multiple spaced seeds are commonly employed. A spaced seed is a string of 1 and \*, where 1 represents a match position and \* represent don't care position. Seeds are used to discover regions with identity, thus, it is imperative to design seeds of high sensitivity, so as to maximize the number of hits.

We present SpEED2, a software program to generate multiple spaced seeds of high sensitivity. It uses a novel seed optimization approach and it outperforms all the leading programs used for designing multiple spaced seeds like Iedera, AcoSeed, and rasbhari. Our algorithm will benefit several software that is dependent on good quality seeds for its operation like Pattern-Hunter for similarity search, SHRiMP and BFAST for read mapping, bestPrimer for designing primers, and many more.

**Keywords:** Multiple Spaced Seeds, Similarity Search, Local Alignment, Heuristic Algorithm

## Lay Summary

Multiple spaced seed is a set of one or more spaced seeds and they are used to find similar regions between two biological sequences. A spaced seed is a string of 1 and \*, where 1 represents a match position and \* represent don't care position. Two sequences are similar if they are highly identical.

When a spaced seed is arranged with the two biological sequences to identify regions of similarity, a hit occurs only if both the sequences are identical in all the match positions. However, we are not interested in the don't care positions.

A measurement metric called sensitivity is used to differentiate good seeds from bad ones. Sensitivity is the probability of detecting similar regions between sequences, good multiple spaced seeds have high sensitivity whereas bad ones will have low sensitivity.

We have developed a software program named SpEED2 that generates multiple spaced seeds of high sensitivity. In this dissertation, we first look into previous related works, then we explain our algorithm and finally compare our results with other leading software concerned with generating multiple spaced seeds and show our software program to be better than the others.

## Acknowledgements

I would like to express my most sincere gratitude to my supervisor Dr. Lucian Ilie for his valuable guidance and advice. His encouragement and feedback led me to complete this dissertation. It was an absolute honour and privilege to work with such a wise person. I would also like to show my heartiest gratitude to all the professors who taught me and helped me build the background for this dissertation.

I would like to thank my father Manas Kumar Mallik and my mother Ruby Mallik for their unconditional love and support throughout this arduous journey.

I acknowledge all my friends and lab mates for always motivating me and helping me overcome all hurdle. Last but not the least, I thank the Department of Computer Science at Western University for funding my graduate studies.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>Lay Summary</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>viii</b>
<b>List of Appendices</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Problem Statement . . . . .	2
1.3 Objective . . . . .	2
1.4 Thesis Contribution . . . . .	2
1.5 Thesis Outline . . . . .	3
<b>2 Background</b>	<b>4</b>
2.1 Sequence Alignment . . . . .	5
2.2 Traditional Methods . . . . .	5
2.2.1 Needleman-Wunsch Algorithm . . . . .	6
2.2.2 Smith-Waterman Algorithm . . . . .	7
2.3 Heuristic Methods . . . . .	8
2.3.1 Suffix Tree Algorithms . . . . .	8
2.3.1.1 MUMmer . . . . .	9
2.3.1.2 Quasar . . . . .	11
2.3.2 Seeding Based Algorithms . . . . .	12
2.3.2.1 FASTA . . . . .	13
2.3.2.2 BLAST . . . . .	13
2.3.2.3 PatternHunter . . . . .	16
2.3.2.4 PatternHunter II . . . . .	18
2.4 Designing Multiple Spaced Seeds . . . . .	18
2.4.1 Iedera . . . . .	19
2.4.2 AcoSeeD . . . . .	20
2.4.3 rasbhari . . . . .	22

<b>3</b>	<b>Methodology</b>	<b>25</b>
3.1	SpEED . . . . .	25
3.1.1	Sensitivity . . . . .	25
3.1.2	Overlap Complexity . . . . .	26
3.1.3	Iterative Hill Climbing with Random Starts . . . . .	26
3.1.4	Algorithm . . . . .	27
3.1.5	SpEED-Fast . . . . .	28
3.2	SpEED2 . . . . .	28
3.2.1	Estimated Sensitivity . . . . .	29
3.2.2	Indel Optimization . . . . .	32
3.2.3	Adaptive Seed Length . . . . .	35
3.2.4	SpEED2 Algorithm . . . . .	36
3.2.4.1	Compute min & max seed lengths . . . . .	36
3.2.4.2	Compute all seed lengths . . . . .	37
3.2.4.3	Create homologous region . . . . .	37
3.2.4.4	Populate the seeds with default value . . . . .	37
3.2.4.5	Iterative hill climbing . . . . .	37
3.2.4.6	Adaptive seed lengths . . . . .	37
3.2.4.7	Indel optimization . . . . .	37
3.2.4.8	Calculate sensitivity . . . . .	37
<b>4</b>	<b>Experimental Results</b>	<b>39</b>
4.1	Operation Environment . . . . .	39
4.2	Experimental Setup . . . . .	39
4.3	Performance Measurement Metric . . . . .	40
4.4	Experimental Results . . . . .	40
4.5	Comparison . . . . .	41
<b>5</b>	<b>Conclusion and Future Works</b>	<b>43</b>
5.1	Summary . . . . .	43
5.2	Conclusion . . . . .	43
5.3	Future Works . . . . .	44
	<b>Bibliography</b>	<b>45</b>
<b>A</b>	<b>SpEED2 Seeds</b>	<b>48</b>
A.1	SHRiMP Seeds . . . . .	48
A.2	PatternHunter II Seeds . . . . .	53
A.3	BFAST Seeds . . . . .	55
A.4	MegaBLAST Seeds . . . . .	57
	<b>Curriculum Vitae</b>	<b>59</b>

# List of Figures

2.1	A comparison of the helix and base structure of RNA and DNA . . . . .	4
2.2	A comparison of local and global alignment . . . . .	8
2.3	The suffix tree for S=acaaacatat . . . . .	9
2.4	Suffix tree for the sequence S = GAACCGACCT . . . . .	9
2.5	Aligning <i>M.genitalium</i> and <i>M. pneumoniae</i> . . . . .	10
2.6	Partition of D into overlapping blocks of size b . . . . .	12
2.7	Identification of sequence similarities by FASTA. . . . .	14
2.8	Schematic of a BLAST search . . . . .	15
2.9	Hit and Extend Process . . . . .	16
2.10	Spaced seed example . . . . .	16
2.11	Spaced seed vs consecutive seed . . . . .	16
2.12	1 – <i>hit</i> performance of spaced model versus consecutive models . . . . .	17
2.13	Performance of multiple spaced seeds . . . . .	19
2.14	The classical PatternHunter spaced seed designed using Iedera . . . . .	20
2.15	AcoSeed algorithm . . . . .	20
2.16	ACO construction graph for the identification of seed lengths . . . . .	21
2.17	Construction graph and seed building procedure . . . . .	22
3.1	An example of a hit using PatternHunter’s spaced seed . . . . .	26
3.2	An example of the overlap complexity of two seeds . . . . .	27
3.3	SpEED algorithm . . . . .	28
3.4	Example of estimated sensitivity calculation . . . . .	30
3.5	Hit detection for estimating sensitivity . . . . .	31
3.6	Comparison between real and estimated sensitivity . . . . .	31
3.7	Difference between real and estimated sensitivity . . . . .	32
3.8	Comparison between real and estimated sensitivity . . . . .	33
3.9	<i>OC</i> comparison between SpEED-Fast and rasbhari . . . . .	34
4.1	Screenshot of SpEED2 output . . . . .	41



# List of Tables

2.1	An example of Pairwise Alignment . . . . .	5
2.2	An example of Needleman-Wunsch algorithm computation matrix . . . . .	6
2.3	Optimal global alignment discovered . . . . .	7
2.4	An example of Smith-Waterman algorithm computation matrix . . . . .	7
2.5	Optimal local alignment discovered . . . . .	7
2.6	Comparison between QUASAR, QUASAR-E and BLAST . . . . .	12
2.7	Sensitivity comparison of different programs . . . . .	24
4.1	Types of seeds used as dataset . . . . .	40
4.2	Comparison of different programs with SpEED2 . . . . .	42
A.1	SHRiMP seeds of weight 10 and homology length 50 . . . . .	48
A.2	SHRiMP seeds of weight 10 and homology length 50 . . . . .	48
A.3	SHRiMP seeds of weight 10 and homology length 50 . . . . .	48
A.4	SHRiMP seeds of weight 11 and homology length 50 . . . . .	49
A.5	SHRiMP seeds of weight 11 and homology length 50 . . . . .	49
A.6	SHRiMP seeds of weight 11 and homology length 50 . . . . .	49
A.7	SHRiMP seeds of weight 12 and homology length 50 . . . . .	50
A.8	SHRiMP seeds of weight 12 and homology length 50 . . . . .	50
A.9	SHRiMP seeds of weight 12 and homology length 50 . . . . .	50
A.10	SHRiMP seeds of weight 16 and homology length 50 . . . . .	51
A.11	SHRiMP seeds of weight 16 and homology length 50 . . . . .	51
A.12	SHRiMP seeds of weight 16 and homology length 50 . . . . .	51
A.13	SHRiMP seeds of weight 18 and homology length 50 . . . . .	52
A.14	SHRiMP seeds of weight 18 and homology length 50 . . . . .	52
A.15	SHRiMP seeds of weight 18 and homology length 50 . . . . .	52
A.16	PatternHunter II seeds of weight 11 and homology length 64 . . . . .	53
A.17	PatternHunter II seeds of weight 11 and homology length 64 . . . . .	54
A.18	PatternHunter II seeds of weight 11 and homology length 64 . . . . .	54
A.19	BFAST seeds of weight 22 and homology length 50 . . . . .	55
A.20	BFAST seeds of weight 22 and homology length 50 . . . . .	55
A.21	BFAST seeds of weight 22 and homology length 50 . . . . .	56
A.22	MegaBLAST seeds of weight 28, homology length 100 and sensitivity of 0.9 . . . . .	57
A.23	MegaBLAST seeds of weight 28, homology length 100 and sensitivity of 0.9 . . . . .	57
A.24	MegaBLAST seeds of weight 28, homology length 100 and sensitivity of 0.9 . . . . .	57
A.25	MegaBLAST seeds of weight 28, homology length 100 and sensitivity of 0.9 . . . . .	58
A.26	MegaBLAST seeds of weight 28, homology length 100 and sensitivity of 0.9 . . . . .	58

# List of Appendices

Appendix A . . . . .	48
----------------------	----

# Chapter 1

## Introduction

The main focus of bioinformatics is to interpret and analyze biological data using mathematical and statistical models. The National Center for Biotechnology Information (NCBI) GenBank currently contains nucleotide sequences for more than 250,000 organisms with supporting bibliographic and biological annotation. The GenBank has been growing exponentially since 1982 and is still growing [1].

There is a need to comb through such a vast repository of data in order to find sequences of interest. If some new unknown sequences are encountered, it is wise to compare those new sequences to similar sequences (from the repository) with known functionality. This will give a lot of insight into the new sequence.

Sequence similarity search is the most common task in bioinformatics. Many applications like gene and protein predictions, sequence assembly, evolutionary and phylogeny study etc. are dependant on sequence similarity search. From a theoretical point of view, the problem of sequence similarity search is complex and is based on the concept of sequence alignment.

More than 4 million web users access the NCBI website daily [2], even if a small fraction of this many users are searching the database, that is still a lot of searches. Traditional approaches of sequence alignment like Needleman-Wunsch and Smith-Waterman algorithms use dynamic programming and are able to find the optimal alignment between two sequences. However these algorithms have quadratic run time (product of the lengths of the two sequences), hence they are not feasible when long sequences are involved. So, heuristic approaches are needed, which trade sensitivity for speed but might miss some true alignments. Seeding based approach is one of the most popular heuristic methods used like FASTA [3], BLAST [4] and PatternHunter [5].

The Basic Local Alignment Search Tool (BLAST) is the most used tool for similarity search. In the case of nucleotide sequences, BLAST finds an exact match of eleven consecutive letters between the two sequences and treats this as a hit. Next, the identified hits are extended greedily in both directions to find HSP or High-Scoring Sequence Pair.

The spaced seed, which was introduced in PatternHunter enhances the consecutive seed of BLAST. This is because the hits detected by spaced seeds are more distributed whereas consecutive seed tends to detect hits in clusters, as a result, spaced seeds do not generate redundant hits, thus detecting more similar regions than consecutive seeds. By using optimized spaced seeds, we can improve the sensitivity drastically and achieve results in time similar to BLAST. Multiple spaced seeds were then introduced in PatternHunter II [6].

It has been seen that well-designed multiple spaced seeds can even achieve sensitivity that is close to that observed for the Smith-Waterman algorithm and run at speeds similar to the BLAST algorithm. Thus, algorithm, which design good multiple spaced seeds are an active area of research.

## 1.1 Motivation

Similarity search based on the strategy of hit and extend is ubiquitous in genomics and proteomics. Here, two biological sequences, such as DNA or protein sequences are aligned by finding short seed matches called hits, which are then extended to high-scoring segment pairs or HSPs.

Optimal spaced seed, which was introduced in PatternHunter [5] by Ma et al., has increased both the sensitivity and the speed of similarity search. Further improvement was achieved by using multiple spaced seeds in PatternHunterII [6] by Li et al. With the use of multiple spaced seeds along with the hit and extend approach, heuristics methods are getting close to Smith-Waterman sensitivity at BLASTn speeds.

However, computing optimal multiple spaced seeds used by PatternHunterII has proven to be NP-hard [7] and current heuristic algorithms can be improved.

## 1.2 Problem Statement

Computing optimal multiple spaced seeds is NP-hard. So this problem is solved using heuristic algorithms. The drawback of these heuristic approaches is that they are either fast and less sensitive or are sensitive and very slow. There is a need for some approach that will produce seeds of high sensitivity in an acceptable time. Our approach aims to alleviate this problem.

## 1.3 Objective

The main objective is to design multiple spaced seeds of high sensitivity within an acceptable amount of time. Using these seeds, similarity search is possible and similar regions often correspond to homologous regions.

## 1.4 Thesis Contribution

The major contribution of this thesis is that our novel approach generates good multiple spaced seeds by modifying the lengths and composition of the seeds. We have implemented SpEED2, which embodies our approach and is capable of producing multiple spaced seeds of high sensitivity.

Various tools in bioinformatics depend on good seeds to solve their respective problems. Problems concerned with gene and protein predictions, phylogeny and evolutionary analysis, read mapping, primer design, and probe design which uses multiple spaced seeds will have increased performance by using multiple spaced seeds designed by SpEED2.

## 1.5 Thesis Outline

In **Chapter 2**, we discuss multiple spaced seeds in detail. We talk about sequence alignment and various implementations using dynamic programming and heuristic methods, their pros and cons.

Then we look at leading software used for similarity search.

After this, we introduce multiple spaced seeds along with prominent works and leading programs.

In **Chapter 3**, we discuss SpEED, followed by our approach and its implementation SpEED2 which is built on top of SpEED. Then we discuss its important algorithms followed by the entire algorithm as a whole.

In **Chapter 4**, we present the experimental results produced by our software program SpEED2 and also compare these results with other leading software like Iedera, AcoSeed and rasbhari.

In **Chapter 5**, we summarize our work. Finally, we present some possible future works.

# Chapter 2

## Background

A biological sequence is a single, continuous molecule of nucleic acid or protein. It can be thought of as a multiple inheritance class hierarchy. One hierarchy is that of the underlying molecule type: DNA, RNA, or protein. The other hierarchy is the way the underlying biological sequence is represented by the data structure. It could be a physical or genetic map, an actual sequence of amino acids or nucleic acids, or some more complicated data structure building a composite view from other entries [8].

In simpler terms, biological sequences are one-dimensional series of nucleotides or amino acids, which are naturally occurring monomers. Polymerization of nucleotides result in nucleic acids, which are biological molecules of DNA (deoxyribonucleic acid) and RNA (ribonucleic acid) as shown in 2.1. An amino acid is another naturally occurring monomer that polymerizes to form a protein. All these are essential building blocks of life on Earth.

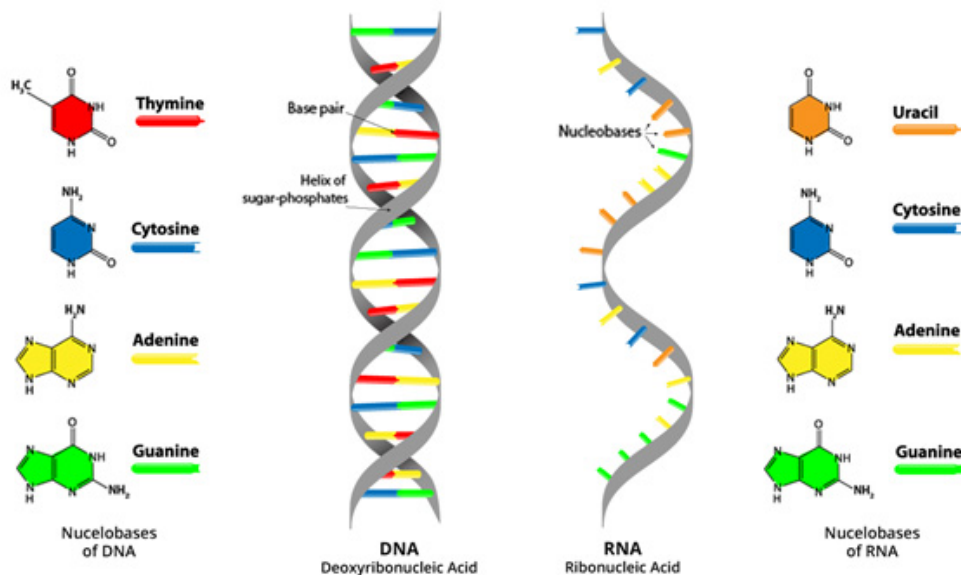


Figure 2.1: A comparison of the helix and base structure of RNA and DNA  
Four types of nitrogen bases found in DNA are adenine (A), thymine (T), guanine (G), and cytosine (C). RNA contains uracil (U) instead of thymine (T). [9]

Sequence analysis of unknown nucleic acids and proteins are crucial in understanding the evolution, function, and structure of organisms. The Human Genome Project [3] identified more than 20,500 genes. Unknown sequences are a result of evolution from existing sequences, hence it is common practice to perform sequence analysis, comparing the unknown to the already known sequences present in the repository. Thus, sequence alignment is a critical step while finding similar regions between two sequences.

## 2.1 Sequence Alignment

Two or more sequences inherited from a common ancestor are said to be homologous. We search for similar sequences in the hope of finding homologous sequences. Sequence alignment helps us determine whether two or more sequences descended from a common ancestor. It also helps us infer a common function by locating functional elements. One important point to keep in mind is that the ultimate goal of bioinformatics is to find the "real" alignment of a sequence according to real evolution. However, this is impossible by sequence alone, thus we are forced to find the "optimal" alignment instead. Optimal alignment and real alignment might be different. The earliest attempts to computationally align sequences were made by using edit distance [10] in 1966. Edit distance is the minimum number of edit operations it takes to convert one sequence to another. Three types of operations - insertion, deletion, and substitution are present to facilitate the conversion.

A great deal of literature was developed following that. Some of the most prominent work includes Needleman-Wunsch's dynamic-programming solution for global alignment [11], the dynamic-programming solution for local alignment by Smith-Waterman [12], the BLAST tool for searching similar regions using length 11 consecutive seed [4]. The PatternHunter introduced spaced seeds [5], and PatternHunter II introduced the use of multiple spaced seeds [6]. The focus shifted towards designing of optimized multiple spaced seeds as evident from SpEED [7, 13, 14] and numerous other works like iedera [15, 16, 17], AcoSeed [18], rasbhari [19], etc. We shall discuss more in the coming sections, but first, let us discuss some basic concepts regarding sequence alignment.

## 2.2 Traditional Methods

Sequence alignment means arranging two sequences so that they are most similar to each other column-wisely according to some scoring function. Let us look at the following example with two sequences  $S_1 = \text{AATGCATT}$  and  $S_2 = \text{GTGATT}$ , where  $\text{match\_score} = +1$  and  $\text{mismatch\_score} = \text{indel\_score} = -1$ . We assume this simple scoring system to illustrate the process.

A	A	T	G	C	A	T	T
G	-	T	G	-	A	T	T

Table 2.1: An example of Pairwise Alignment

The pairwise alignment between sequence  $S_1$  and sequence  $S_2$ . It shows three different possible alignments for each position - match, mismatch and indels.

The blue letters indicate match positions in Table 2.1. The alignment score is  $+5-3 = 2$  and this is the optimal alignment. Indel is a molecular biology term for an insertion or a deletion of bases in the genome of an organism due to genetic variation over time. In Table 2.1, the second column (containing A and -) represents an indel and the - represent a gap. Global and local alignments are two main types of alignment and their algorithms are discussed below.

### 2.2.1 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm uses dynamic programming to find global alignment [11]. The main idea behind this algorithm is that the optimal alignment of the complete sequence can be found by optimally aligning component sub-parts. In global alignment, the optimal path must stretch from beginning to end in both the sequences.

To perform a Needleman-Wunsch alignment, a matrix is created to store the score  $M(i, j)$ , which is calculated as follows:

$$M(i, j) = \max \begin{cases} M_{i-1,j-1} + S(A_i, B_j) \\ M_{i-1,j} + gap \\ M_{i,j-1} + gap \end{cases}$$

Here *gap* is the gap penalty and function *S* returns the score for match/mismatch between two letters  $A_i$  and  $B_j$ . A matrix is filled using the above-mentioned recurrence relation, which states that the value of any cell is the maximum among the mentioned three options ( $M_{i-1,j-1}$  is the neighboring cell in the upper-left diagonal to the current cell,  $M_{i-1,j}$  is the cell above the current cell and  $M_{i,j-1}$  is the cell to the left of the current cell). Once the matrix is filled, a trace-back is performed to determine the sequence of operations that lead to the final score from the start. This algorithm has a time complexity of  $O(mn)$  and space complexity of  $O(mn)$ , where  $m$  and  $n$  are the lengths of the two sequences.

Let's look at an example. We assume the two sequences and scoring system as follows. Given  $S_1 = \text{CTTAAG}$ ,  $S_2 = \text{ATTAG}$ . Match\_score = 2, mismatch\_score = -1 and indel\_score = -2. We assume this simple scoring system to illustrate how the algorithm works.

		A	T	T	A	G
	0	-2	-4	-6	-8	-10
C	-2	-1	-3	-5	-7	-9
T	-4	-3	1	-1	-3	-5
T	-6	-5	-1	3	1	-1
A	-8	-4	-3	1	5	3
A	-10	-6	-5	-1	3	4
G	-12	-8	-7	-3	1	5

Table 2.2: An example of Needleman-Wunsch algorithm computation matrix

The matrix shown in Table 2.2 produces the global alignment shown in Table 2.3 with a global alignment score of 5.



C	T	T	A	A	G
A	T	T	-	A	G

Table 2.3: Optimal global alignment discovered

### 2.2.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm uses dynamic programming to find optimal local alignments [12]. The Needleman-Wunsch algorithm was slightly modified to achieve local alignments. Here, the alignment path may start and end internally, it does not need to reach the edges of the matrix. To perform a Smith-Waterman alignment, a matrix is created to store the score  $M(i, j)$ , which is calculated as follows:

$$M(i, j) = \max \begin{cases} M_{i-1, j-1} + S(A_i, B_j) \\ M_{i-1, j} + gap \\ M_{i, j-1} + gap \\ 0 \end{cases}$$

This means that zero is the lowest value possible in the scoring matrix and all other components of the recurrence relation remain the same as the Needleman-Wunsch algorithm, negative numbers are not allowed as in global alignment. This algorithm has a time complexity of  $O(mn)$  and space complexity of  $O(mn)$ , where  $m$  and  $n$  are the lengths of the two sequences.

Local alignments are performed everywhere possible along two sequences. Let's look at an example. We assume the two sequences and scoring system as follows. Given  $S_1 = \text{GAT-ACCTTG}$ ,  $S_2 = \text{AATAGTCT}$ . Match\_score = 2, mismatch\_score = -1 and indel\_score = -2. We assume this simple scoring system to illustrate how the algorithm works.

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

Table 2.4: An example of Smith-Waterman algorithm computation matrix

G	A	T	A	C	T
A	A	T	A	G	T

Table 2.5: Optimal local alignment discovered

The matrix shown in Table 2.4 produces the local alignment shown in Table 2.5 with a local alignment score of 7.

Local alignments are performed everywhere possible along two sequences as shown in Figure 2.2.

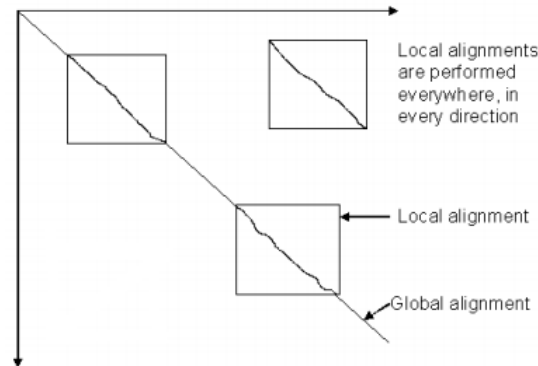


Figure 2.2: A comparison of local and global alignment

The global alignment stretches from the beginning till the end between the two given sequences, whereas, local alignment detects smaller matches between the sequences and need not stretch from the beginning till the end. [20]

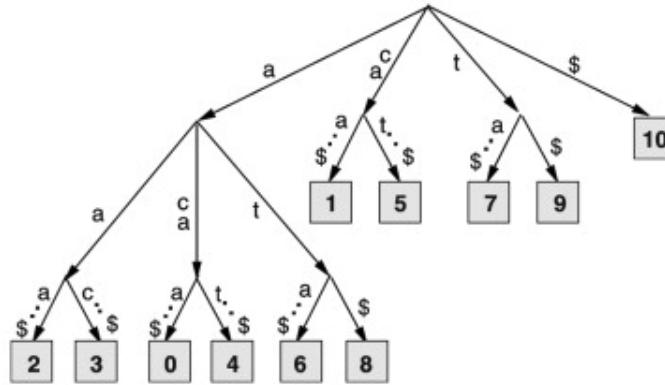
## 2.3 Heuristic Methods

A heuristic algorithm is designed to solve a particular problem faster than traditional methods by trading accuracy or precision for speed. The traditional methods require quadratic run-time and memory. We have already discussed in the previous chapter why this is unacceptable, thus the need for heuristic methods. We shall discuss two common strategies of similarity search: suffix trees and seeding.

### 2.3.1 Suffix Tree Algorithms

Given a string  $S$  of  $n$  characters, a suffix tree for  $S$  is a rooted directed tree with exactly  $n + 1$  leaves numbered 0 to  $n$ . Each internal node, other than the root, has at least two children and each edge is labeled with a non-empty substring of  $S$  and no two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf  $i$ , the concatenation of the edge-labels on the path from the root to the leaf  $i$  exactly spells out the string  $S_i$ , where  $S_i = S[i..n - 1]$  denotes the  $i$ -th non-empty suffix of the string  $S$ ,  $0 \leq i \leq n$  [21]. Figure 2.3 shows an example of suffix tree.

A suffix tree quickly detects all unique sub-sequences for a given sequence because all unique sub-sequences are associated with a leaf in a suffix tree, thus, it can be used to align long DNA sequences. Weiner [22] and McCreight [23] gave us very efficient algorithms for constructing suffix trees. Using Ukkonen's algorithm [24], a suffix tree can be constructed in linear-time and searching can also be done in linear time. There are two main disadvantages of using suffix trees for sequence alignments: they perform well on highly similar sequences

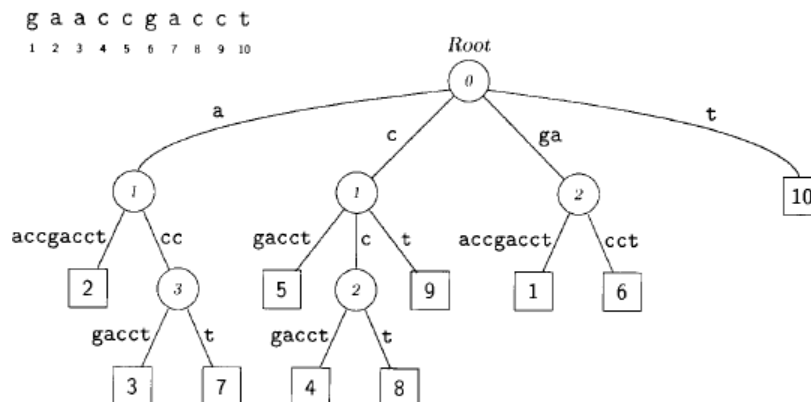
Figure 2.3: The suffix tree for  $S = \text{acaaacatat}$ 

Traversing from the root to any leaf produces a complete suffix of the pattern  $S$ . Any valid suffix must end with a \$ symbol. [21]

but their performance degrades when mismatches are introduced i.e.- they are very poor in handling mismatches. The other disadvantage is that they require a lot of memory. Here, we discuss two suffix tree-based heuristic algorithms - MUMmer and Quasar.

### 2.3.1.1 MUMmer

MUMmer is a software system for aligning long sequences of DNA that have high similarity. The system takes three inputs: two DNA sequences of high similarity - *Genome\_A* and *Genome\_B* as well as the length of the shortest MUM the system will identify. A MUM is a subsequence that occurs exactly once in *Genome\_A* and once in *Genome\_B* and is not contained in any longer sequence. For highly similar sequences, the shortest MUM length is 50 bp, for a less similar sequence, it is 20 bp. In the output, exact matches and all single nucleotide polymorphisms (SNPs) are identified as well as significant repeats [25].

Figure 2.4: Suffix tree for the sequence  $S = \text{GAACCGACCT}$ 

Travelling from the root to any leaf will produce a suffix of  $S$ . If we consider the left most leaf (leaf 2), the suffix the path represents is *accgacct*. [25]

MUMmer is built using three component ideas: suffix trees as shown in Figure 2.4, the longest increasing sub-sequence (LIS) and Smith-Waterman alignment. All three components are tightly coupled into a single system. The alignment process consists of the following steps:

- **Maximal Unique Matching (MUMs) sub-sequence decomposition :** The most important step in the alignment process of MUMmer is to identify MUMs. When the program is run, a suffix tree  $T$  is constructed for *Genome\_A* and the suffixes of *Genome\_B* are appended to  $T$ . All leaf nodes indicate whether it is suffix from *Genome\_A* or *Genome\_B*. All MUMs can be discovered in just one run over the suffix tree  $T$ .

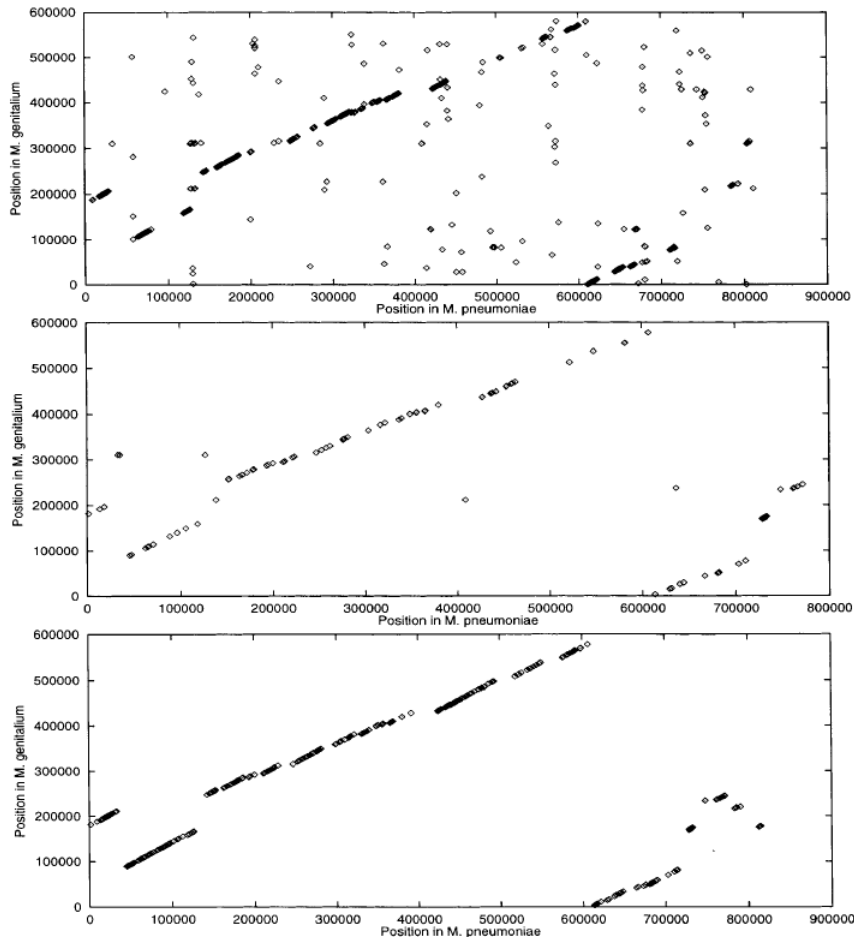


Figure 2.5: Aligning *M.genitalium* and *M. pneumoniae*

The top figure is the alignment produced by FASTA, the middle one by 25mer and the bottom one by Maximal Unique Matching. The MUM alignment is the cleanest and is continuous among all three alignments. [25]

- **Sorting the MUMs :** Once all the MUMs are found, they are sorted according to their position in *Genome\_A*. Then, the corresponding matching positions are ordered in *Genome\_B*. A variant of LIS algorithm [26] finds the longest set of MUMs that are in ascending order for both the sequences.

- **Closing the gaps :** After the global MUM-alignment is discovered, several algorithms are used to complete the alignment by closing the local gaps. A gap is any interruption present in MUM alignment which might occur because of SNP interruption, an insertion or a repeat.

Figure 2.5 illustrates the alignment at the level of complete genome. In all three figures, *M.genitalium* and *M. pneumoniae* are aligned using FASTA (top), 25mers (an oligonucleotide of 25 nucleotides - middle) and MUMs (bottom). The MUM alignment is the cleanest and is continuous among all three alignments. The MUM alignment also clearly shows five translocations of *M.genitalium* and *M. pneumoniae*, which is in agreement with the work of Himmelreivh *et al.* [27]. Translocation means a change in location and in genetics, it refers to the phenomenon when part of a chromosome is transferred to another chromosome. This is certainly not evident from the other two alignments.

### 2.3.1.2 Quasar

QUASAR or Q-gram Alignment based on suffix arrays [28] is a similarity search tool which is capable of quickly detecting sequences that are highly similar to the sequence that is being queried. It uses a suffix array to index the whole database and applies modified  $q$ -tuple filtering on the suffix array. QUASAR can be run in two modes - a RAM resident suffix array and a disk resident suffix array. Since this program works well when both the sequences have high similarity, QUASAR is a good fit when searching Expressed Sequence Tags (EST) databases since they are derived from the same gene. ESTs are small pieces of DNA sequence (usually a few 100 bp long) and they represent genes expressed in certain cells. ESTs are commonly used to identify unknown genes and map their positions within a genome.

It has a memory requirement of  $5|D|$  (5 times the size of whole database) and a running time complexity of  $O(|D|\log|D|)$  during the preprocessing step of suffix array creation and  $O(|S| \cdot |D|)$  while searching for a specific  $q$ -gram.  $|D|$  is the length of the whole database and  $|S|$  is the length of the query sequence.

Let us now look at some of the key aspects of QUASAR. The algorithm for determining all sequences in a database  $D$  that have a local similarity to a query sequence  $S$ . We say that a sequence  $d \in D$  is locally similar to  $S$ , if there exists at least one pair  $(S[i : i + w - 1], d')$  of sub-strings with the following properties:

- The query sequence is  $S$  and the database to be searched is  $D$ . A sequence  $d \in D$  is locally similar to  $S$  if
- $d'$  is a sub-string of  $d$ ,  $S[i : i + w - 1]$  is a sub-string of  $S$  of length  $w$ , where  $d'$  and  $S[i : i + w - 1]$  have edit distance at most  $k$ .

We will now discuss in high-level the algorithm used here.

- **Suffix Array as Index Data Structure :** Use suffix array for all  $q$ -grams( $Q$ ) in  $D$  (known as hitlist). Search for  $S$  against all  $Q$ . Searching  $S$  against whole  $D$  is avoided and only  $Q$  (a small portion of  $D$ ) is searched.

- **Block Addressing :**  $D$  is divided into blocks of size  $b$  ( $b \geq 2w$ ) and each block assigned is numbered as shown in Figure 2.6. The block numbers are incremented whenever  $q - gram$  is found in the block. All blocks having more than equal to  $t$  numbers are checked for an approximate match using an alignment algorithm like BLAST.

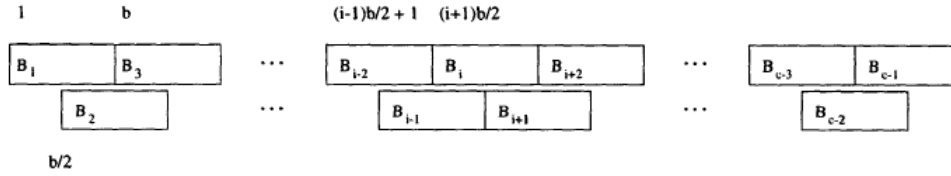


Figure 2.6: Partition of  $D$  into overlapping blocks of size  $b$

The figure illustrates the block addressing process of Quasar. The database  $D$  is divided into smaller blocks of size  $b$  and are numbered. [28]

- **Window Shifting and Alignment :** Using the above two steps, all approximate matches can be found for the first window  $S[l : w]$ .  $S[2 : w + 1]$  is the next window,  $S[l : q]$  is the old  $q - gram$  and  $S[w - q + 2 : w + 1]$  is new  $q - gram$ . Then counter where, value is not  $t$ , of all blocks having copies of this specific  $q - gram$  is decremented. All occurrences of the new  $q - gram$  are found using suffix array and the corresponding block counters are incremented. The window, having length  $w$ , is shifted over the string  $S$  until its end.

DB	Size Mbps	Queries Size (bps)	Identical Results	False Neg E-value	Filtration Ratio	CPU time (s) QUASAR	CPU time (s) QUASAR-E	CPU time (s) BLAST
Mouse	73.5	368	91.4%	$10^{-12}$	0.24%	0.123	0.128	3.37
Human	279.5	393	97.1%	$10^{-14}$	0.17%	0.38	0.39	13.27

Table 2.6: Comparison between QUASAR, QUASAR-E and BLAST [28]

Table 2.6 shows sensitivity and running times when searching mouse and human EST databases with block size of 1024 bps. Query sequences of sizes 368bps and 393bps were searched in the Mouse and Human EST databases. QUASAR searched through 73.5 Million base pairs (Mbps) and 279.5 Mbps in 0.12s and 0.38s respectively. QUASAR-E with the suffix array on the disk achieves nearly the same performance with 0.13 s and 0.39 s. Both QUASAR and QUASAR-E are approximately 30 times faster than BLAST. From left to right the table depicts the database, its size, average query length, identical results search percentage, average  $E - values$  of the first missed sequence, the filtration ratio and lastly CPU times of QUASAR, QUASAR-E and BLAST.

### 2.3.2 Seeding Based Algorithms

The seed-based approach is built upon the idea of filtration, where the alignment of a pair of sequences is obtained by first detecting short identical segments of exact matches, called hits, and then extending these matches greedily on both the sides for approximate matches, which

results in regions known as high-scoring segment pair (HSP). In essence, these algorithms rely on hit and extension of small exact matches. In this section, we will discuss four seed-based heuristic approaches - FASTA, BLAST, PatternHunter and finally PatternHunter II.

### 2.3.2.1 FASTA

In 1985, FASTAP [29] was released that supported only protein to protein similarity search. Later, in 1988, DNA to DNA similarity search along with other features was added to the package and it was renamed as FASTA [3]. The FASTA package was the first product that utilized seeding algorithms for similarity search against a database. It uses a short sequence of all 1s of length  $k$  to find exact matches between the query sequence and database sequence and then the Smith-Waterman algorithm is performed. Thus,  $k$  value determines the sensitivity and speed of the search, and it is a trade-off. The higher the  $k$  value faster the search and the smaller the  $k$  value, the more sensitive is the search. However, with a smaller  $k$  value, there is a high chance of detecting regions that are false hits. The default value of  $k$  for DNA is chosen to be 5 or 6. The search algorithm uses four steps to determine the pairwise similarity score. Let us now look at them.

- A look-up table stores the positions of all  $k$  - *tuple* of the query sequence. A  $k$  - *tuple* is string of  $k$  consecutive letters. The database sequence is read, one  $k$  - *tuple* at a time and searched in the lookup table - a match that is found is known as hit. The 10 best diagonal regions are discovered using a simple formula based on the number of  $k$  - *tuple* matches and the distance between the matches as depicted in Figure 2.7 (A).
- All the 10 regions from the previous step are scored using a scoring matrix and the best initial regions are stored. The returned regions are High Scoring Sequence Pairs (HSPs) as depicted in Figure 2.7 (B).
- In the third step, a joining procedure is used to merge several HSPs (having a similarity score greater than a threshold value) to form a single optimal alignment as in Figure 2.7 (C). A modified version of Smith-Waterman algorithms is run to find the optimal alignments within the window of diagonals discovered previously which are the maximum gap lengths.
- After the final alignment is found, FASTA analyzes the result in order to distinguish it from the random alignments. The dotted lines indicate the optimized alignment bounds and optimal score is calculated based on this alignment - Figure 2.7 (D).

### 2.3.2.2 BLAST

BLAST or Basic Local Alignment Search Tool [4] was introduced in 1990 for quickly finding a match to a query sequence in the database. BLAST identifies alignment between two sequences by identifying hits (matching  $k$  consecutive letters, called  $k$  - *mer*), followed by a more time-consuming process of hit extension. BLAST then provides a statistical measure of the significance of the alignment detected to prove that it is not any random alignment.

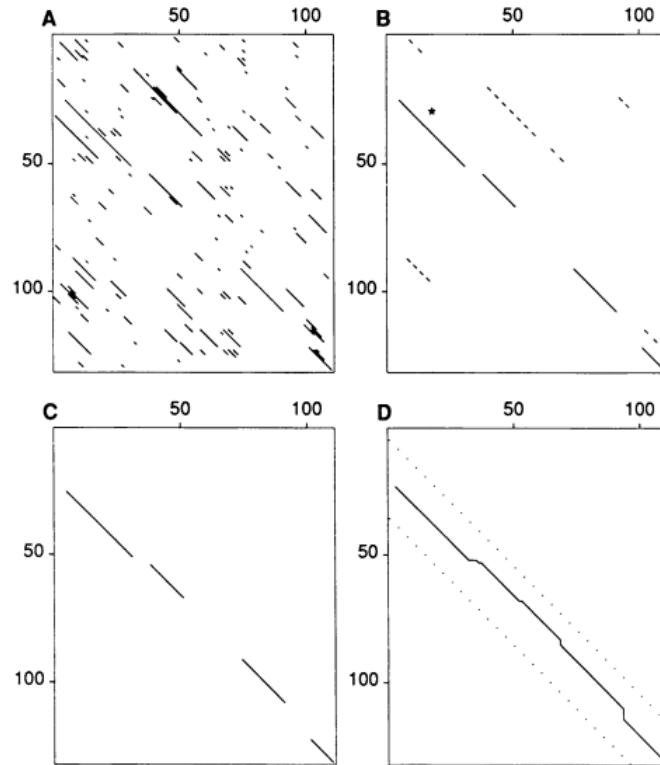


Figure 2.7: Identification of sequence similarities by FASTA.

The four steps used by the FASTA program to calculate the initial and optimal similarity scores between two sequences are shown. (A) Identify regions of identity. (B) Scan the regions using a scoring matrix and save the best initial regions. Initial regions with scores less than the joining threshold (27) are dashed. The asterisk denotes the highest scoring region reported by FASTP. (C) Optimally join initial regions with scores greater than a threshold.

The solid lines denote regions that are joined to make up the optimized initial score. (D) Recalculate an optimized alignment centered around the highest scoring initial region. The dotted lines denote the bounds of the optimized alignment. The result of this alignment is reported as the optimized score. [3]

The first version of BLAST was a standalone application and lacked many important features [30]. Later, in 1996, BLAST became a web service with added capabilities [31]. The latest release of BLAST [30] in 2009 started using non-consecutive seed (spaced seed) for DNA comparison. Today, BLAST is a family of several programs including gapped BLAST [32], MegaBLAST [33] etc.

Figure 2.8 shows the key steps of BLAST. We shall look at them here.

- **Setup (Hit generation) :** First, a scoring system is decided for a match, mismatch, and indels. The query sequence is broken into  $k - mer$ , which are strings of length  $k$ . In the case of nucleotide-to-nucleotide search,  $k$  is 11 and for protein-to-protein search, it is 3. The starting index of all  $k - mers$  is stored into a lookup table. BLAST then scans the database looking for exact matches between the indexed  $k - mers$  and strings found



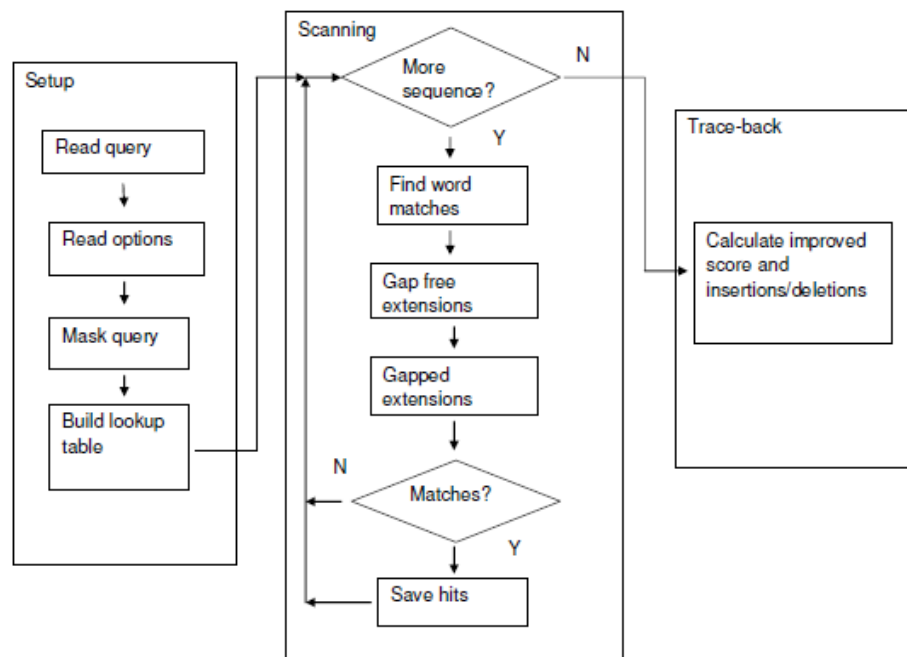


Figure 2.8: Schematic of a BLAST search

The first phase is "setup". The query is read, low-complexity or other filtering might be applied to the query, and a "lookup" table is built. The next phase is "scanning". Each subject sequence is scanned for words ("hits") matching those in the lookup table. These hits are further processed, extended by gap-free and gapped alignments, and scored. Significant "preliminary" matches are saved for further processing. The final phase in the BLAST algorithm, called the "trace-back", finds the locations of insertions and deletions for alignments saved in the scanning phase. [30]

within the database sequences (not strictly true for proteins). A match found is known as a hit.

- **Scanning (Hit Extension) :** When a hit occurs, BLAST attempts to extend the hit region in both forward and backward direction to produce an alignment. BLAST will continue this extension process as long as the alignment score keeps on increasing or until the alignment score drops by a critical amount (known as dropoff) because of mismatch or indels. The alignment formed is called a Highest-Scoring Segment Pair (HSP).
- **Statistical Significance :** Each alignment found by BLAST is assigned a statistical value, called Expect Value or *E – value*. The *E – value* is the number of times an alignment is better than an alignment found by random chance. A higher *E – value* threshold is less stringent and thus capable of returning a randomly matched sequence. Sequences with *E – value* less than  $1e-04$  can be considered homologous with an error rate of less than 0.01%.

Figure 2.9 shows how Blast consecutive weight-11 seed performs hit and extend operation



Figure 2.9: Hit and Extend Process

Hit detected (blue region) by weight-11 consecutive BLAST seed. Hit region is extended greedily in both directions to form high sequence pair (HSP).

on two random DNA sequences.

### 2.3.2.3 PatternHunter

PatternHunter [5], introduced in 2002, implements a novel seeding and hit-processing algorithm that generates a hit in a similar region with high probability while having a low expected number of random hits at the same time. PatternHunter introduced the use of spaced seed, which is much more sensitive than the consecutive BLAST seed. A spaced seeds can be defined as a binary string of 0s and 1s. 1 represents a match position whereas a 0 represents a don't care position and the seeds must start and end with 1. The length of a spaced seed is the number of characters in it and the weight of the seed is the number of 1s. Let us look at an example of a spaced seed.

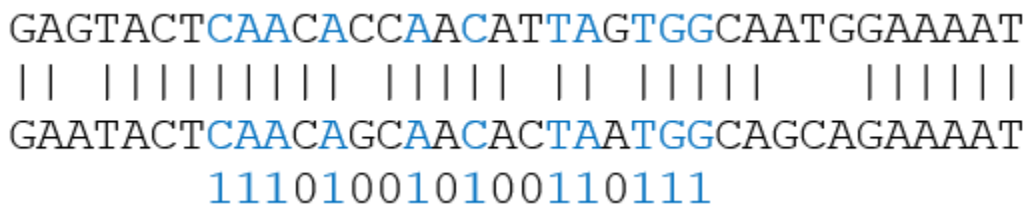


Figure 2.10: Spaced seed example

Hit detection by weight-11 spaced seed. When two sequences are aligned, a hit occurs if all bases match in the corresponding match positions (1s) of the seed. The don't care positions (0s) need not match.

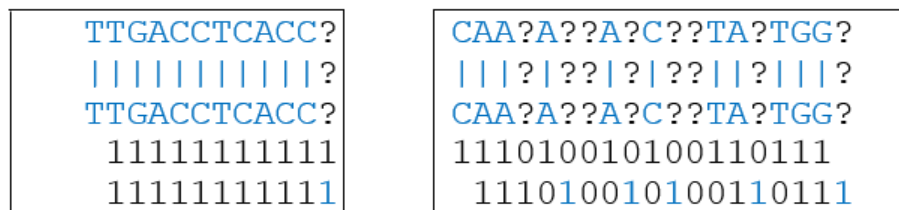


Figure 2.11: Spaced seed vs consecutive seed

When using a consecutive seed, many redundant hits are required to detect a similar region. In the case of spaced seeds, lesser number of hits are required to detect a similar region. Thus, spaced seed is able to discover regions which the consecutive seed is unable to.

The spaced seed in Figure 2.10 is  $111 * 1 ** 1 * 1 ** 11 * 111$ . The length of this seed is 18 and the weight is 11. 1 represents match position and \* represent don't care position. We can see in the above figure that the seed hits a particular region (shown in blue) only when the two sequences match at all positions of 1s, however, the don't care positions are not considered.

In Figure 2.11, we see a consecutive seed and a spaced seed in action. From PatternHunter [5], we get to know why spaced seed models are more effective. The total number of hits detected by both seed models are approximately equal. Also, from Figure 2.11 we see that a consecutive seed usually uses more than one hit to detect one similar region (redundant hits). Spaced seeds use fewer hits to detect the same, thus spaced seeds discover new undetected regions.

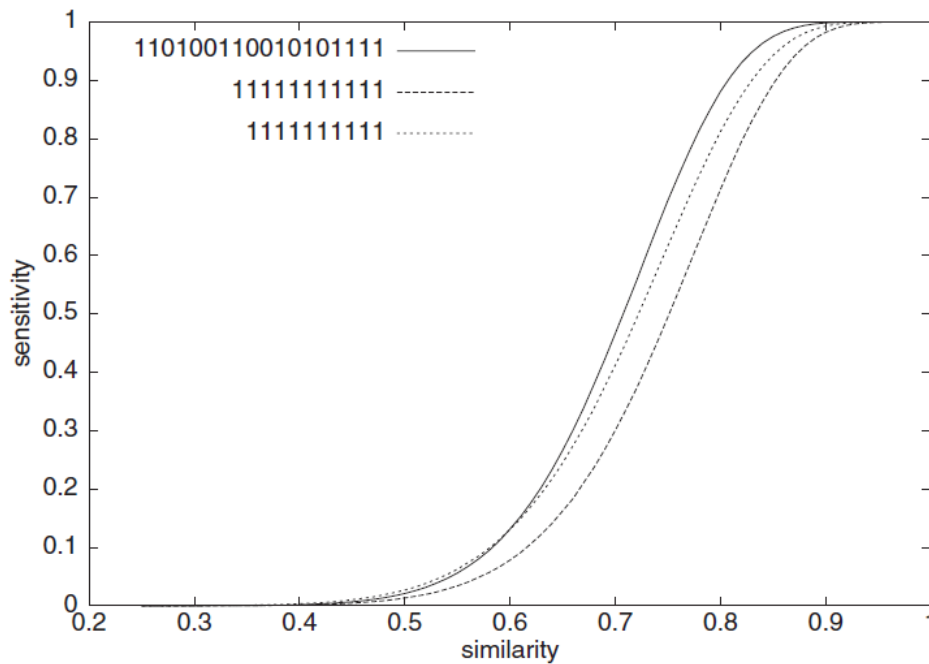


Figure 2.12: 1 – hit performance of spaced model versus consecutive models

The figure shows 1 – hit performance of *weight* – 11 spaced seed model versus *weight* – 11 and *weight* – 10 consecutive seed models. Similarity is on the X-axis and sensitivity is on the Y-axis. Similarity shows how identical the two sequences are and sensitivity is the probability of detecting a hit. [5]

We shall look at the PatternHunter algorithm. The main steps of the algorithm are as follows.

- **Hit generation :** A lookup table is maintained with the starting index position of all the sub-sequences of the first sequence. Next, the second sequence is aligned with the spaced seed and the starting position of all alignments is looked up in the lookup table. If found in the table, a hit occurs. For each hit, PatternHunter looks up its diagonal in another hashtable, to find the rightmost matched position on the diagonal.

- **Hit extension :** In the second step, the hits discovered in the previous step are greedily extended, stopping only when the alignment score drops by a predetermined value. Resulting regions formed after this process is known as a Highscoring Segment Pair (HSP).
- **Gapping extension :** In this step, all HSPs are extended to the left across gaps to link with another HSP (if any suitable HSP is found). This step produces optimal partial alignment and partial alignment score is calculated. Joining optimal partial alignment will result in optimal alignment.

Figure 2.12 shows 1-hit performance of *weight* – 11 spaced seed model versus *weight* – 11 and *weight* – 10 consecutive seed models. We see that spaced seed of *weight* – 11 outperforms both *weight* – 11 and *weight* – 10 Blast seed. The plot has similarity on X-axis and sensitivity on Y-axis. Similarity shows how identical the two sequences are whereas sensitivity shows the probability of detecting a hit.

#### 2.3.2.4 PatternHunter II

PatternHunter II [6] uses optimized multiple spaced seed instead of a single spaced seed to detect similar regions between two sequences. Given a homologous region of fixed length and similarity, we can increase the probability of getting hits in two ways - either decrease the weight of seed or we can use more than one seed. Reducing the weight by one increases the expected number of hits by a factor of four (in case of nucleotide sequence). Doubling the number of seeds increases the expected number of hits by a factor of two. Another way to increase the probability of hits of seeds is to design better seeds (this will be discussed in the next section).

While using multiple spaced seeds, a hit occurs if a region is hit by at least one seed. PatternHunter II works in a similar way to PatternHunter, the major difference is that it uses multiple hash table, one for each seed, instead of one. For each sub-string of the query sequence, all hits generated from all tables are used for gapped extensions (process discussed in previous section). PatternHunter II achieves sensitivity close to that of Smith-Waterman at speeds comparable to MegaBLAST.

Figure 2.13 shows the performance of the multiple spaced seeds. From low to high, the solid curves are the hit probabilities of using 1, 2, 4, 8 and 16 *weight* – 11 spaced seeds, respectively whereas the dashed curves are the hit probabilities of using the single optimal spaced seed of weight 10, 9, 8 and 7 seeds, respectively.

## 2.4 Designing Multiple Spaced Seeds

We have already established the fact that selecting good multiple spaced seeds are paramount to seed-based heuristic search methods which we discussed in the previous section. In this section, we will look at software like Iedera, SpEED, AcoSeed, and rasbhari. These software tools produce multiple spaced seeds of high sensitivity and shall discuss each of them in more detail.

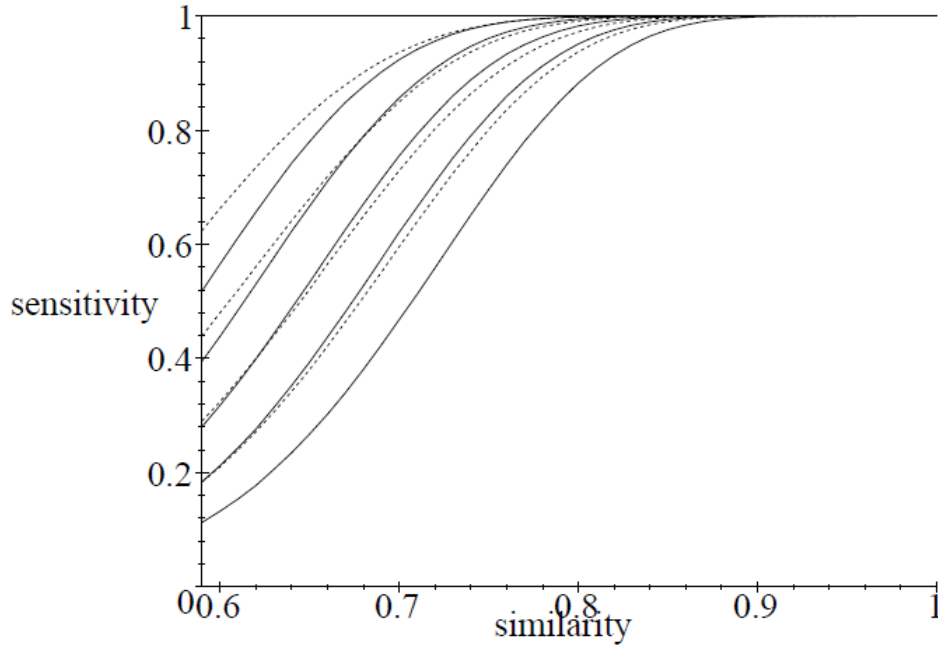


Figure 2.13: Performance of multiple spaced seeds

From low to high, the solid curves are the hit probabilities of using 1, 2, 4, 8 and 16 *weight* – 11 spaced seeds, respectively whereas the dashed curves are the hit probabilities of using the single optimal spaced seed of weight 10, 9, 8 and 7 seeds, respectively. [6]

### 2.4.1 Iedera

Iedera is a tool to select and design spaced seeds, transition constrained spaced seeds, or more generally subset seeds, and vectorized subset seed patterns. [15, 16, 17]. Spaced seeds can be perfectly represented using the subset seed model and subset seed is an extension of spaced seeds that deal with a non-binary alignment alphabet and, on the other hand, still allows an efficient hashing method to locate seeds.

Iedera can be applied to both lossy and lossless seed design. It is already used to design spaced seeds templates for read mappers, to design subset seed templates for protein sequences or nucleic sequences.

In Iedera, a spaced seed is a string over the binary alphabet  $\{\#, -\}$ , where  $\#$  is a match position and  $-$  is a don't care position and the spaced seed is represented using  $\pi$ . Using these notations, a spaced seed  $\pi \in \{\#, -\}^s$  hits an alignment  $A \in \{0, 1\}^*$  at a position  $p$  if for all  $i \in [1..s]$ ,  $\pi[i] = \#$  implies  $A[p + i - 1] = 1$ , where  $s$  is the length or span of  $\pi$ .

Iedera is capable of producing spaced seeds based on both a Bernoulli model as well as a Hidden Markov model. Figure 2.14 shows the classical PatternHunter 1 spaced seed designed using Iedera.

###-#-#-#-###	0.999999761581	0.467122	0.532878
(SEED PATTERN)	(selectivity)	(SENSITIVITY)	(distance to 1,1)

Figure 2.14: The classical PatternHunter spaced seed designed using Iedera PatternHunter seed can be designed by using the command : `iedera -spaced -w 11, 11 -s 11, 18`. Here, the seed weight is set to 11, and the span is at most 18 and homology region is set to 64 by default. [16]

### 2.4.2 AcoSeed

In this study, an Ant Colony Optimization (ACO) based algorithm, called AcoSeed [18], is used to generate optimal multiple spaced seeds. Ant Colony Optimization (ACO) [34] is a meta-heuristic technique that simulates the behavior of real ant colony.

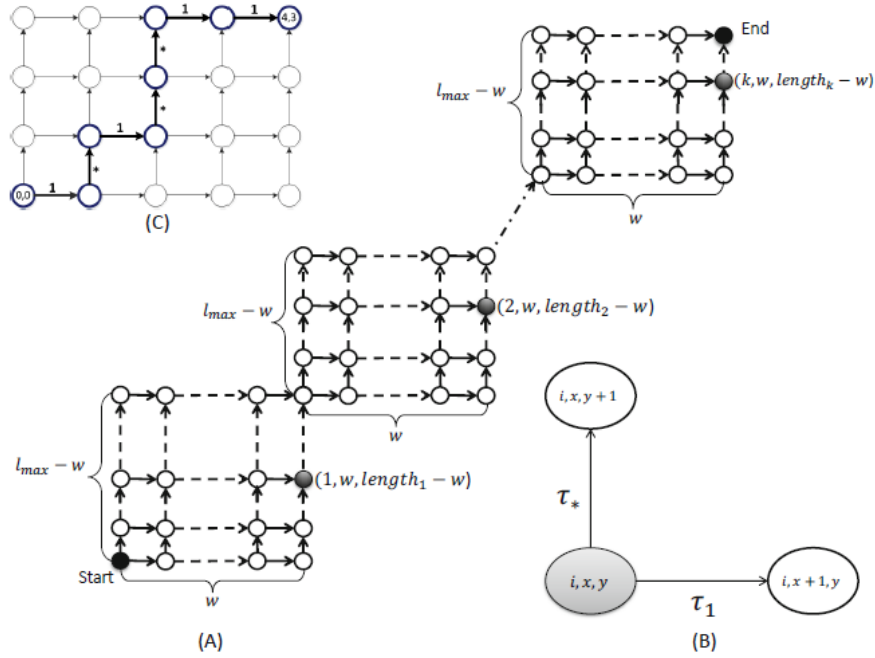


Figure 2.15: AcoSeed algorithm

(A) shows the construction graph for building  $k$  spaced seeds, each of length  $w$ . (B) shows the ant's possible travel path, which can be up or right. (C) shows an example of building spaced seeds of weight 4 and length 7. The path taken by an ant is RURUURR which corresponds to the seeds  $1 * 1 ** 11$ . [18]

AcoSeed uses an adaptation of the MAX-MIN Ant system which allows an ant colony to travel in the form of a construction graph to create a spaced seed. The important steps of AcoSeed algorithms are:

- **Construction Graph** A construction graph has  $k$  rectangles of length  $w \cdot (l_{max} - w)$ . Thus, the set has  $k$  seeds each of weight  $w$ . Each ant builds  $k$  seeds by traveling on each rectangle

either up or right from starting node  $(i, 0, 0)$  for rectangle  $i$  where  $i = 1, \dots, k$ . Travelling right adds 1 and travelling up adds \*. The pheromone concentration  $\tau$  denotes how likely the ant colony chooses either Up (\*) or Right (1).

Figure 2.15(A) shows the construction graph for building  $k$  spaced seeds, each of length  $w$ . Figure 2.15(B) shows the ant's possible travel path, which can be up or right. Figure 2.15(C) shows an example of building spaced seeds of weight 4 and length 7. The path taken by an ant is RURUURR which corresponds to the seeds  $1 * 1 * * 11$ .

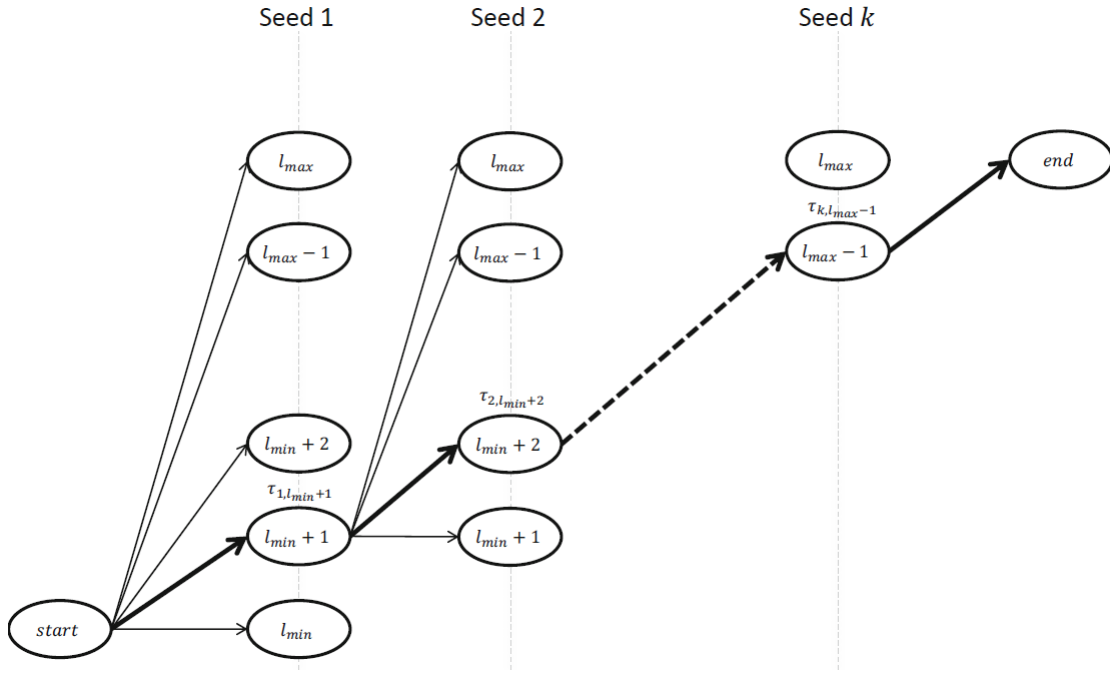


Figure 2.16: ACO construction graph for the identification of seed lengths

From the starting node in the construction graph, each ant chooses one of the following nodes  $(1, l_{min}), (1, l_{min} + 1), \dots, (1, l_{max})$  so that seed 1 will end up having length  $l_{min}, l_{min} + 1, \dots, l_{max}$  respectively. As the seed length increases, the ant chooses the node  $(i, length_{i-1}), (i, l_{min} + 1), \dots, (i, l_{max})$  where  $i = 2, \dots, k$ . [18]

- **ACO-Based Seed Length Identification** Another ACO algorithm is used to identify the length of each seed based on the construction graph in previous step. Each ant begins from the starting node and chooses one of the following nodes  $(1, l_{min}), (1, l_{min} + 1), \dots, (1, l_{max})$  so that seed 1 will end up having length  $l_{min}, l_{min} + 1, \dots, l_{max}$  respectively. As the seed length increases, the ant chooses the node  $(i, length_{i-1}), (i, l_{min} + 1), \dots, (i, l_{max})$  where  $i = 2, \dots, k$ . This is shown in Figure 2.16.
- **AcoSeed Algorithm** Figure 2.17 shows the AcoSeed algorithm.
- **Local Search Using Overlap Complexity** After each ant completes building a spaced seed, its quality is measured by using overlap complexity [7, 13]. The overlap complexity, which has polynomial runtime, is an approximation of sensitivity, which has exponential runtime. Starting from the initial spaced seed constructed by the current ant, the local

search greedily swaps a 1 and a \* for each seed to obtain a new spaced seed with a better approximated sensitivity.

```

Data:  $w, k, p, N$ 
Output: The optimal spaced seed  $s_{best}$  set w.r.t the sensitivity
begin
   $s_{g-best} \leftarrow null$  ; Estimating  $l_{min}, l_{max}$ ;
  while stop conditions not satisfied do
    foreach  $i = 1..N_{ant}$  do
      Determine the seed length;
       $s_i \leftarrow \text{SolutionConstruction}()$ ; {seed built by the ant  $i^{th}$ }
       $s_i \leftarrow \text{LocalSearch}(s_i)$ ; {using OC heuristics}
      Computing sensitivity for  $s_i$ :  $F(s_i)$  ;
     $s_{i-best} \leftarrow \text{argmax}(F(s_1), F(s_2), \dots, F(s_{n_a}))$ ;
    ApplyPheromoneUpdate ( $s_{i-best}$ );
    Update the global best seed  $s_{g-best}$ ;
  Output  $s_{g-best}$ ;
end

```

Figure 2.17: Construction graph and seed building procedure [18]

### 2.4.3 rasbhari

”Rapid Approach for Seed optimization Based on a Hill-climbing Algorithm that is Repeated Iteratively” (or rasbhari in short) is an algorithm based on iterative hill climbing that generates multiple spaced seeds of very high sensitivity [19]. rasbhari works a lot like SpEED [13] with some key differences in the optimization step, SpEED is described in great detail in the next chapter.

The software program accepts 4 inputs - seed weight, similarity level, number of seeds and length of the homologous region. Based on these inputs, rasbhari creates a random seed set and optimizes it based on overlap complexity and seed contribution using iterative hill-climbing with random starts. The sensitivity of the optimized seed is calculated and this is the output of the program along with the seed. Overlap complexity and sensitivity are discussed in detail in the next chapter and the important steps of the rasbhari algorithm are described next.

- **OC Contribution :** The Overlap Complexity (OC) contribution of each seed can be found as a by-product while calculating the overlap complexity [7, 13] of the entire seed set. The contribution of the  $r^{th}$  seed ( $C_r$ ) is given by:

$$C_r = \sum_{r'} \alpha_{rr'}$$



$$\text{where, } \alpha_{rr'} = \sum_{s=1-l_{r'}}^{l_{r-1}} 2^{\sigma_{rr'}[s]}$$

$$\text{thus, } C_r = \sum_{r'} \sum_{s=1-l_{r'}}^{l_{r-1}} 2^{\sigma_{rr'}[s]}$$

In other words, the contribution of the  $r^{th}$  seed is the summation of overlap complexity value of the  $r^{th}$  seed with all other seeds in the set. The contribution of each seed is required in the next step.

- **Modified Hill-Climbing Algorithm :** In this algorithm, a swap between a match position and a don't care position is made and  $OC$  is evaluated on a specific seed based on the contribution value calculated in the previous step. Instead of looking at all possible swap possibilities in all seeds, the algorithm looks at those seeds first which have high contribution value. So, all the seeds in the set are sorted in descending value of  $C_r$  and seeds are chose for optimization in that order. The modified hill climbing algorithm used in rasbhari when running with default values is shown below.

```

for  $i$  from 1 to 5000 do
  for  $j$  from 1 to 100 do
    generate random multiple seed  $S_{ij}$ 
    sort the seeds of  $S_{ij}$  decreasingly by  $OC$  contribution
     $S_{cur} \leftarrow$  the first seed of  $S_{ij}$ 
    repeat 25000 times
      swap random 1 and * in  $S_{cur}$ 
      if  $OC$  decreases then keep this as the new  $S_{ij}$ 
      else  $S_{cur} \leftarrow$  the next seed (restart from first if  $S_{cur}$  is the last)
     $S_i \leftarrow$  the multiple seed  $S_{ij}$  that minimizes  $OC(S_{ij})$ , for all  $j = 1..100$ 
     $S \leftarrow$  the multiple seed that maximizes  $Sens(S_i)$ , for all  $i = 1..5000$ 
  return  $S$ 

```

If swapping a match position and don't care position does not improve the current seed set, the next seed from sorted order is chosen and proceed in the same way. If swapping leads to improvement in  $OC$ , then swap is accepted and again the new contributions of all the seeds are calculated and seeds are again sorted in descending order. The hill climbing is performed

The hill climbing is continued until a user-defined number of times (25,000 by default). After this step, rasbhari gets a set of multiple spaced seeds with the lowest overlap complexity. This whole process is repeated 100 times and the best set is chosen. Then sensitivity of this set (with lowest  $OC$ ) is computed. This whole process, in turn, is repeated 5,000 times. This is similar to SpEED, but in SpEED the sensitivity is calculated after one round of hill climbing (total 5,000 iterations). By contrast, rasbhari runs the modified hill-climbing routine 100 times before calculating the sensitivity for the best seed set from these 100 runs. The final output of rasbhari is the set of multiple spaced seeds with the highest sensitivity from the 5,000 iterations. Results are shown in Table 2.7; rasbhari generates the best seeds in most of the seed types. The highest sensitivity values are shown in bold.

rasbhari produces better quality seeds than SpEED in terms of sensitivity but it is almost five orders of magnitude slower than SpEED when total time (overlap complexity optimization time + sensitivity computation time) is considered. The main reason for this is that rasbhari seeds are on average almost 1.8 times longer than SpEED seeds (having the same weight) on average and we know that computing sensitivity takes an exponential amount of time.

This concludes our background studies. We have seen traditional methods of sequence alignment like Needleman-Wunsch algorithm and Smith-Waterman algorithm. Then we looked into different heuristic algorithms like MUMmer, BLAST and PatternHunter. Then, in the end, we looked at algorithms like AcoSeeD and rasbhari which are concerned with creating multiple spaced seeds. Next, we move on to SpEED and the Methodology of our developed software.

w	p	Iedera	SpEED	AcoSeeD	rasbhari
<b>SHRiMP2: 4 patterns (H = 50)</b>					
10	0.75	90.6820	90.9098	90.9513	<b>90.9614</b>
	0.80	97.7586	97.8337	97.8521	<b>97.8554</b>
	0.85	99.7437	99.7569	99.7614	<b>99.7618</b>
11	0.75	83.2413	83.3793	<b>83.4728</b>	83.4679
	0.80	94.9350	94.9861	95.037	<b>95.0386</b>
	0.85	99.2189	99.2431	99.2478	<b>99.2506</b>
12	0.80	90.3934	90.5750	90.6328	<b>90.6648</b>
	0.85	98.0781	98.1589	98.1766	<b>98.1824</b>
	0.90	99.8773	99.8821	99.8853	<b>99.8864</b>
16	0.85	84.5795	84.8212	<b>84.9829</b>	84.969
	0.90	97.2806	97.4321	97.4712	<b>97.5035</b>
	0.95	99.9331	99.9388	99.9419	<b>99.9441</b>
18	0.85	72.1695	73.1664	<b>73.27</b>	73.2209
	0.90	93.0442	93.7120	93.7778	<b>93.78</b>
	0.95	99.6690	99.7500	<b>99.7599</b>	99.7557
<b>PatternHunterII: 16 patterns (H = 64)</b>					
11	0.70	92.0708	93.2526	-	<b>93.4653</b>
	0.75	98.3391	98.6882	-	<b>98.7573</b>
	0.80	99.8366	99.8820	-	<b>99.8907</b>
<b>BFAST: 10 patterns (H = 50)</b>					
22	0.85	60.1535	60.8127	-	<b>60.9919</b>
	0.90	87.9894	88.5969	-	<b>88.8005</b>
	0.95	99.2196	99.3659	-	<b>99.4099</b>

Table 2.7: Sensitivity comparison of different programs  
[19]

# Chapter 3

## Methodology

In this chapter, we present a new software program capable of producing highly sensitive multiple spaced seeds called SpEED2. SpEED2 is built on top of SpEED [7, 13, 14], so first, we describe the software program SpEED and some underlying concepts, then, we move on to our work.

### 3.1 SpEED

Given the number of seeds, the weight of each seed, homology region length, and similarity level, finding optimal multiple spaced seeds with these given parameters is NP-hard [7, 13]. It is not possible to find the optimal seed by searching exhaustively because searching involves two exponential-time steps - there are exponentially many seeds to be tried and computing the sensitivity of each seed takes an exponential amount of time [7]. The simplest way to bypass these two difficulties is by not considering all seeds and by avoiding computing sensitivity, which is exactly what SpEED does [13].

First, we discuss sensitivity, then discuss how computing sensitivity is avoided and finally discuss how exhaustive search is avoided. An important point to keep in mind is that the sensitivity of seed has to be computed in order to gauge the quality of seed found but the number of sensitivity computations can be minimized.

#### 3.1.1 Sensitivity

Let us say there are two DNA sequences  $S_1$  and  $S_2$  having similarity  $p$ . When these two sequences are compared, we get another sequence  $R$  consisting of 1's (corresponding to matches) and 0's (corresponding to mismatches) that occur with probability  $p$  and  $1 - p$ , respectively as shown in Figure 3.1. Therefore, given an infinite Bernoulli random sequence  $R$  and a seed  $s$ , we can say that  $s$  hits  $R$  (ending) at position  $k$  if aligning the end of  $s$  with position  $k$  of  $R$  results in every 1 in  $s$  to align with a 1 in  $R$ .

The sensitivity of a seed  $s$  is the probability that  $s$  hits  $R$  at or before position  $n$  [5, 7, 35]. Sensitivity depends on both the similarity level  $p$  as well as the length of the random homologous region  $n$ . The sensitivity of a multiple spaced seed  $S$  is defined as the probability that at least one seed of  $S$  hits a sequence  $R$  at or before position  $n$ . The sensitivity of multiple

DNA seq. $S_1$	A C G A G G C A C T G T A T G T A T A T C T A
DNA seq. $S_2$	A G T A G G C A A T G C A T T T A A A T C T C
matches/mism.	= ≠ ≠ = = = = ≠ = = ≠ = = ≠ = = = ≠
Bernoulli seq. $R$	1 0 0 1 1 1 1 1 0 1 1 0 1 1 0 1 1 1 1 0
spaced seed $s$	1 1 1 * 1 * * 1 * 1 * * 1 1 * 1 1 1

Figure 3.1: An example of a hit using PatternHunter's spaced seed

An example of a hit using PatternHunter's spaced seed. All 1's in the seed (the last row) must correspond to matches between the sequences. The spaced seed  $s$  hits the Bernoulli sequence  $R$  (ending) at the third position from the right. [7]

spaced seeds is calculated using in exponential time and space by a dynamic programming algorithm in [6].

### 3.1.2 Overlap Complexity

In SpEED, a substitute for sensitivity known as the overlap complexity ( $OC$ ) is introduced in [7], which is inversely proportional to sensitivity. A good seed must have a high sensitivity value and a low overlap complexity. The overlap complexity between two seeds  $s_1$  and  $s_2$  is given by:

$$OC(s_1, s_2) = \sum_{i=1-|s_2|}^{|s_1|-1} 2^{\sigma[i]}$$

where  $\sigma[i]$  is the number of pairs of 1's aligned together when a copy of  $s_2$  shifted by  $i$  positions is aligned against  $s_1$ . The shift  $i$  takes values from  $1 - |s_2|$  to  $|s_1| - 1$ . Let us look at an example with two seeds  $s_1 = 11 * * 1 * 1$  and  $s_2 = 1 * 11$  as shown in Figure 3.2. Overlap Complexity ( $OC$ ) value is  $25 = 2 + 4 + 2 + 2 + 4 + 2 + 2 + 4 + 1 + 2$ .

For a multiple seed  $S = \{s_1, s_2, \dots, s_k\}$ , the overlap complexity is defined by:

$$OC(S) = \sum_{1 \leq i \leq j \leq k} OC(s_i, s_j)$$

The overlap complexity can be computed in polynomial time. SpEED finds good seeds using another polynomial-time heuristic algorithm. It starts with a fixed seed and repeatedly modifies it to improve its overlap complexity (more precisely decrease the overlap complexity). We discuss this procedure in the next section.

### 3.1.3 Iterative Hill Climbing with Random Starts

SpEED randomly chooses an initial set of seeds and improves these seeds based on  $OC$ . To improve the current seed set  $P$ , the hill-climbing algorithm looks at all triplets  $(r, i, j)$  where  $P_r$  is a seed in set  $P$ , and  $i$  and  $j$  are a match position and a don't-care position in  $P_r$ , respectively. For each such triplet  $(r, i, j)$ , the algorithm considers the seed set that would be obtained from  $P$  by swapping  $i$  and  $j$  in  $P_r$ . The  $OC$  is calculated for all seed sets that can be obtained in this manner, and the  $P$  with the lowest  $OC$  is selected as the next seed set  $P$ , until  $OC$  cannot be decreased any further. This whole process is repeated iteratively.

	shift $i$	$\sigma[i]$
* * * 1 1 * * 1 * 1 * * *		
1 * 1 1 * * * * * * * *	-3	1
* 1 * 1 1 * * * * * * * *	-2	2
* * 1 * 1 1 * * * * * * *	-1	1
* * * 1 * 1 1 * * * * * *	0	1
* * * * 1 * 1 1 * * * * *	1	2
* * * * * 1 * 1 1 * * * *	2	1
* * * * * * 1 * 1 1 * * *	3	1
* * * * * * * 1 * 1 1 * *	4	2
* * * * * * * * 1 * 1 1 *	5	0
* * * * * * * * * 1 * 1 1	6	1

Figure 3.2: An example of the overlap complexity of two seeds

$$OC(11 * * 1 * 1; 1 * 11) = \sum_{i=-3}^6 2^{\sigma[i]} = 25 [7]$$

The number of triplets  $(r, i, j)$  to be considered depends on the product of the number of seeds  $m$  in the seed set  $P$  and the length of each seed  $l$ . For each of the triplets, the  $OC$  is calculated for the seed set obtained by swapping  $i$  and  $j$  in  $P_r$ . The seed set  $P_{min}$  with the lowest  $OC$  is obtained by simulating all possible combinations of the triplets  $(r, i, j)$ . The sensitivity of  $P_{min}$  is computed and the whole process is repeated 5000 times.

### 3.1.4 Algorithm

The input to the algorithm is the number of seeds ( $k$ ), seed weight ( $w$ ), homology length ( $H$ ) and sequence similarity value ( $p$ ). In step 1 to step 7 from Figure 3.3, SpEED decides on the length of all  $k$  seeds;  $m$  is the length of the shortest seed whereas  $M$  is the length of the longest seed. In the practical implementation of SpEED [13],  $M$  value is not fixed to 25;  $m$  and  $M$  values are derived from a precalculated array of numbers based input. The other  $k - 2$  seed lengths are calculated by using interpolation. In step 7, we end up with all  $k$  seeds of desired lengths and each seed is filled up with the pattern shown in step 6.

From step 8 onwards, the algorithm is concerned with improving the initial seed set by using iterative hill climbing based on overlap complexity. The initial seed set has a high overlap complexity value and all possible swaps between a match position (1) and a don't care position (\*) are simulated and the swap is performed if and only if the overlap complexity of seed set decreases, else not. The algorithm greedily chooses a swap that produces the maximum decrease in overlap complexity value.

After the seed set with the lowest overlap complexity is discovered, the sensitivity of that particular set is calculated. This whole above-mentioned process (except calculating  $m$  and  $M$  values) is repeated 5,000 times and the seed set with the highest sensitivity value is reported back to the user as output. The complete SpEED algorithm is shown in Figure 3.3.

MULTIPLESEEDS( $w, k$ )  
- given: the weight  $w$  and the number of seeds  $k$   
- returns: a multiple seed  $S$  with  $k$  seeds of weight  $w$  and high sensitivity

```

    // find the length of the seeds – half are equally spaced
    // in the interval  $m..M$ , the others have length  $M$ 
1.  $m = \text{round\_up } \frac{4w}{3}$  // shortest seed
2.  $M = 25$  // longest seed
3.  $h = \frac{2(M-m)}{k}$  // float
4. for  $i$  from 1 to  $k$  do
5.    $\ell_i \leftarrow \min(\text{round\_up}(m + i \times h), 25)$ 
6.    $s_i \leftarrow \star^{\ell_i - w} 1^w$ 
7.  $S \leftarrow \{s_1, s_2, \dots, s_k\}$ 
   // swap 1's and *'s to improve sensitivity
8. swaps  $\leftarrow 0$ 
9. while  $\exists r, i, j$  with  $\text{OC}(\{s_1, \dots, s_{r-1}, \text{flip}(s_r, i, j), s_{r+1}, \dots, s_k\}) < \text{OC}(S)$  and  $(\text{swaps} \leq k \times w)$  do
10.   choose a triple  $(r, i, j)$  that reduces  $\text{OC}(S)$  the most
11.    $S \leftarrow \{s_1, \dots, s_{r-1}, \text{flip}(s_r, i, j), s_{r+1}, \dots, s_k\}$ 
12.   swaps  $\leftarrow \text{swaps} + 1$ 
13. return( $S$ )

```

Figure 3.3: SpEED algorithm

The MULTIPLESEEDS algorithm which, given the weight and lengths of the seeds, computes a multiple seed with low overlap complexity and, therefore, high sensitivity. [7]

### 3.1.5 SpEED-Fast

Two speed-up ideas have been suggested in [13] : the first algorithm computes the overlap complexity ( $OC$ ) faster, and the second algorithm speeds up the hill climbing procedure. These two algorithms have been implemented in "Efficient computation of spaced seeds" [14].

The first algorithm computes the overlap complexity faster by converting all seeds into 64-bit integers i.e., 1 and \* are represented as bits 1 and 0. Thus,  $1 * * 11$  is converted to integer 10011 which is 19. The  $OC$  is then computed by shifting the bits and performing logical AND on each bit of the seeds.

The second algorithm improves the speed of the hill climbing procedure by reducing the work for the  $OC$  computation of each swap between a 1 and a \* (step 9 in Figure 3.3). Only the new  $OC$  values, ie., those involving  $S_r$ , are recomputed.

## 3.2 SpEED2

Our work, SpEED2 is built on top of the existing algorithms described above. We have modified the algorithm and added procedures to estimate the sensitivity, indel optimization of seeds and adapt the seed lengths. We discuss estimating sensitivity, indel optimization and adap-



```

000000000000001100101011110101101111011111111111111110111000101101
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111
                                     111011011111

```

Figure 3.4: Example of estimated sensitivity calculation

The seed 111011011111 is compared against a homologous region  $A_i$ . A hit is said to occur if the seed aligns perfectly with the region in any position (shown in blue)

where *count* is the number of elements in  $A$  that are hit by the given seed set and  $|A|$  is the total number of elements in the array  $A$ . In SpEED2, this whole process of hit detection is efficiently implemented using bit-wise left shift (shown in Figure 3.5) along with bit-wise AND operation (shown in Figure 3.5). Bit-wise AND operation between a seed and a region (directly above the seed) will always produce that particular seed if a hit is detected.

It must be noted that in [36], an array of size  $10^6$  was used to estimate the sensitivity correctly up to the third significant digit. For our purpose of differentiating a good multiple spaced seeds from a bad one, we required precision up to the fourth decimal place. The most intuitive solution to this problem was to use a longer array  $A$  and this is exactly what was done. The challenge was to choose an array size that would not only produce estimated sensitivity numerically equal to real sensitivity but also perform the computation quickly. The multiple spaced seeds with the best estimated sensitivity obtained after 100 iterations (of various weights, similarity value and a different number of seeds) were tested using different array lengths -  $10^7, 2 * 10^7, 5 * 10^7, 10^8, 10^9$  and compared against the seed's real sensitivity and each other. One instance of the testing process is shown in Figure 3.6 and Figure 3.7.

The SHRiMP seed of similarity 0.75 was run 10 times with different array lengths like  $10^6, 10^7, 5 * 10^7, 10^8$  and  $10^9$  and the resulting estimated sensitivity value was compared with the real sensitivity and each other. In Figure 3.6, we have the array lengths in the X-axis and sensitivity in the Y-axis. The red horizontal line represents the real sensitivity value. In Figure 3.7, we are comparing the absolute difference between estimated and real sensitivity values



```
000000000000000110010101111010110111101111111111110111000101101
                                111011011111
AND
000000000000000110010101111010110111101111011110110111101110111000101101
```

Figure 3.5: Hit detection for estimating sensitivity

Bit-wise AND operation between a seed and a region will always result in that particular seed if a hit is detected.

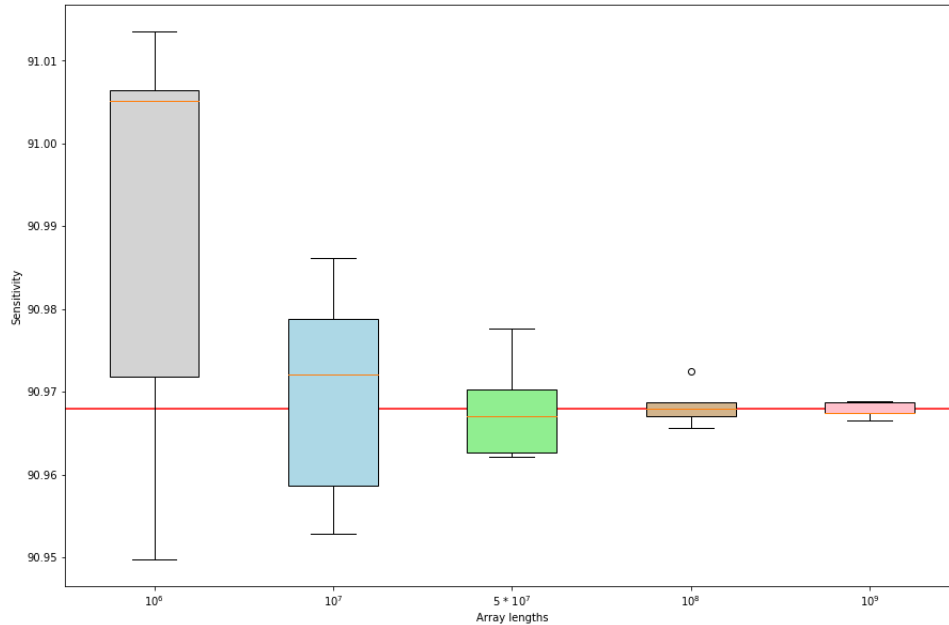


Figure 3.6: Comparison between real and estimated sensitivity

The SHRiMP seed of similarity 0.75 was run 10 times with different array lengths like  $10^6$ ,  $10^7$ ,  $5 * 10^7$ ,  $10^8$  and  $10^9$  and the resulting estimated sensitivity value was compared with the real sensitivity and each other.

for SHRiMP seed of similarity 0.75 with the array lengths in X-axis and absolute sensitivity difference in the Y-axis.

We see that  $10^8$  produces results that mimic the real value. Also, it was observed that  $10^8$  and  $10^9$  lengths produced a very similar result with  $10^9$  running 10 times slower (algorithm has linear runtime). We see such a result from Figure 3.8, where SHRiMP seed of similarity 0.75 was run 10 times with the array length of  $10^8$  and  $10^9$  and the resulting estimated sensitivity value was compared with the real sensitivity and each other with the red horizontal line representing the real sensitivity value. The estimates of array of size  $10^8$  is slightly more dispersed than  $10^9$  estimates, however on an average,  $10^8$  estimates are closer to the real one (as evident by the orange lines inside the box plots).

The earlier versions of SpEED would crash if the code consumed all system memory while computing sensitivity of long seeds, however, SpEED2 can detect whether sensitivity computation would crash the code. Before sensitivity is computed, SpEED2 calculates the total amount

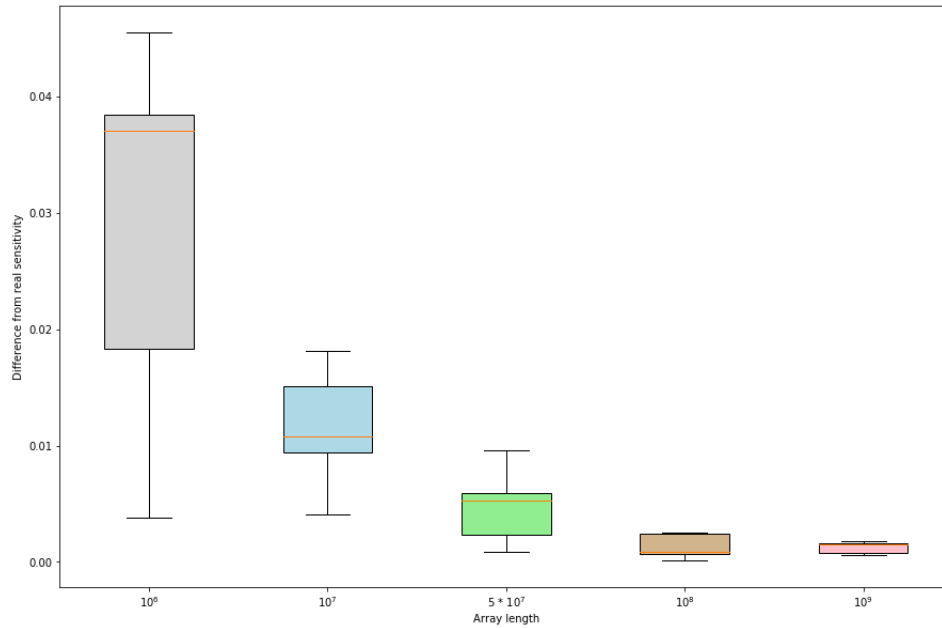


Figure 3.7: Difference between real and estimated sensitivity

Absolute difference between estimated and real sensitivity values for SHRiMP seed of similarity 0.75 with the array lengths in X-axis and absolute sensitivity difference in the Y-axis.

of memory, sensitivity computation would consume and aborts the process if more than 90% of system memory would be utilized and uses estimated sensitivity instead. Estimated sensitivity calculation consumes an additional memory of 1 MB on top of the 1.6 GB required for maintaining the homologous array, irrespective of the type of seed whereas computing the actual sensitivity of seeds varies greatly according to the type of seed - weight 10 SHRiMP seeds consume less than 32 GB of memory whereas MegaBLAST seeds require more than 1 TB of memory to compute sensitivity. This switching mechanism makes our code highly robust and fault-tolerant.

### 3.2.2 Indel Optimization

Indel optimization is the novel algorithm that improves seeds designed by SpEED-Fast and produces multiple spaced seeds of high sensitivity. Indel stands for insertion/deletion and this algorithm either inserts or deletes a don't care position in a randomly selected seed in a randomly selected position.

Works like AcoSeed [18] and rasbhari [19] tried to design more sensitive seeds, compared to SpEED [7, 13], by optimizing the hill-climbing algorithm. These approaches tried to maximize the cost function, which is sensitivity and believed that optimizing the hill-climbing approach would prevent their algorithm from getting stuck in a local maximum, as a result, seeds with higher sensitivity value would be discovered. rasbhari produces more sensitive seeds than SpEED and if we compare them, we instantly notice that rasbhari seeds are longer, approximately 80% longer than SpEED seeds and have considerably lower *OC* value.

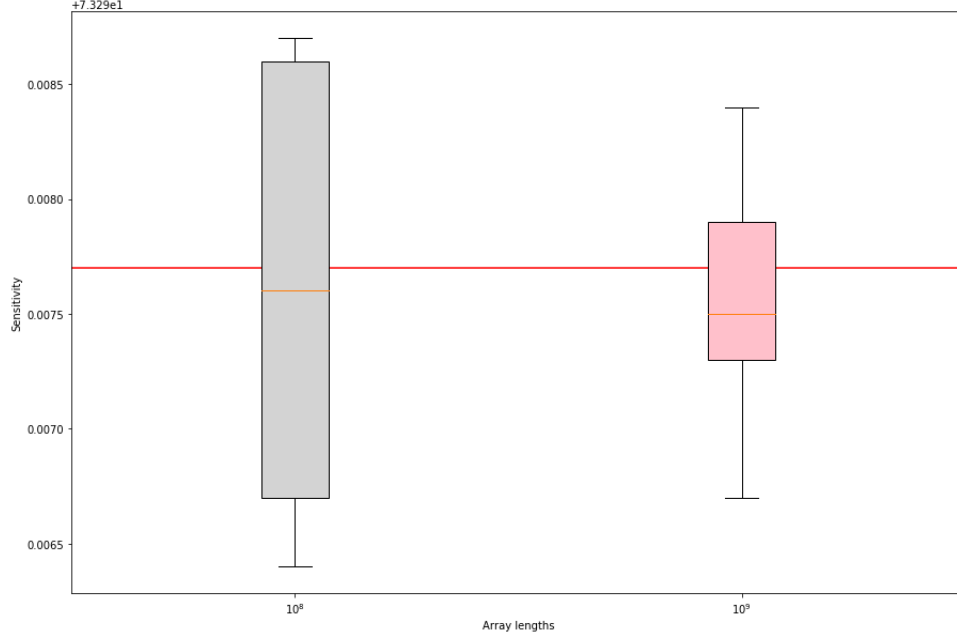


Figure 3.8: Comparison between real and estimated sensitivity  
SHRiMP seed of similarity 0.75 was run 10 times with the array length of  $10^8$  and  $10^9$  and the resulting estimated sensitivity value was compared with the real sensitivity and each other where the red horizontal line representing the real sensitivity value.

Let us look at some cases, we will refer to the shortest seed of the set as  $m$  and the longest seed as  $M$ . In case of SpEED,  $m = 12$  and  $M = 23$  for weight 10 SHRiMP seeds and  $m = 23$  and  $M = 36$  in case of rasbhari. For weight 18 SHRiMP seeds,  $m = 22$  and  $M = 36$  in case of SpEED and  $m = 39$  and  $M = 60$  in case of rasbhari. For PatternHunter II seeds,  $m = 14$  and  $M = 27$  in case of SpEED and  $m = 25$  and  $M = 39$  in case of rasbhari. For BFAST seeds,  $m = 25$  and  $M = 37$  in case of SpEED and  $m = 47$  and  $M = 72$  in case of rasbhari.

Figure 3.9 shows the  $OC$  value of the multiple spaced seed designed after the 1<sup>st</sup> iteration where the blue line is that of rasbhari and the red line is that of SpEED-Fast. On the Y-axis, we have  $OC$  and on X-axis, we have SHRiMP seeds of different weights with similarity of 0.75. It can be seen that rasbhari seeds consistently have a lower  $OC$  value.

Experimentally, we have observed that this reduction in  $OC$  value is obtained because of the longer lengths of rasbhari seeds. We modified SpEED so that it designed seeds with lengths equal to rasbhari seed lengths and the  $OC$  of these seeds was lower or same as those of rasbhari.

Thus we concluded that the seed lengths affect the  $OC$  value which in turn, affects the sensitivity. So in SpEED2, we explored this uncharted approach of designing seeds by modifying the seed lengths. We wanted an approach that not only modifies the length of seeds but also changes the seed structure simultaneously, inserting or deleting don't care positions to seed set in random positions meet our requirement. The indel optimization algorithm is discussed below.

The algorithm takes 3 parameters as input - a seed set  $S$  that needs to be optimized,  $n$  the number of seeds in the set and  $curSens$  which is the sensitivity of  $S$ . This algorithm is preceded by optimization using iterative hill-climbing based on  $OC$ , thus  $S$  is already somewhat

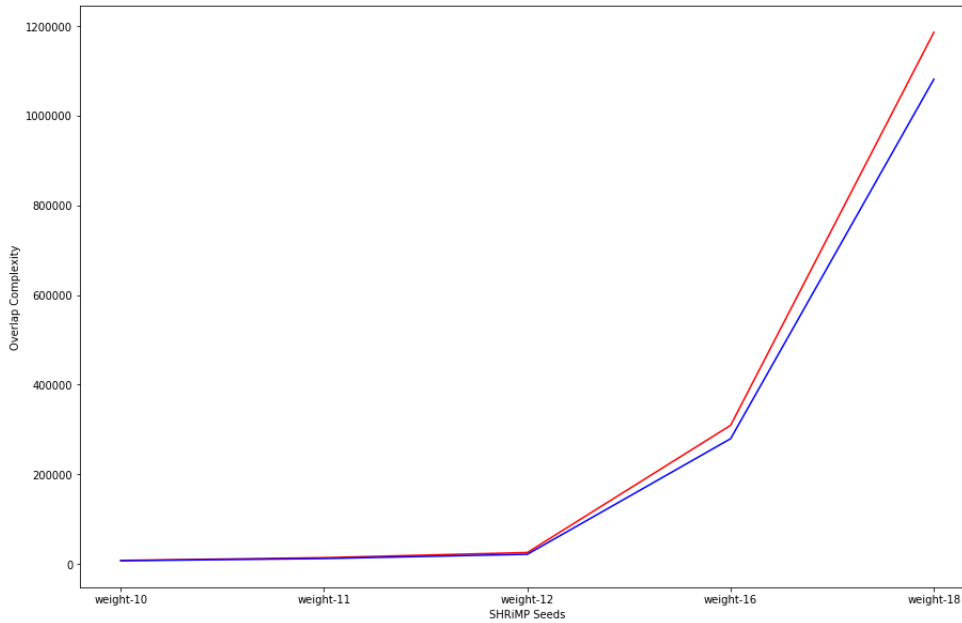


Figure 3.9: *OC* comparison between SpEED-Fast and rasbhari

The figure shows the *OC* value of the multiple spaced seed designed after the 1<sup>st</sup> iteration where the blue line is that of rasbhari and the red line is that of SpEED-Fast.

optimized and *curSens* is the highest value of sensitivity value seen before indel optimization is performed. In this algorithm, we first randomly select one seed  $s$  out the given set  $S$ , then randomly choose whether to insert or delete a don't care position in  $s$ . For both the choices, a valid position  $p$  is randomly chosen and indel is performed converting  $S$  to  $S'$ . After that,  $S'$  is optimized by swapping between a 1 and a \* to lower the overlap complexity (*OC*) similar to that of SpEED. In case of insertion,  $p$  can be any position except the first and last and in case of deletion, a  $p$  is chosen such that a \* already exists in  $p^{th}$  position of  $s$ . This whole process is repeated 200 times and the best seed designed is stored as  $S''$  and its corresponding sensitivity is stored as *curSens*. After trying out a different number of iterations ranging from 50 to 500, we settled for 200 which strikes a good balance between optimizing and time requirement. The seed set with the highest sensitivity from the 200 runs is stored in  $S''$  and this is our optimized multiple spaced seed, which is returned as output.

The seeds have positions from 0 to  $length(seed)-1$ . One drawback of the indel optimization algorithm is that it has a high runtime because the algorithm uses sensitivity as cost function while optimizing and we have already discussed that sensitivity computation is very expensive in terms of both memory and time. This algorithm tries to improve the initial seed set 200 times, thus sensitivity is computed 200 times. We tried indel optimization by using *OC* as the cost function as this would have made the computation very fast but it failed to produce good results. Another issue with the indel optimization algorithm is that the length of some seed may increase rapidly due to \* insertion and computing the sensitivity of lengthy multiple spaced seeds is very expensive. Thankfully SpEED2 detects if sensitivity computation is possible with the available memory and switches to estimated sensitivity for that particular iteration. The multiple spaced seeds obtained after indel optimization have high sensitivity, and they are

available in Appendix A. We have also shown in the next chapter that SpEED2 seeds are more sensitive than the seeds produced by other leading software.

#### Indel Optimization ( $S, n, curSens$ )

```

- given: an unoptimized seed set  $S$ , number of seeds  $n$ , and
          $curSens$  which is the sensitivity of  $S$ 
- returns: optimized seed set  $S''$ 

1. for  $i$  from 1 to 200 do
2.    $choice = 0$  or 1
3.    $seed = S[s]$ ,  $0 \leq s \leq n - 1$ 
4.   if ( $choice == 0$ )
5.      $pos = p$ , where  $1 \leq p \leq length(seed) - 2$ 
6.      $S' = \text{add } * \text{ at } pos \text{ in } seed.$ 
7.   else if ( $choice == 1$ )
8.      $pos = p$ , where  $1 \leq p \leq length(seed) - 2$ 
        and  $seed[p] == *$ 
9.      $S' = \text{remove } * \text{ at } pos \text{ in } seed.$ 
10.   $sens = \text{calculate sensitivity of } S'$ 
11.  optimize  $S'$  by swapping 1 and  $*$  based on  $OC$ 
12.  if ( $sens > curSens$ )
13.     $curSens = sens$ 
14.     $S'' = S'$ 
15. end for
16. return ( $S''$ )

```

### 3.2.3 Adaptive Seed Length

Adaptive seed length is the third algorithm which helps in designing highly sensitive seeds. The SpEED2 algorithm contains 2 nested loops - an outer loop for optimizing OC using hill climbing and an inner loop for indel optimization using sensitivity. The outer loop runs 1000 times and the inner loop runs 200 times for each outer loop. This algorithm is triggered after every 10 iteration of the outer loop.

Let the length shortest seed of the set be  $m$  and the length of the longest seed be  $M$  and length of all the other seeds are obtained from a pre-computed array of seed lengths (discussed in section 3.1.4). As discussed above, the length of seed plays an important role in determining the sensitivity, so we want to adapt the length of initial unoptimized seed from the result obtained from good indel-optimized seeds from previous iterations of the outer loop. Adapting the seed lengths improves the performance of both the optimization algorithms - OC optimization using hill climbing and indel optimization using sensitivity. The arithmetic means of  $m$

and  $M$  values of the last 10 indel-optimized seeds are calculated and used as the initial seeds for the next 10 iterations. Again this process is repeated every 10 iteration.

The performance of the hill-climbing algorithm is improved because the length of the initial unoptimized seed is the same as those of highly optimized seeds obtained after indel optimization, thus its OC will be lower than any randomly chosen initial seed set of predetermined length. The performance of indel optimization is increased as no iteration is wasted to bring seeds closer to lengths of highly sensitive seeds. The indel optimization algorithm immediately focuses on improving seed sensitivity by modifying the seed pattern (a combination of 0s and 1s) instead of worrying about the lengths. Without using this algorithm, the initial seed set was predetermined and many iterations of the indel optimization would be wasted in improving the seed lengths and in the case of hill-climbing optimization, there is no provision to change seed lengths.

Also, without this algorithm, the length of initial seeds ( $m$  and  $M$  value and all other seed lengths) is determined from a pre-computed array and this would have made SpEED2 less flexible and more dependant on the values from the pre-computed array. Also, our code learns the features of good seeds that were designed before and builds on top of that and this learning is updated every so often. In other words, this algorithm learns from previous mistakes and experiences.

### 3.2.4 SpEED2 Algorithm

Here we first describe the key steps in the sequence they are executed then discuss the complete SpEED2 algorithm as a whole.

#### SpEED2 Key Steps:-

1. Compute min & max seed lengths
2. Compute all seed lengths
3. Create homologous region
4. Populate the seeds with default value
5. Iterative hill climbing
6. Adapt the seed lengths
7. Indel optimization
8. Calculate sensitivity

#### 3.2.4.1 Compute min & max seed lengths

In this step, the lengths the shortest seed ( $m$ ) and the longest seed ( $M$ ) is computed. These two values depend on the number of seeds  $k$ , seed weight  $w$ , and homology length  $N$  and are calculated using regression lines based on values present in precomputed arrays. The same procedure was used in the original SpEED version.

### 3.2.4.2 Compute all seed lengths

After  $m$  and  $M$  values are computed, the length of the other seeds is computed heuristically using interpolation. The same procedure was used in the original SpEED version.

### 3.2.4.3 Create homologous region

In this step, a homologous region is created by filling up an array with  $10^8$  strings of length  $H$  containing 0s and 1s based on similarity value  $p$ . This array is used for estimating the sensitivity when computing the actual sensitivity becomes infeasible. This step was discussed in detail in the "Estimated Sensitivity" section of Chapter 3.

### 3.2.4.4 Populate the seeds with default value

All the seeds are populated in a default format - 1 \* .. \* 11..1. The start and ending positions have matched positions. In the remaining positions, the first half is filled with \* and last half is filled with 1.

### 3.2.4.5 Iterative hill climbing

SpEED2 randomly chooses an initial set of seeds and improves these seeds based on overlap complexity ( $OC$ ). This has been discussed before in the section "Iterative Hill-Climbing with Random Starts" section of Chapter 3. The same procedure was used in the original SpEED version.

### 3.2.4.6 Adaptive seed lengths

Using this procedure, the minimum seed length ( $m$ ) and maximum seed length ( $M$ ) in the seed set are modified based on previous results. All other seed lengths are calculated based on the  $m$  and  $M$  values, so basically all seed lengths are chosen so as to maximize sensitivity. For every 10 hill climbing iterations, the arithmetic mean of  $m$  and  $M$  of the best seeds are calculated and these values are used to create the initial seed from the 11<sup>th</sup> through to the 20<sup>th</sup> iteration. Again, this process is repeated for the next 10 iterations.

### 3.2.4.7 Indel optimization

The seed set is optimized by randomly inserting a don't care (\*) or deleting a don't care (\*) in a randomly chosen position in any seed. This step was thoroughly discussed in the section "Indel Optimization" section of Chapter 3.

### 3.2.4.8 Calculate sensitivity

The quality of the designed seed is checked either using sensitivity or by estimating the sensitivity when calculating actual sensitivity becomes infeasible. This step has been discussed in great detail in "Estimated Sensitivity" and "Sensitivity" sections section of Chapter 3.

The complete SpEED2 algorithm is stated below.

SpEED2 ( $w, n, p, H$ )

- given: Weight of seed  $w$ , number of seeds  $n$ , similarity value  $p$  and homology region length  $H$
- returns: Optimized seed set  $S_{opt}$  and  $bestSens$  its sensitivity value

1. Compute lengths of  $m$  and  $M$
2. Create homologous region
3. **for**  $i$  **from** 1 **to** 1000 **do**
4.     for every 10 iterations, adapt seed lengths by recalculating  $m$  and  $M$  values
5.     All seeds  $s$  in seed set  $S$  are populated as 1 \* .. \* 11..1
6.     Iterative hill climbing is performed on  $S$  by swapping a 1 with a \* based on  $OC$
7.      $S$  is modified to  $S'$  after  $OC$  optimization
8.     **for**  $j$  **from** 1 **to** 200 **do**
9.         Perform indel optimization
10.         $S'$  changed to  $S''$  after indel optimization
11.         $S''$  is optimized by swapping 1 and \* using  $OC$
12.     **end for**
13.      $S_{sens} = S''$ , where  $S''$  has highest sensitivity
14. **end for**
15.  $S_{opt} = S_{sens}$ , where  $S_{sens}$  has highest sensitivity
16.  $bestSens$  is the sensitivity of  $S_{opt}$
17. **return**  $S_{opt}$  and  $bestSens$



# Chapter 4

## Experimental Results

In this chapter we present the experimental results produced by SpEED2 and also compare these results with the leading software programs.

### 4.1 Operation Environment

SpEED2 is implemented in C++. All executions have been carried out on SHARCNET (Shared Hierarchical Academic Research Computing Network), which is a consortium of high performance computing cluster, more specifically, we ran our code on the dusky cluster whose characteristics are described below:

- Processor: 32 cores
- RAM: 1000.0 GB
- Operating System: CentOS 6.3
- Compiler: GCC version 5.1.0
- Storage: 500 GB

Execution of SpEED2 is serial in nature so the code needs only one processor to run. SpEED2 requires a lot of memory during sensitivity computation of lengthy seeds but the program can also be run in memory deficient environment. In that case, seed quality would be measured using estimated sensitivity instead of sensitivity.

### 4.2 Experimental Setup

To compare our software program with existing state-of-the-art programs, we generated multiple spaced seed based on the parameter settings that were practically used in a number of popular biological sequence alignment/search programs such as SHRiMP [37], PatternHunter II [6], BFAST [38], and MegaBLAST [33]. Let us look at each type of seed set. We will use  $k$  to denote the number of seeds,  $p$  for similarity,  $w$  for seed weight and  $H$  for length of homology region.

Name	Seeds	Weight	Similarity	Homology length
SHRiMP	4	10, 11, 12, 16, 18	0.75, 0.80, 0.85, 0.90, 0.95	50
PatternHunter II	16	11	0.70, 0.75, 0.80	64
BFAST	10	22	0.85, 0.90, 0.95	50
MegaBLAST	1, 2, 4, 8, 16	28	0.90	100

Table 4.1: Types of seeds used as dataset

SHRiMP consists of 15 types of seed sets, where each set has  $k = 4$  and  $H = 50$ . The weights,  $w$  have values 10, 11, 12, 16, 18 and  $p$  ranges from 0.75 to 0.95. See Table 4.1 for exact configuration of seeds. PatternHunter II seed sets have  $k = 16$ ,  $w = 11$ ,  $H = 64$ , and  $p$  value of 0.70, 0.75 or 0.80. BFAST seeds sets have  $k = 10$ ,  $w = 22$ ,  $H = 50$ , and  $p$  value of 0.85, 0.90 or 0.95. Finally, we use MegaBLAST seeds which have  $w = 28$ ,  $H = 100$ , and  $p = 0.90$  and the  $k$  has values 1, 2, 4, 8 or 16.

### 4.3 Performance Measurement Metric

We have used two measurement metrics:-

- **Sensitivity**
- **Estimated Sensitivity**

Sensitivity is the best metric to measure the performance of seeds as evident from all previous related works. However, in our work, we have also used estimated sensitivity wherever sensitivity computation fails. Sensitivity has been used for SHRiMP, PatternHunter II, and BFAST seeds whereas estimated sensitivity has been used for MegaBLAST seeds. We have discussed both in great detail in the previous chapter.

### 4.4 Experimental Results

The multiple spaced seeds generated by SpEED2 are given in Appendix A. The output (multiple spaced seeds) from our program is slightly different from those shown in Appendix A. It is in the form of strings consisting of 0s and 1s, where a 0 (instead of \*) represent don't care position and a 1 represents match position. A sample output of SpEED2 is shown in Figure 4.1. In this instance our program was run using the command `./SpEED 10 4 0.75 50`.

We ran the code 10 times for all datasets and considered the best result among the 10 runs. The quality of SHRiMP, PatternHunter II, and BFAST seeds were measured using sensitivity, whereas estimated sensitivity was used to measure the quality of MegaBLAST seeds. As discussed before, sensitivity calculation is exponential in runtime and memory consumption and the lengthy MegaBLAST seeds consumed more memory than our system had. SpEED2 is able to detect if the system would run out of memory and crash while computing the sensitivity and would calculate estimated sensitivity instead.

```

Generating 4 seeds of weight 10 for similarity level 0.75 and length of homology region 50
SpEED-fast seed set is:
1101101011111
11010100000110010111
11100010010010000101011
11100000101000010000110011

Computed in 0.022598 seconds
Real Sensitivity is 0.904245

The program starts computing better seeds ...
If you reach a set of seeds with your desired sensitivity you can kill the program ...

--- random try number 2 ---
seeds:
1111010110111
11100100011010111
1111000101000101011
110100110000001000010001011
Real sensitivity = 0.906206
time (since beginning): 1.90463

--- random try number 5 ---
seeds:
11110011010111
11101001000110111
11101000100101000111
11011000010000000100010111
Real sensitivity = 0.908047
time (since beginning): 2.43364

```

Figure 4.1: Screenshot of SpEED2 output

SpEED2 is designing 4 seeds of weight 10 with similarity level 0.75 and region length 50.

## 4.5 Comparison

In this section, we compare the seeds designed by SpEED2 with the seeds designed by other leading software with the help of sensitivity and estimated sensitivity. The multiple spaced seeds we found are available in Appendix A. As shown in Table 4.2, SpEED2 performs very well when compared with other leading software like Iedera [15, 16, 17], SpEED [7, 13, 14], AcoSeed [18], and rasbhari [19]. The dataset used is shown in Table 4.1.

In each row, the value with the highest sensitivity is in bold font with a dark green background. The second highest sensitivity value for each row is in pale green. SpEED2 performs well over all our datasets, producing the best seeds in all 26 cases. Also, it is evident that rasbhari provides the second best seeds in most of the datasets.

Nowadays, longer seeds like that of MegaBLAST are more commonly used for similarity search as they can be used to query huge genome databases relatively quickly. SpEED2 and SpEED are the only software programs able to design MegaBLAST seeds, all others fail (mostly due to a lack of memory). We already know that computing sensitivity is a very expensive operation both in terms of memory and runtime (both exponential). Naturally, systems run out of memory trying to calculate sensitivity of lengthy MegaBLAST seeds. SpEED2 overcomes this by using estimated sensitivity which consumes constant amount of memory.

Now, we shift our focus to rasbhari. The seeds computed by rasbhari are very close in terms of sensitivity to those computed by us. The improvement achieved by our program might not seem very significant initially, but, one must keep in mind that in the context of read alignment, a 100-fold coverage of the human genome, a 1 percent improvement in seed sensitivity would

mean that 3 billion more nucleotides could be mapped [7].

Test		Sensitivity				
Weight	Similarity	Iedera	SpEED	AcoSeeD	rasbhari	SpEED2
<b>SHRiMP - 4 seeds and Homology Length - 50</b>						
10	0.75	90.6820	90.9026	90.9513	90.9614	<b>90.9680</b>
10	0.80	97.7586	97.8384	97.8521	97.8554	<b>97.8584</b>
10	0.85	99.7437	99.7575	99.7614	99.7618	<b>99.7624</b>
11	0.75	83.2413	83.3570	83.4728	83.4679	<b>83.5077</b>
11	0.80	94.9350	95.0351	95.037	95.0386	<b>95.0547</b>
11	0.85	99.2189	99.2476	99.2478	99.2506	<b>99.2510</b>
12	0.80	90.3934	90.6389	90.6328	90.6648	<b>90.6672</b>
12	0.85	98.0781	98.1514	98.1766	98.1824	<b>98.1858</b>
12	0.90	99.8773	99.8836	99.8853	99.8864	<b>99.8872</b>
16	0.85	84.5795	84.8347	84.9829	84.969	<b>84.9974</b>
16	0.90	97.2806	97.4386	97.4712	97.5035	<b>97.5162</b>
16	0.95	99.9331	99.9405	99.9419	99.9441	<b>99.9453</b>
18	0.85	72.1695	73.1478	73.2700	73.2209	<b>73.2977</b>
18	0.90	93.0442	93.7651	93.7778	93.7800	<b>93.7856</b>
18	0.95	99.6690	99.7547	99.7599	99.7557	<b>99.7633</b>
<b>PatternHunter II - 16 seeds and Homology Length - 64</b>						
11	0.70	92.0708	93.3406	-	93.4653	<b>93.4892</b>
11	0.75	98.3391	98.7156	-	98.7573	<b>98.7624</b>
11	0.80	99.8366	99.8859	-	99.8907	<b>99.8910</b>
<b>BFAST - 10 seeds and Homology Length - 50</b>						
22	0.85	60.1535	60.9329	-	60.9919	<b>61.0326</b>
22	0.90	87.9894	88.7120	-	88.8005	<b>88.8499</b>
22	0.95	99.2196	99.3959	-	99.4099	<b>99.4216</b>
<b>MegaBLAST - 1, 2, 4, 8. 16 seeds and Homology Length - 100</b>						
28	0.90	-	69.3208	-	-	<b>69.3780</b>
28	0.90	-	79.6679	-	-	<b>81.2266</b>
28	0.90	-	87.5677	-	-	<b>89.5934</b>
28	0.90	-	92.7762	-	-	<b>94.7527</b>
28	0.90	-	95.9170	-	-	<b>97.5533</b>

Table 4.2: Comparison of different programs with SpEED2

Another aspect to keep in mind is that rasbhari is much more expensive in terms of memory requirement as well as runtime when compared to SpEED2 because rasbhari seed are consistently 70% to 80% longer. As a result, rasbhari can not compute MegaBLAST seeds and takes more than 16 times longer to compute seeds when compared to our program.

# Chapter 5

## Conclusion and Future Works

In this chapter, we will discuss the summary of our work and possible future works to achieve further improvements.

### 5.1 Summary

In this thesis, we have considered the problem of designing multiple spaced seeds for similarity search. We have discussed various sequence alignment techniques including both traditional and heuristic approaches and looked at some of the most popular algorithms and programs. Then we discussed the topic of multiple spaced seeds along with some prominent software and their shortcomings.

We then presented our work, which generates highly sensitive seeds and then discussed how this increased performance is achieved by using indel optimization along with adaptive seed length. Finally, we compare the performance of our algorithm with other state-of-the-art software like Iedera, SpEED, AcoSeed, and rasbhari.

### 5.2 Conclusion

Nearly half a century has elapsed since Needleman-Wunsch's algorithm for global alignment, still, sequence similarity search remains an active field of research. The main reason for this is that sequence similarity helps to discover the evolutionary, structural, and functional relationship between unknown biological sequences by comparing them with known ones. Also, biological repositories, such as GenBank of NCBI, are ever-increasing and efficient algorithms are constantly required to make sense of such large quantities of data.

Seeding based approach is very popular when it comes to searching biological repositories. For this technique to work, we must use multiple spaced seeds of high sensitivity. Our work produces seeds of very high sensitivity within an acceptable amount of time.

The contribution of SpEED2 is that it generates good multiple spaced seeds and this, in turn, helps to perform tasks that require similarity search like gene and protein predictions, phylogeny, and evolutionary analysis, read mapping and primer design more accurately. Programs like PatternHunter which is used for homology search, SHRiMP, and BFAST used for

read mapping, and bestPrimer used for designing primers will benefit by using our software program.

## 5.3 Future Works

There is plenty of scope for future works. Firstly, we can make the program a lot faster by parallelization. Estimating the sensitivity and indel optimization are suitable candidates where we could parallelize the execution since the loops are pretty much independent of each other.

Secondly, there is scope for improving the design of our software program. Currently, SpEED2 can only handle up to a homology length of 128, but, in future, we would like it to be able to work with longer regions.

Thirdly, we could also employ deep learning models in our future works to understand hidden features behind designing highly sensitive multiple spaced seeds. It could produce even better seeds.

# Bibliography

- [1] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and David L Wheeler. Genbank. *Nucleic acids research*, 33(suppl\_1):D34–D38, 2005.
- [2] NCBI Web Traffic. <https://www.ncbi.nlm.nih.gov/stats/>.
- [3] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.
- [4] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.
- [5] Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
- [6] Ming Li, Bin Ma, Derek Kisman, and John Tromp. Patternhunter ii: Highly sensitive and fast homology search. *Journal of bioinformatics and computational biology*, 2(03):417–439, 2004.
- [7] Lucian Ilie and Silvana Ilie. Multiple spaced seeds for homology search. *Bioinformatics*, 23(22):2969–2977, 2007.
- [8] Biological Sequences. <https://www.ncbi.nlm.nih.gov/IEB/ToolBox/SDKDOCS/BIOSEQ.HTML>.
- [9] DNA vs RNA – 5 Key Differences and Comparison. <https://www.technologynetworks.com/genomics/lists/what-are-the-key-differences-between-dna-and-rna-296719>.
- [10] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- [11] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.
- [12] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.
- [13] Lucian Ilie, Silvana Ilie, and Anahita Mansouri Bigvand. Speed: fast computation of sensitive spaced seeds. *Bioinformatics*, 27(17):2433–2434, 2011.

- [14] Silvana Ilie. Efficient computation of spaced seeds. *BMC research notes*, 5(1):123, 2012.
- [15] Gregory Kucherov, Laurent No  , and Mikhail Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. *Journal of bioinformatics and computational biology*, 4(02):553–569, 2006.
- [16] Iedera :: subset seed design tool. <https://bioinfo.lifl.fr/yass/iedera.php>.
- [17] Gregory Kucherov, Laurent No  , and Mikhail Roytberg. Subset seed automaton. In *International conference on implementation and application of automata*, pages 180–191. Springer, 2007.
- [18] Dong Do Duc, Huy Q Dinh, Thanh Hai Dang, Kris Laukens, and Xuan Huan Hoang. Acoseed: An ant colony optimization for finding optimal spaced seeds in biological sequence search. In *International Conference on Swarm Intelligence*, pages 204–211. Springer, 2012.
- [19] Lars Hahn, Chris-Andr   Leimeister, Rachid Ounit, Stefano Lonardi, and Burkhard Morgenstern. Rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. *PLoS computational biology*, 12(10):e1005107, 2016.
- [20] Michael Brudno. Sequence Alignment Notes (Needleman-Wunsch, Smith-Waterman). <https://www.cs.utoronto.ca/brudno/bcb410/lec2notes.pdf>.
- [21] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.
- [22] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.
- [23] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.
- [24] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [25] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.
- [26] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge university press, 1997.
- [27] Ralf Himmelreich, Helga Plagens, Helmut Hilbert, Berta Reiner, and Richard Herrmann. Comparative analysis of the genomes of the bacteria mycoplasma pneumoniae and mycoplasma genitalium. *Nucleic Acids Research*, 25(4):701–712, 1997.
- [28] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron. *q*-gram based database searching using a suffix array (quasar). 1998.



- [29] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.
- [30] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden. Blast+: architecture and applications. *BMC bioinformatics*, 10(1):421, 2009.
- [31] Thomas L Madden, Roman L Tatusov, and Jinghui Zhang. Applications of network blast server. In *Methods in enzymology*, volume 266, pages 131–141. Elsevier, 1996.
- [32] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.
- [33] Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L Madden, Richa Agarwala, and Alejandro A Schäffer. Database indexing for production megablast searches. *Bioinformatics*, 24(16):1757–1764, 2008.
- [34] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE, 1999.
- [35] Jeremy Buhler, Uri Keich, and Yanni Sun. Designing seeds for similarity search in genomic dna. *Journal of Computer and System Sciences*, 70(3):342–363, 2005.
- [36] Lavinia Egidi and Giovanni Manzini. Multiple seeds sensitivity using a single seed with threshold. *Journal of bioinformatics and computational biology*, 13(04):1550011, 2015.
- [37] Matei David, Misko Dzamba, Dan Lister, Lucian Ilie, and Michael Brudno. Shrimp2: sensitive yet practical short read mapping. *Bioinformatics*, 27(7):1011–1012, 2011.
- [38] Nils Homer, Barry Merriman, and Stanley F Nelson. Bfast: an alignment tool for large scale genome resequencing. *PloS one*, 4(11):e7767, 2009.

# Appendix A

## SpEED2 Seeds

### A.1 SHRiMP Seeds

Similarity = 0.75
1111*1*11*111
111*1**11***1111
111*1*1***1**1**111
11*11****1****1***1*1*11
Sensitivity = 0.909680

Table A.1: SHRiMP seeds of weight 10 and homology length 50

Similarity = 0.8
111*11*11*111
111**1*1*1***1111
111*1*1***1****11*11
111***1**1**1****1*1*11
Sensitivity = 0.978584

Table A.2: SHRiMP seeds of weight 10 and homology length 50

Similarity = 0.85
111*11*11*111
1111***1*1*1**111
11*11****1***1*1*111
11*1*1****1**1**1***111
Sensitivity = 0.997624

Table A.3: SHRiMP seeds of weight 10 and homology length 50

Similarity = 0.75
111*11*11**1111
1111***1*1*11**111
111*1*1**1****1**1*111
111***1**1***1****1*1*111
Sensitivity = 0.835077

Table A.4: SHRiMP seeds of weight 11 and homology length 50

Similarity = 0.8
111*11*11*1*111
111**1*1***11*1111
111*1***11**1**1*111
1111***1**1****1***1*111
Sensitivity = 0.950547

Table A.5: SHRiMP seeds of weight 11 and homology length 50

Similarity = 0.85
111*11*11*1*111
1111**1*1***11*111
111*1***1**1****1**1111
111*1****1****1****1***1*111
Sensitivity = 0.992510

Table A.6: SHRiMP seeds of weight 11 and homology length 50

Similarity = 0.8
1111*1*11*11*111
111*1**11**1*1*1111
111*11***1****1**1**1111
1111**1*****1*****1**1***1*111
Sensitivity = 0.906672

Table A.7: SHRiMP seeds of weight 12 and homology length 50

Similarity = 0.85
111*11*111*1*111
111*1**1**1*1**1*1111
111*11***1**1***1*1**111
1111**1*****1***1*****1*1*111
Sensitivity = 0.981858

Table A.8: SHRiMP seeds of weight 12 and homology length 50

Similarity = 0.9
111*11*111*1111
1111**1*1**1**1*1111
111*1***1**1***1*1**1111
1111**1*****1***1*****1**1*111
Sensitivity = 0.998872

Table A.9: SHRiMP seeds of weight 12 and homology length 50

Similarity = 0.85
1111*11*11*111*1*1111
1111*1*1*1****111**11*1111
111*11*1***11**1****11*1*1111
11111****1**1**1***1*1****1**11111
Sensitivity = 0.849974

Table A.10: SHRiMP seeds of weight 16 and homology length 50

Similarity = 0.9
1111*1*11*11*111*1111
11111**11**1*1*11**1*1111
1111*1*1***11****1***11**11111
1111*11***1*****1**1***1*****1*1*1111
Sensitivity = 0.975162

Table A.11: SHRiMP seeds of weight 16 and homology length 50

Similarity = 0.95
111*111*11*11*1*11111
111111***1**1**1*1***11*1111
1111***111*1**1***1***1**1*1111
1111*11****1***1*****1*1*****11*1*111
Sensitivity = 0.999453

Table A.12: SHRiMP seeds of weight 16 and homology length 50

Similarity = 0.85
1111*111*1*111*11*11111
11111**11*111**1**1*1*11111
11111*1**1**1***111**1*1**11111
111111***11***1*1*****1***1**11*1111
Sensitivity = 0.732977

Table A.13: SHRiMP seeds of weight 18 and homology length 50

Similarity = 0.9
11111*1*11*111*111*1111
111*11*1*111*****11*1**111111
1111*11*11*****1***11**1*1*1*1111
11111*1***1***1**1***1***1***1*11*1111
Sensitivity = 0.937856

Table A.14: SHRiMP seeds of weight 18 and homology length 50

Similarity = 0.95
11111*1*11*11**111*11111
11111**11***1*1*1*11*1**11111
1111*1*1*1**1*11*****111**11*111
1111*1*1**1**1*****11**1*****111*1111
Sensitivity = 0.997633

Table A.15: SHRiMP seeds of weight 18 and homology length 50

## A.2 PatternHunter II Seeds

Similarity = 0.7
1111*1*111*111
11*11***1*11*1**111
11*11*11***1*1*111
111*1*1**1***1*11*11
11*1**1*1**11**1*1*11
11*1*1***11***1**1*111
111**11*****11***1**111
111*****1*1*1***1***1111
11*1*1*****1**1**1**1**111
11*11**1*****1***1**1*1*11
111**1***1***1*1***11*11
111*1*****1*1*****11***111
11*1*1*****1**1*****1*****1*111
1111***1*****1***1*****1*1*11
111***1***1*****1*****1**1***111
111**1**1*****1*****1*1**111
Sensitivity = 0.934892

Table A.16: PatternHunter II seeds of weight 11 and homology length 64

Similarity = 0.75
1111***1*11*11*11
111*11*1***11**111
11*11***11*1*1*1*11
11**1*1*1***11***1111
111*1**1*11*****11*11
111***11**1***1*1**1*11
11*11*1*****1**1*1***111
111*1*1***1***1*1*1*11
11*1*1*1*****1***1***1**111
11*1**1***1***1***1*1**11
111*1***1***1*****1**1**111
111***1***1***1***1***1*111
111*1***1***1***1***1*1*11
111**1***1***1*****1***1*111
111*1***1***1***1***1***1*11
11*1**1*****1*****1*****1*111
111*1***1***1***1***1***1*11
11*1**1*****1*****1*****1*111
111*1***1***1***1***1***1*111
Sensitivity = 0.987624

Table A.17: PatternHunter II seeds of weight 11 and homology length 64

Similarity = 0.8
111*111*1*11*11
111*1*1***1**111*11
111*11*11**1***111
111***11*1*1***1*111
11*11*1***111**1*11
111**1**11***11**111
1111**1***1*1***1*111
11*1***1*11*****11*1*11
11*11***1*1***1**1*111
111***11***1***1***1*111
11*1***1*1***1***1*1*111
111*1***1***1*1***11*11
11*1*1*1***1***1***1***111
11*1**1***1***1***1***1*111
111**1***1***1***1***1*11
1111***1***1*****1***1*1*11
Sensitivity = 0.998910

Table A.18: PatternHunter II seeds of weight 11 and homology length 64



## A.3 BFAST Seeds

Similarity = 0.85
1111*11111*1111111*11*1111
1111111*111*11**1*11111*1111
1111*111*1*1**111*1111**1*11111
1111*11**11111*1*11***1*111*1111
11111*1*1*11*111***111**11**11111
11111*111***1***111*11*1*1*11*1111
1111**11**111*1**1**1*1*1**111**11111
1111*1**1*11***11*11***11*1*1*111111
1111*11*11***11*1***11**1*11**11*1111
11111*11***11*1**1***1*1**1***11**111111
Sensitivity = 0.610326

Table A.19: BFAST seeds of weight 22 and homology length 50

Similarity = 0.9
1111*1111*111*111111*11111
1111111**11*111*11*1**111*1111
1111*1*11111*11**1*1*11**111111
11111*111**1*11*1*11*111***11111
1111*11**1*11***11111*11*11*1111
11111**11*1*1**111***1*1**11*111111
1111*11*1***111*1*1*1***111*1**11111
11111*11*1*11***1***11*11***1*11*1111
11111*1*1*11***1*11*1*1**11***11*1111
1111*111***11*11**1***1***1*1*1*111111
Sensitivity = 0.888499

Table A.20: BFAST seeds of weight 22 and homology length 50

Similarity = 0.95
11111*11111*1111*111*11*111
1111*111**1*111*11*11111*1111
1111*11*1*111*111***111*111*111
11111*1*1111***1*11*11*1**1*11111
11111*1*1***11*1*11*1***1111*11111
1111**111*11**1**1***1*111*11**11111
1111*11**1*1**11111***11**1*1*1*1111
11111*11**1**11***1*1*11*1***1111*111
1111*1**1*1*1**1**111***1**11**1**111111
111111*1*1***11***1*****11**1**1*1**111111
Sensitivity = 0.994216

Table A.21: BFAST seeds of weight 22 and homology length 50

## A.4 MegaFAST Seeds

Seeds : 1
11111*1*111***11**11*11**11*1*11*1*11*11111
Estimated sensitivity = 0.69378

Table A.22: MegaFAST seeds of weight 28, homology length 100 and sensitivity of 0.9

Seeds : 2
11111*1*11*111*1*111**11*111**1*11*11111
111111**11*11**1***11***1**1**1*1***1*1***11*1*1*11111
Estimated Sensitivity = 0.812266

Table A.23: MegaFAST seeds of weight 28, homology length 100 and sensitivity of 0.9

Seeds : 4
1111*111*1111**1*11*111*11*1*111**11111
1111111**1***11*1*1**111***1*1*1**11*1*11*11111
11111*1*11***1*11**1***1*11**11***1*1**11**111*11111
11111*1*11*1*1***1***1**1**1***1***1***11***11***1*11*11111
Estimated sensitivity = 0.895934

Table A.24: MegaFAST seeds of weight 28, homology length 100 and sensitivity of 0.9



# Curriculum Vitae

**Name:** Arnab Mallik

**Post-Secondary Education and Degrees:** West Bengal University of Technology  
Kolkata, India  
2011 - 2015 B.Tech.

University of Western Ontario  
London, ON, CA  
2018 - 2019 M.Sc.

**Honours and Awards:** Western Graduate Research Scholarship  
2018-2019

**Related Work Experience:** Teaching Assistant  
The University of Western Ontario  
2018 - 2019

Software Engineer  
Infosys India Ltd.  
2015 - 2018