2011

A FAST ALGORITHM FOR COMPUTING HIGHLY SENSITIVE MULTIPLE SPACED SEEDS

Anahita Mansouri Bigvand

A FAST ALGORITHM FOR COMPUTING HIGHLY SENSITIVE MULTIPLE SPACED SEEDS

(Spine title: A Fast Algorithm for Computing Good Multiple Spaced Seeds)

(Thesis format: Monograph)

by

Anahita Mansouri Bigvand

Graduate Program in Computer Science

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

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Supervisor:

.....
Dr. Lucian Ilie

Joint Supervisor:

.....
Dr. Silvana Ilie

Examiners:

.....
Dr. Hanan Lutfiyya

.....
Dr. Kamran Sedig

.....
Dr. Stuart Rankin

The thesis by

Anahita Mansouri Bigvand

entitled:

A Fast Algorithm for Computing Highly Sensitive Multiple Spaced Seeds

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

.....
Date

.....
Chair of the Thesis Examination Board

Abstract

The main goal of homology search is to find similar segments, or local alignments, between two DNA or protein sequences. Since the dynamic programming algorithm of Smith-Waterman is too slow, heuristic methods have been designed to achieve both efficiency and accuracy. Seed-based methods were made well known by their use in BLAST, the most widely used software program in biological applications. The seed of BLAST trades sensitivity for speed and spaced seeds were introduced in PatternHunter to achieve both. Several seeds are better than one and near perfect sensitivity can be obtained while maintaining the speed. Therefore, multiple spaced seeds quickly became the state-of-the-art in similarity search, being employed by many software programs. However, the quality of these seeds is crucial and computing optimal multiple spaced seeds is NP-hard. All but one of the existing heuristic algorithms for computing good seeds are exponential. Our work has two main goals. First we engineer the only existing polynomial-time heuristic algorithm to compute better seeds than any other program, while running orders of magnitude faster. Second, we estimate its performance by comparing its seeds with the optimal seeds in a few practical cases. In order to make the computation feasible, a very fast implementation of the sensitivity function is provided.

Keywords: Similarity search, multiple spaced seeds, overlap complexity, sensitivity.

Acknowlegements

I would like to thank my supervisors Lucian Ilie and Silvana Ilie for their support, encouragement, and guidance. Lucian Ilie has been abundantly helpful in developing new ideas on this research. He carefully read many drafts of this thesis and his comments have greatly improved the presentation.

I would also like to express my deepest gratitude to my parents for their endless love, encouragement and support.

My greatest appreciations go to Mojtaba Kheiri for his unflagging love and support throughout my studies.

Contents

Certificate of Examination ii

Abstract iii

Acknowlegements iv

List of Figures vii

List of Tables viii

List of Appendices ix

1 Introduction 1

2 Seeds 3
- 2.1 Basic Definitions . . . 3
- 2.2 BLAST . . . 4
- 2.3 Other Similarity Search Softwares . . . 5
- 2.4 Spaced Seeds . . . 5
- 2.5 Multiple Spaced Seeds . . . 6
- 2.6 Computing Sensitivity of Multiple Seeds . . . 7
- 2.7 Other Types of Seeds . . . 9
  - 2.7.1 Different Filtration Types: Lossy and Lossless . . . 10
  - 2.7.2 Vector Seeds . . . 11
  - 2.7.3 Transition Constrained Seeds . . . 13
  - 2.7.4 Subset Seeds . . . 14
- 2.8 Applications of Seeds in Software Programs . . . 15

3 Algorithms for Computing Seeds 17
- 3.1 Hardness of the Problem . . . 17
- 3.2 Mandala . . . 18
  - 3.2.1 Mandala's Problem . . . 18
  - 3.2.2 Local Search Method . . . 18
  - 3.2.3 Greedy Covering Algorithm . . . 19
- 3.3 Iedera . . . 19
- 3.4 PatternHunter . . . 20
- 3.5 BFAST . . . 20

4 SpEED **22**
4.1 Overlap Complexity . . . 22
4.1.1 Definition . . . 22
4.1.2 Polynomial Time Algorithm . . . 24
4.2 Finding Good Lengths . . . 25
4.2.1 Reducing the Number of Lengths to be Guessed . . . 26
4.2.2 Finding Proper Min and Max . . . 28
4.3 The Engineered Algorithm . . . 28
4.3.1 Preprocessing and Analysis of Data . . . 28
4.3.2 SpEED . . . 32

5 Experiments **34**
5.1 PatternHunter II . . . 34
5.2 BFAST . . . 35
5.3 SHRiMP . . . 36
5.3.1 SHRiMP - weight 10 . . . 36
5.3.2 SHRiMP - weight 11 . . . 37
5.3.3 SHRiMP - weight 12 . . . 38
5.3.4 SHRiMP - weight 16 . . . 39
5.3.5 SHRiMP - weight 18 . . . 39
5.4 PerM . . . 40
5.5 SToRM . . . 41

6 Evaluation of Overlap Complexity **42**
6.1 Limitations on Exhaustive Search . . . 42
6.2 Fast Computation of Sensitivity . . . 43
6.3 Tests . . . 44

7 Conclusion **46**

Bibliography **47**

A Lists of Arrays **50**

Curriculum Vitae **55**

List of Figures

2.1 BLAST vs PaterHunter seeds 6
2.2 Dynamic programming algorithm for sensitivity 8

4.1 An example showing the intuition behind overlap complexity 23
4.2 An example of overlap complexity of two seeds 24
4.3 The MULTIPLESEEDS algorithm 25
4.4 Intermediate seeds for finding PatternHunter's seed 25
4.5 The MAKELENGTH algorithm 26
4.6 The COMPUTESEEDSWITHMINMAX algorithm 27
4.7 The MINMAX algorithm 29
4.8 The IDENTIFYMOVE algorithm 30
4.9 The SINGLEMOVE algorithm 30
4.10 The *min* and *max* values for $k = 2$ 31
4.11 The *min* and *max* values for $k = 3$ 31
4.12 The *min* and *max* values for $k = 4$ 31
4.13 The *min* and *max* values for $k = 10$ 32
4.14 The SPEED algorithm 33
4.15 The PRECOMPUTEDMINMAX function 33

6.1 The elements of B^r for the seed $1 * 1 * 1$. 43

List of Tables

2.1 The f table. 9

5.1 Results of SpEED versus Mandala for PatternHunter's parameters 35
5.2 Results of SpEED versus Mandala for BFAST's parameters 36
5.3 Comparing SpEED with Mandala for SHRiMP of $w = 10, N = 35$ 37
5.4 Comparing SpEED with Mandala for SHRiMP of $w = 10, N = 50$ 37
5.5 Comparing SpEED with Mandala for SHRiMP of $w = 11, N = 35$ 37
5.6 Comparing SpEED with Mandala for SHRiMP of $w = 11, N = 50$ 38
5.7 Comparing SpEED with Mandala for SHRiMP of $w = 12, N = 35$ 38
5.8 Comparing SpEED with Mandala for SHRiMP of $w = 12, N = 50$ 38
5.9 Comparing SpEED with Mandala for SHRiMP of $w = 16, N = 35$ 39
5.10 Comparing SpEED with Mandala for SHRiMP of $w = 16, N = 50$ 39
5.11 Comparing SpEED with Mandala for SHRiMP of $w = 18, N = 35$ 39
5.12 Comparing SpEED with Mandala for SHRiMP of $w = 18, N = 50$ 40
5.13 Comparing SpEED with Mandala for PerM of $w = 12, N = 35$ 40
5.14 Comparing SpEED with Mandala for PerM of $w = 12, N = 50$ 40
5.15 Comparing SpEED with Mandala for SToRM of $w = 12, N = 35$ 41
5.16 Comparing SpEED with Mandala for SToRM of $w = 12, N = 50$ 41

6.1 Optimal seed versus seeds computed by overlap complexity 44
6.2 Optimal seed versus seeds computed by overlap complexity 45
6.3 Optimal seed versus seeds computed by overlap complexity 45

List of Appendices

Appendix A 50

Chapter 1

Introduction

The main goal of homology search is to find similar segments or local alignments between two biological sequences, such as DNA or protein sequences. As the classical dynamic programming method of Smith-Waterman [34] is too slow to be practical, heuristic algorithms such as BLAST [1] are used. BLAST uses a filtration technique in which positions with short consecutive matches between two sequences, or *hits*, are identified first and then extended into local alignments. Alignments with high scores are reported, while those with low scores are discarded. These short sequence matches are called seeds. In similarity search, *seeds* are short sequence motifs which, if shared by two sequences, are assumed to witness a potential similarity.

While classically, contiguous seeds have been used, *spaced seeds*, introduced by Califano and Rigoutsos [9] and popularized by PatternHunter [26], where the hits were no longer required to consist of consecutive matches, have been shown to be more sensitive. Indeed, PatternHunter looks for 18 consecutive nucleotides in each sequence such that only those positions that are specified by 1's in the string $111 * 1 * *1 * 1 * *11 * 111$ are required to match. The other positions are actually don't cares. This string is called a spaced seed. BLAST seed is a contiguous seed and it can be represented by 11 consecutive matches or 11111111111. The number of 1's is called the weight of a seed.

Seeds are used in the framework of pattern matching. To find approximate matches of a given string in a sequence, we can discard those parts of the sequence where matching can not occur. Such filtration is done by small patterns (seeds). Therefore, one of the main applications of seeds is in approximate string matching. Seeds are also used in similarity search to find alignments between biological sequences.

Multiple spaced seeds are an extension to the basic seed model. Multiple spaced seeds is a set of seeds that hits when any of the seeds hits. Multiple spaced seeds perform better in a variety of applications. They provide faster and more sensitive homology search [31, 26]. Multiple spaced seeds can be used for finding homologous coding regions in DNA sequences [4]. They also perform well in applications like the problem of oligonucleotide selection [22].

Finding optimal (multiple) spaced seeds is NP-hard but even finding good ones is very difficult. Therefore heuristic algorithms are used to find good sets of seeds. Exhaustive search involves two exponential-time steps:

- There are exponentially many seeds to be evaluated based on their sensitivities and

- Computing the sensitivity of each seed takes exponential time as well.

Heuristic algorithms such as those of [3, 36] tried to alleviate the second exponential problem by approximating the sensitivity. For the former, the number of seeds to be considered has been reduced by various heuristics like those of [12, 37] but their algorithms were still exponential.

Heuristic algorithms that were proposed were all exponential in theory and slow in practice. Hence, there are no fast software programs for computing seeds. However, the algorithm of [18] is the only one that computes good multiple spaced seeds in polynomial time and it is the focus of this thesis. It introduces the overlap complexity measure that has turned out to be well correlated with sensitivity but it is much easier to compute. It takes polynomial time instead of the exponential time required for sensitivity. Therefore, it can be used instead of sensitivity in computations.

Our contributions in this thesis are two fold. First, there has been no algorithm to compute better seeds than the one of [18]. We intend to engineer this algorithm in order to improve the quality of the seeds. Our goal is to provide the fastest and most sensitive software program for computing seeds.

Our second goal is to assess the overlap complexity measure. In other words, we study how close to optimal is the sensitivity of the seeds produced by our programs. Or, differently put, how much our algorithm can be improved. We performed several meaningful exhaustive tests and compared the optimal results with the results of our program.

The thesis is organized as follows. In Chapter 2, we introduce seeds. Multiple spaced seeds, the sensitivity concept and an algorithm for computing it are presented in this chapter. Besides, this chapter includes other types of seeds and applications of spaced seeds. In Chapter 3, some algorithms that compute seeds are studied. In Chapter 4, we propose a new algorithm that is named SpEED. This algorithm solves the limitation of the algorithm proposed in [18]. The algorithm of [18] does not provide seeds lengths for the general case. Therefore, our approach is to propose a heuristic algorithm to find a set of length for a set of seeds that we want to design. We propose a modified polynomial time algorithm that generates seeds, which uses the algorithm that finds good seed lengths. Chapter 5 includes our experiments. We compute seeds with the SpEED algorithm to improve the seeds that have computed or used by different software programs, namely, PatternHunter II [26], BFAST [16], SHRiMP [33], PerM [11] and SToRM [15]. We compared our program with Mandala software (the best software that computes seeds) in terms of sensitivity and time for each of the mentioned test cases. Indeed, we compute seeds for the parameters of those software programs but with different similarity levels both with our program and with Mandala to compare our seeds with Mandala's. Our seeds are more sensitive in all cases and our program is several orders of magnitude faster. This is why, in some cases, such as BFAST, our program computes seeds in seconds whereas Mandala could not finish in one day. We also improve all the seeds that are used by those software programs mentioned. In Chapter 6, we evaluate the performance of our algorithm. We perform exhaustive testing for some feasible test cases and investigates how close to optimal is the sensitivity of the seeds produced by our programs. The results confirm that sensitivities of our seeds are very close to optimal sensitivities in all three cases. The most important contribution in this part is a fast implementation of the sensitivity function so that exhaustive testing becomes possible.

Chapter 2

Seeds

2.1 Basic Definitions

DNA, or deoxyribonucleic acid, is the hereditary material in humans and almost all other organisms. It contains the genetic instructions used in the development and functioning of all known living organisms with the exception of some viruses. The main role of DNA molecules is the long-term storage of information. The information in DNA is stored as a code made up of four chemical bases called nucleotides: adenine (A), guanine (G), cytosine (C), and thymine (T).

Proteins are organic compounds made of amino acids arranged in a linear chain. They are polymers and have an alphabet of 20 amino acids. A protein has several functions. It may serve as a structural material (e.g., keratin), as enzymes, as transporters (e.g., hemoglobin), as antibodies, or as regulators of gene expression.

For our purposes, a DNA molecule can be represented as a sequence of four different nucleotides A, C, G, T and a protein can be represented as a sequence of 20 different amino acids.

Similarity search is a task that compares a biological sequence such as a DNA sequence against another or to a database to find similar segments between them [19]. In other words, the main goal of this task, which is also called as homology search, is to find similar segments or local alignment between two DNA or protein sequences [26]. The concept of homology between two biological sequences is used to infer that two genes or their protein products are related by evolution. Homologous sequences are expected to have common functional role in enzymatic activity, cellular functions, or overall cellular processes. They may have common structural features, such as in their protein tertiary structure. Attributing structure, function, or process to a protein sequence experimentally can be expensive in time and effort. Therefore, biologists look at other sequences that are similar to predict homology and to infer these features. This approach has been used widely for structure prediction and function prediction [10].

Many programs have been developed for the similarity search task. BLAST [1] is the most widely used one.

2.2 BLAST

Since the dynamic programming method of Smith-Waterman [34] is too time consuming, heuristic approaches like BLAST were introduced. **B**asic **L**ocal **A**lignment **S**earch **T**ool, or BLAST, is an algorithm that approximates alignments that optimize a measure of local similarity, the maximal segment pair score. It follows the idea of hit-and-extend approach. BLAST first finds short exact matches, called *hits*. A BLAST hit consists of several consecutive positions (the default is 11). For each, an alignment is built that extends the hit on both sides. If the alignment score exceeds a threshold, the alignment is reported, otherwise it is discarded. Some significant alignments do not contain 11 consecutive matches; thus, they are not discovered by BLAST. To find hits, we create a hash table of all the words of length 11 in one of the sequences and then search for each word of length 11 from the other sequence in the table [1].

BLAST introduced the concept of contiguous seeds. BLAST seed can be represented as 11111111111. This seed is used as a pattern for determining hits. The performance of a particular seed can be characterized by its *sensitivity* and *specificity*. We need to recall a few concepts from statistics:

- *true positives* are alignments that contain hits (detected alignments),
- *true negatives* are subsequences that are not alignments and which do not contain hits (non-alignments correctly non-detected),
- *false positives* are subsequences that are not alignments but contain hits (non-alignments wrongly identified as alignments), and
- *false negatives* are alignments that do not contain hits (alignments that are missed).

Using these, we define

- $\textit{sensitivity} = \frac{\text{true positives}}{\text{all positives}} = \frac{\text{true positives}}{\text{true positives + false negatives}}$
- $\textit{specificity} = \frac{\text{true negatives}}{\text{all negatives}} = \frac{\text{true negatives}}{\text{true negatives + false positives}}$

Sensitivity is also called *true positive rate* and it is equal to $1-$ *false negative rate*. Specificity is also called *true negative rate* and it is equal to $1-$ *false positive rate*. A good seed has high sensitivity and specificity or, equivalently, low rates of false positives and false negatives.

As mentioned previously, there may be some significant alignments that are discarded by BLAST just because they do not contain 11 consecutive matches. This phenomenon increases the rate of false negatives, decreasing sensitivity.

On the other hand, also in the case of BLAST, just because two sequences happen to have 11 consecutive matches need not imply the existence of a significant alignment. This phenomenon causes false positives, thus decreasing specificity.

Obviously, each hit increases the running time since the algorithm attempts to extend it to a full alignment. Therefore, a seed with large false positive rate slow down the computations. Ideally, we want seeds with both low false positive and negative rates. Unfortunately, there is a trade-off between these two measures. Longer BLAST seeds have fewer false positives and more false negatives rather than shorter ones. BLAST deals with a dilemma of sensitivity

versus speed. While improving sensitivity needs shorter seeds, enhancing speed needs longer seeds. Mega-BLAST uses seeds of length 28 to improve the speed. It is much faster than BLAST but significantly less sensitive [38].

BLAST is actually a family of programs including BLASTN, BLASTP, PSI-BLAST and MegaBLAST and so on. BLASTN returns the most similar DNA sequence from database to the DNA query specified by the user. BLASTP compares protein queries to protein databases. PSI-BLAST has three major refinements on the original BLAST program. (i) "Two-hit" method was used and it requires the existence of two non-overlapping word pairs on the same diagonal, and within a distance A of one another, before an extension is invoked. (ii) The ability to generate gapped alignment has been added. (iii) BLAST searches can be iterated with a position-specific score matrix generated from significant alignments found in round i and used for round $i + 1$ [2].

2.3 Other Similarity Search Softwares

Many programs have been developed for the similarity search task. These include FASTA [30] that was proposed before BLAST, SIM [17], SENSI [35], MUMer [14], REPuter [25] and BLAT [20] and more recently PatternHunter [31]. Heuristic searches with FASTA, BLAST and SIM are slow for modern genomic data and miss many alignments. SENSI is faster but it works for ungapped alignments. Smith-Waterman approaches are too slow to be practically used. Softwares such as MegaBLAST, MUMer and BLAT were developed to increase the speed of BLAST for highly similar sequences. MUMer uses suffix trees. Using suffix trees may contribute to two problems. (i) They are meant to deal with precise matches. (ii) They need a large amount of space [31].

2.4 Spaced Seeds

PatternHunter introduced a novel seed model that impressively increases both sensitivity and speed. This means that the dilemma of BLAST type of search is solved by PatternHunter [31]. The important novelty of PatternHunter was the use of "optimal spaced seeds." The concept of spaced seed was introduced by Califano and Rigoutsos [9] and popularized by PatterHunter. Spaced seeds are often represented as a string of 1's and 0's, where 1's indicate required matches at those positions, while 0's indicate don't care positions or positions that may mismatch. While BLAST looks for matches of w consecutive letters as seeds, PatternHunter proposes to use nonconsecutive w letters as seeds. The number of 1's in the seed is called the *weight* of a seed. The *length* of a seed is its overall length or the number of 1's plus the number of 0's in a seed. The spaced seed used by PatternHunter is 111010010100110111 of weight 11 and length 18. Ma et al. [31] propose a simple probabilistic model of alignments to characterize the sensitivity of a spaced seed. In this model, a local alignment is represented as a binary sequence, in which 1 represents a match and 0 a mismatch. The probabilistic model has two parameters: N, the length of the alignment and p, the probability of a match. Therefore, it is a sequence of N independent Bernoulli random variables $X_0, X_1, \ldots X_{N-1}$ with $Pr(X_i = 1) = p$ for each i. Sensitivity of a seed is then the probability that the seed hits an

alignment sampled from this model. Computation of sensitivity is discussed in Section 2.6.

While the expected number of hits of the PatternHunter seed is the same as the expected number of hits of the BLAST seed, the probability of having at least one hit by PatternHunter seed is greater than that of the BLAST seed. This means that the PatternHunter seed is more sensitive. PatternHunter's seed, 111010010100110111, has the highest sensitivity of all seeds of weight 11 and length at most 20 in the Bernoulli alignment model with parameters N = 64 and p = 0.7 [31]. Its sensitivity is 47%, compared to the BLAST consecutive seed of the same weight, which has sensitivity only 30%. Even the BLAST seed of weight 10 has lower sensitivity, 41%, and four times higher false positive rate. Hence, by using the PatternHunter seed of weight 11, we may expect to find more alignments, in shorter time, than using the BLAST seed of weight 10.

We will give an argument for the big difference between the sensitivities of BLAST and PatternHunter seeds. The reason for increased sensitivity in PatternHunter is that the event of having a match at different positions are more independent for spaced seeds. If a model and its shifted copy shares many 1's in common, then a base mismatch in any position will make both matches fail. This is a simple example of non-independent events. To put it more simply, we consider the BLAST seed and $p = 0.7$. If a BLAST seed has a hit at position i of an alignment, it has probability 0.7 to have another hit at position $i+1$ since ten of the eleven required matches are guaranteed by the presence of a hit at position i. For the PatternHunter seed this probability is $0.7^6 = 0.117649$ since 6 more matches are required [6]. This is shown in Figure 2.1

Figure 2.1: For an alignment that has Bernoulli model with match probability p, if a BLAST seed has a hit at position i of the alignment, the probability of a hit at position $i + 1$ is p. That is because it needs one another additional required match (red 1). PatternHunter seed requires 6 additional matches (red 1's) and hence the probability of hit at position $i + 1$ is p^6

PatternHunter was used to compare human genome against the mouse genome at a speed over a hundred times faster than BLASTN at the same sensitivity.

2.5 Multiple Spaced Seeds

The design goal of PatternHunter II is to solve the sensitivity problem by using multiple spaced seeds. Multiple spaced seeds are sets of seeds that hit whenever one of them hits the sequence. PatternHunter II aims to achieve a sensitivity approaching that of Smith-Waterman with a speed similar to BLASTN. It was noticed in [26] that more spaced seeds can increase the sensitivity. We will define some concepts and notations that were used in [26].

We denote a spaced seed as a binary string. Let a be a seed. The length of seed a is denoted by $|a|$ and weight of the seed is denoted by $||a||$. The weight of a seed is the number if 1's in a seed, while the length of a seed is number of 1's plus number of 0's where 1 means a required match and a 0 means a don't care position. Since the weight of all seeds in a set of seeds are the

same, we can denote the weight of the multiple seeds by w. Since if the first and the last bits of a seed are 0s, they can be ignored in local alignment algorithms, we will focus our attention on seeds that have 1 at the beginning and at the end. As an example, BLAST seed can be represented as 11111111111. It has weight 11 and length 11. PatternHunter's default seed is 111010010100110111 with weight 11 and length 18.

A homologous region is similarly represented by a binary sequence with a 1 representing a match and a 0 representing a mismatch. The similarity p represents the probability of a 1. A substring from position i to j (excluding j) is denoted by $R[i : j]$. Hence, $R = R[0 : |R|]$. A seed hits a homologous region R, if R has a substring $R[j : j + |a|]$, such that $R[j + i] = 1$ whenever $a[i] = 1$ for $0 \leq i < |a|$. We also say that a hits R at position j.

For a multiple spaced seed $A = \{a_1, a_2, \ldots, a_k\}$, we say that A hits a region R if one of $a_i \in A$ hits R.

Based on the definitions, the hit probability of multiple seeds is obviously not less than the hit probability of any one of them. Therefore, multiple seeds will increase the sensitivity of homology search. However, the search program needs to examine all hits generated by all seeds that makes the program slower.

2.6 Computing Sensitivity of Multiple Seeds

Computing the hit probability of a single seed was investigated in [19]. Sensitivity is computed using a dynamic programming algorithm. We will give the algorithm that was used in PatternHunter II [26] for multiple spaced seeds that is the extension of the one proposed for a single seed in [19].

Let $A = \{a_1, a_2, \ldots, a_k\}$ be a multiple spaced seed and R a random homologous region of length L with similarity p. For a binary string b and $|b| \leq i \leq L$, we define:

$$f(i, b) = Pr(A \text{ hits } R[0 : i] \mid b \text{ is a suffix of } R[0 : i]). \tag{2.1}$$

Based on this definition, the hit probability of A on R is equal to $f(L, \epsilon)$ where ϵ is the empty string and L is the length of the homologous region. Therefore, for any $i > |b|$ we have:

$$f(i, b) = (1 - p)f(i, 0b) + pf(i, 1b). \tag{2.2}$$

Also, for $i = |b|$:

$$f(i, b) = \begin{cases} 1 & \text{if } A \text{ hits } R[0 : i] \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

The goal is to compute $f(i, b)$ in terms of other $f(i', b')$ values that were computed earlier and stored in a table.

If a suffix b of a region R is hit by A, then $f(i, b) = 1$.

Compatibility: A binary string b is compatible with a seed a if $b[|b| - j] = 1$ whenever $a[|a| - j] = 1$ for $0 < j \leq \min(|a|, |b|)$. Therefore, if a suffix b of a region R is not compatible with a, then a can not hit the tail of R.

Sensitivity (A, p, L)

input: A (seed set), p (similarity level), L (length of hit region)
output: The probability that A hits a random region of length L with p similarity level

```
1. compute the compatible suffix set B
2. for i from 0 to L do
3.     for b in B from longest to shortest do
4.         if i < |b|
5.             f(i, b) ← 0
6.         else if i = |b|
7.             if A hits b
8.                 f(i, b) ← 1
9.             else
10.                f(i, b) ← 0
11.        else
12.            f_0 ← f(i − |b| + |b'|, 0b'), where 0b' = B(0b)
13.            if A hits 1b
14.                f_1 ← 1
15.            else
16.                f_1 ← f(i, 1b)
17.            f(i, b) ← (1 − p)f_0 + pf_1
18. return f(L, ϵ)
```

Figure 2.2: The pseudocode of the dynamic programming algorithm for computing the sensitivity of a set of seeds.

Let B be the set of binary strings that are not hit by A but compatible with some $a \in A$. We define $B(x)$ as the longest proper prefix of x that is in B. We know that $\epsilon \in B$. We have 2 cases:

1. If $b \in B$, then b is compatible with some $a \in A$ and so is $1b$.
2. If $1b \notin B$, then it must hit by some $a' \in A$, and $f(i, 1b) = 1$

If $0b \notin B$, then it can not be hit by A. In other words, since $0b$ is not in B, it is incompatible with all $a' \in A$. In that case, $f(i, 0b)$ equals $f(i - |b| + |b'|, 0b')$, where $0b' = B(0b)$.

Using these notations, $f(i, b)$ can be computed by the dynamic programming algorithm given in Figure 2.2

Example: We use an example to clarify the algorithm. Suppose the seed is $A = \{101, 1001\}$ and the length of homologous region is $L = 4$. We want to compute the sensitivity using the algorithm that has been discussed.

First, we need to compute the B set. This set is the set of binary strings that are not hit by A but compatible with some $a \in A$. Therefore, for each $a \in A$, B set contains all binary strings with length smaller than a and compatible with it. For example, for $a = 101$, we can add 1, 01, 11 to B. Also, for $a = 1001$ we can add 1, 01, 11, 001, 011, 101, 111. However, 101 and 111 are hit by A. Therefore, they can not be in B. Hence, $B = \{\epsilon, 1, 01, 11, 001, 011\}$

Based on the algorithm, we compute the f table from the longest elements of B to the shortest ones. Hence, we should sort B set in decreasing length order and thus the first element that we compute is $f(0,011)$. As mentioned in the algorithm, $f(i,b) = 0$ if $i < |b|$. Therefore, for instance, $f(0,011) = f(0,001) = 0$. The f table is shown in 2.1.

	011	001	11	01	1	ϵ
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	p	p	p	p^2
4	p	p	$p + p(1-p)$	$p + p(1-p)$	$p + p(1-p)$	$p^2(3-2p)$

Table 2.1: The f table.

As an example to find the value of $f(4,\epsilon)$, based on the algorithm, we need to see what $B(0)$ is. Since $B(0) = \epsilon$, the value of $f_0 = f(3,\epsilon) = p^2$ and $f_1 = f(4,1) = p + p(1-p)$ are computed from the values of the f table that has already computed and $f(4,\epsilon) = (1-p)p^2 + p(p+p(1-p)) = p^2(3-2p)$.

The Sensitivity algorithm as mentioned in [26] has time complexity $O((|A| + M + L) \times \sum_{a \in A} |a| \times 2^{|a|-w})$ where A is the seed set and M is the maximum length of a seed in A.

Besides the algorithm mentioned here and the one of [19], there are other papers that have discussed computing sensitivity. To mention a few, Choi and Zhang also studied how to calculate the sensitivity of a spaced seed and proposed a new algorithm for identifying the optimal spaced seed [13]. In [4], one considers conserved regions determined by hidden Markov model particularly for coding regions. These regions have three-periodic structure and variation in conservation level in the region. A dynamic programming algorithm for sensitivity computation for HMM is discussed in this paper. The same authors introduced an extension to spaced seeds entitled as vector seeds in [5]. They gave an algorithm for computing sensitivity under their model. A general approach for computing sensitivity is proposed in [23]. This approach can be applied to any definitions of seeds. It treats three components of seed sensitivity problem: (i) a set of target alignments (ii) an associated probability distribution and (iii) a seed model. They are all specified by distinct finite automata. In [21], the concept of homogenous alignment is introduced and a dynamic programming algorithm is proposed for computing the sensitivity under this model.

2.7 Other Types of Seeds

Apart from spaced seeds, other concepts and extensions were proposed in this area. In this section, we will first describe different types of filtration - lossy and lossless. Then we will debate on some other types of seeds such as vector seeds, transition constrained seeds and subset seeds.

This section is not necessary for understanding the contribution of the thesis. It's purpose is to give a more general picture of this area.

2.7.1 Different Filtration Types: Lossy and Lossless

Filtering is a widely used technique in biological sequence analysis. It comes from approximate string matching. Indeed, to find approximate matches of a given string in a sequence, we can discard those parts of the sequence that matching can not occur. This filtration is done by small patterns (seeds). Two types of filtering should be distinguished:

- Lossless: A lossless filtration guarantees to detect all sequence fragments of interest. Lossless filtration has been investigated in the context of approximate string matching problem [8, 22].
- Lossy: A lossy filtration does not guarantee to detect all interesting sequence fragments. In other words, it may miss some of them but still tries to detect a majority of them. Local alignment algorithms usually use a lossy filtration.

In this subsection, we will briefly focus on the related lossless filtration while the rest of this thesis is mainly related to lossy filtration.

Recall that the two measures used to evaluate the efficiency of a lossy filtration are sensitivity and selectivity, defined before.

In lossless filtration, since there would be no parts of sequence missed by the filter (there is no false negatives), sensitivity is not used as a measure. Indeed, in this case the sensitivity is always equal to 1. Therefore, only the selectivity parameter makes sense and is hence the main characteristic of the filtration efficiency.

Clearly, the choice of patterns used in filtration is crucial. Gapped seeds (spaced seeds, gapped q-gram) have been shown to enhance the filtration efficiency over the traditional technique of contiguous seeds. In lossy filtration, using spaced seeds became popular since PatternHunter. In lossless filtration for approximate pattern matching, spaced seeds were studied in [8] and have been shown to improve the filtration efficiency. An extension of lossless single seed which was discussed in [8] was proposed in [22]. A family of seeds (multiple spaced seed) were used rather than a single seed. We mention some problems associated with lossless filtration here but we will not go into details.

In lossless filtration context, we will use another visual representation of seeds adopted in [8]. In this representation, seeds are words over the two-letter alphabet $\{\#, -\}$, where # occurs at all matching positions while - occurs at all positions in between. Two sequences are similar if the hamming distance between them is smaller than a certain threshold. For instance, sequences CACTCGT and CACACTT are similar within hamming distance 2. This similarity can be detected by the seed ## – # at position 2 or by the seed ### – # at position 1. A *similarity* of two sequences of length m is a binary word $w \in \{0,1\}^m$ that represents a sequence of matches (1's) and mismatches (0's). A match or hit of a seed on a similarity has been defined above. A seed Q is said to detect a similarity w if Q has at least one occurrence in w. Given a similarity length m and a number of mismatches k, consider all similarities of length m containing k 0's and $(m-k)$ 1's. These similarities are called (m,k)*–similarities*. A seed Q solves the (m,k)–problem iff all of $\binom{m}{k}$ (m,k)*–similarities* w are detected by Q. For example, seed # – ## – –# – ## solves the (15,2)-problem as mentioned in [22].

There is a strong correlation between weight of a seed and the selectivity of filtration procedure. With a larger weight less similarities pass through the filter as false positives become

smaller and therefore selectivity improves. Increasingly, a smaller weight reduces the filtration efficiency. Therefore, the goal is to solve an (m,k)–problem by a seed with the largest possible weight. Solving (m,k)–problem by a single seed is the problem that has been studied in [8] whereas [22] solves the (m,k)–problem by a family of seeds.

Multiple spaced seeds perform better than a single seed. In [8], it has been shown that a seed solving $(25,2)$– problem has the maximal weight 12. The only seed that has this property is:

– # – –### – # – –### –

Obviously, its reversal is also a seed with this property but we do not consider it as being different. However, the problem can be solved by the family of the following two seeds with weight 14:

– ## – – – ##### –
– ## – – – ##### – ## – – –

Clearly, two seeds of weight 14 have larger selectivity than one seed of weight 12 due to the decrease in the number of false positives.

A dynamic programming algorithm was proposed in [8] to compute the *optimal threshold* of a given seed. The optimal threshold of a given seed is the minimal number of its occurrences over all possible (m,k)–*similarities*. Also, an extension of this algorithm was proposed in [22]. The filtering efficiency of a q-gram clearly depends on the threshold. Small threshold means low filtering efficiency. Moreover, minimum coverage of a seed is another property of seeds that can affect the filtering efficiency. The minimum coverage is the minimum number of characters that need to match between a pattern and a text substring for there to be t matching q-grams. Small minimum coverage means a high filtering efficiency [8]. Several combinatorial results have been presented that allow us to construct efficient families composed of seeds with a periodic structure. Periodic seeds are obtained by iterating a smaller seed. Such seeds often turn out to be among the maximally weighted seeds solving the (m,k)–problem. This is interestingly in contrast with lossy framework where optimal seeds usually have an irregular structure. An important practical application of lossless filtration is the selection of reliable oligonucleotide for DNA microarray [22].

2.7.2 Vector Seeds

PatternHunter, which uses spaced seeds improves run-time and sensitivity of homology search over BLAST. Similar strategies were developed by other researchers. In particular, BLAT [20] allows a fixed number of mismatches in the region that makes up a hit. For example, we may require at least 11 matches in a region of length 12. The mismatch may occur at any of the twelve positions. This property could not be expressed by BLAST seed.

Vector seeds, introduced in [5], unify and further generalize the hit definitions used by PatternHunter and BLAT. They can also be applied to protein homology search, where programs traditionally use more complicated hit definitions reflecting the properties of amino acid substitution matrices (score matrices) used to score alignments.

In [5], an ungapped pairwise local alignment is represented as a sequence of real numbers, each corresponding to a position in the alignment. This representation can help in defining a

hit in seed vector model. Such a sequence of positional scores is called an *alignment sequence*. In the simplest case, we represent pairwise alignments as binary sequences like what we introduced in previous sections. Each 1 represents a match while each 0 represents a mismatch. For protein alignments, we represent the alignment between sequences $Y = y_1y_2 \dots y_n$ and $Z = z_1z_2 \dots z_n$ by the sequence of positional scores, $(s_{y_1,z_1}, s_{y_2,z_2}, \dots, s_{y_n,z_n})$, where $S = (s_{i,j})$ is the scoring matrix.

Definition. A vector seed is an ordered pair $Q = (v, T)$, where v is the seed vector $(v_1, v_2, \dots, v_l)$ of real numbers and T is the seed threshold value.

An alignment sequence $X = (x_1, x_2, \dots, x_n)$ hits the seed Q at position p if $\sum_{i=1}^{l}(v_i.x_{p+i-1}) \geq T$. That is, the dot product of the seed vector and the alignment sequence of length l beginning at position p is at least the threshold T .

We now show how to express several examples of hit definition as vector seeds. Vector seeds can generalize the spaced seeds of PatternHunter, the mismatching seeds of BLAT and the minimum word score seeds used by BLASTP.

Expressiveness of vector seeds: Spaced seeds are a special case of vector seeds. To cast them in the vector seed framework we can use binary alignment sequences. To construct a vector seed (v, T) equivalent to a spaced seed Q, we set the weight vector v equal to the spaced seed string, and the threshold will be equal to the weight of the seed Q. For example, the nucleotide BLAST seed 11111 of weight 5 is equivalent to the vector seed ((1, 1, 1, 1, 1), 5) and the spaced seed 1011101 is equivalent to the vector seed ((1, 0, 1, 1, 1, 0, 1), 5). Similarly, the seeding strategy used in BLAT can be formulated as a vector seed over the binary alignment sequence. For example, a BLAT hit definition that requires at least 7 matches in a region of length 9 corresponds to the vector seed ((1, 1, 1, 1, 1, 1, 1, 1, 1), 7). Moreover, the BLASTP rule that a hit is three consecutive positions having total score at least 13 corresponds to the vector seed ((1, 1, 1), 13). However, vector seeds can also encode more complicated concepts. For example, if the alignment sequence is binary, the vector seed ((1, 2, 0, 1, 2, 0, 1, 2), 8) requires matches in all positions with seed vector value of two, while allows one mismatch in the three positions with value one. Those positions with value 0 are non-relevant to a hit. This can not be expressed using spaced seeds or BLAT seeds. Vector seeds are not universally expressive. For example, there is no way in the vector seed model to require that three of the four codons are match in the first two positions each. The seed ((1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1), 6) also allows one mismatch each in two codons.

Now, we discuss how to identify hits in a sequence database. Assume we have two sequences (or sequence databases) and want to find all hits between them. If hits are required to be exact matches of length k, the common approach is to create a hash table of all k-mers in one of the sequences and then search for each k-mer of the other sequence in the table. If hits are not exact matches (such as in BLAT or BLASTP), we can take each k-mer in the second sequence, generate a list of k-mers that would produce a hit and search for these k-mers in the hash table. This approach extends to the vector seed scenario. We need to hash only characters on positions corresponding to non-zero elements in the vector seed. Hence, we seek vector seeds with small support (number of non-zero elements in a vector seed) that allow for a small number of hash table entries to be examined for each position in a query sequence. Otherwise, it would not be practical.

Three models has been investigated in [5]. (i) PatternHunter Bernoulli model with simi-

larity level 0.7. (ii) Three-periodic model of alignments in protein coding regions, where each triplet is emitted as a unit, chosen from a probability distribution over $\{0,1\}^3$. Each triplet is independent of the others in this model. Three-periodicity of an alignment means that some of the positions of a codon are less conserved than others. Such models can be used to effectively model the conservation in coding alignments [4]. (iii) For protein sequences, the alignment is represented as a sequence of BLOSUM62 scores ranging from -4 to 11, a positionally independent model similar to PatternHunters model is used.

Vector seeds offer a wider vocabulary for seed matches than spaced seeds. For example, they allow certain positions to be more important than others. They allow a fixed number of mismatches in some positions and an arbitrary number in others. Extensions of vector seeds improve the sensitivity. Especially, with the coding sequence nucleotide alignments, those alignment programs that use vector seeds have better sensitivity and the same specificity rather than those using spaced seeds.

In [5], an algorithm was proposed for computing sensitivity. This algorithm is an extension to the algorithm of [19] for computing sensitivity. The alphabet has changed and need not be binary, and that the definition of a hit is the more complicated dot product property. In [4], an extension of the original [19] algorithm to the case where the alignment sequence is generated by a hidden Markov model, was proposed. Although the spaced seeds can be helpful for proteins, the improvements are not as dramatic as for nucleotides. This is mainly because of two reasons. First, the BLASTP seed is very short, and thus it is hard to improve it by spacing. Also, spaced seeds consider only matches and mismatches, and not the richer similarity measure introduced by amino acid substitution matrices. On the other hand, vector seeds and their generalizations (like multiple seeds) allowed researchers to achieve improvements compared to BLASTP seed. For example, [7] reports a collection of eight vector seeds that achieve almost the same sensitivity as the BLASTP seed while reducing the number of false positives four to five times. In this approach, a hit is a position where at least one vector seed from the collection has a hit.

2.7.3 Transition Constrained Seeds

Transition-constrained seeds are proposed in [32] as an extension to spaced seeds due to the possibility of distinguishing transition and transversion mismatches. We briefly describe the transition-constrained seed model as discussed in [32]. Its idea is based on the well-known feature of genomic sequences that transition mutations (nucleotide substitutions between purins or between pyrimidins) occur relatively more often than transversions (other substitutions).

In real genomic sequences transitions are twice more frequent than transversions. For example, [32] referred to real genomic sequences that the transition/transversion rate (ti/tv) is greater than one on average. Transitions are much more frequent in coding sequences as most of silent mutations are transitions. Furthermore, ti/tv is often greater for related species, as well as for specific DNA.

Transition-constrained seeds are defined on the ternary alphabet $1, @, 0$, where @ stands for a match or a transition mismatch ($A \leftrightarrow G, C \leftrightarrow T$), and 1 and 0 have the same meaning as for spaced seeds. The weight of a transition- constrained seed is defined as the sum of the number of 1's plus half the number of @'s since a transition carries one bit of information while a match carries two bits.

Transition constraint seeds in [32] were used for Bernoulli alignment model and for Markov alignment model. We will briefly discuss some of their contributions. In order to compute the detection capacity of transition-constrained seeds, Bernoulli alignment model, was used. To apply this model, a gapless alignment was modeled by a Bernoulli sequence over the ternary match/transition/transversion alphabet with the match probability 0.7 (like PatternHunter match probability) and the probabilities of transition/transversion varying according to the ti/tv ratio. The sequence length is set to 64 (like PatternHunter region length). Seed with weights between 9 and 11 are investigated. Transitions and transversions are assumed to occur with equal probability 0.15. The results of [32] claim that transition-constrained seeds have better sensitivity than best spaced seeds of the same weight. However, not only the efficiency of transition-constrained seeds depends on the ti/tv ratio but the weight computation, where each @ counts as 1/2, introduces additional complexities.

Apart from Bernoulli model, transition-constrained seeds for Markov alignment model was investigated. In aligning homologous coding sequences, due to protein coding constraints a distribution of errors was shown. Actually, transitions often occur at the third codon position, as these mutations are almost always silent for the resulting protein. Markov models can be used for homologous coding regions as shown in [4]. Several methods were proposed in [4, 3] to compute the hit probability of spaced seeds with respect to gapless alignments specified by (hidden) Markov models. Experimens for Markov model of order 5 was done by designing the optimal spaced and transition-constrained seeds of weight 9 - 11 with respect to this Markov model. Results show that transition-constrained seeds increase the sensitivity with respect to this Markov model too. Therefore, transition-constrained seeds perform better with respect to both Bernoulli and Markov model.

2.7.4 Subset Seeds

Spaced seeds use the simplest possible binary match-mismatch alignment model that allows an efficient implementation by hashing all occurring combinations of matching positions. Vector seeds are a powerful generalization of spaced seeds. They allow us to use arbitrary alignment alphabet and, on the other hand, provide a flexible definition of a hit based on a cooperative contribution of seed positions. Despite higher expressiveness, vector seeds have more complicated algorithms and direct hashing methods at the seed location stage are impossible.

Subset seeds, [23, 24], have an intermediate expressiveness between spaced and vector seeds. This concept allows an arbitrary alignment alphabet and, on the other hand, still allows using a direct hashing for locating seed, which maps each string to a unique entry of the hash table. A seed automaton for subset seeds was proposed and has been shown that it differs from the Aho-Corasick automaton. Subset seeds generalize spaced seeds based on the idea to distinguish between different types of mismatches in the alignments. This leads to representing both alignments and seeds as words over larger alphabets.

We consider an alignment alphabet A. We assume that A contains the symbol 1, interpreted as match. A subset seed is defined as a word over a seed alphabet B, such that:

- letters of B denote subsets of alphabet A
- B contains a letter # that denotes subset $\{1\}$

- A subset seed $b_1 b_2 \dots b_m \in B_m$ matches an alignment fragment $a_1 a_2 \dots a_m \in A_m$ if $\forall i \in [1 \dots m], a_i \in b_i$

As before, the length of seed π is its length, and the #-weight of seed is the number of # in π. For example, for DNA sequences over the alphabet A, C, G, T, in [32] the alignment alphabet is set to $A = \{1, h, 0\}$ that represents respectively a match, a transition mismatch ($A \leftrightarrow G, C \leftrightarrow T$), or a transversion mismatch (other mismatch). In this case, the appropriate seed alphabet is $B = \{\#, @, -\}$ corresponding respectively to subsets $\{1\}$, $\{1, h\}$, and $\{1, h, 0\}$. Thus, the seed $\pi = \#@-\#$ matches the alignment $A = 10h1h1101$ at positions 4 and 6. The #-weight of π is 2 and its length is 4. Unlike the weight of ordinary spaced seeds, the #-weight cannot serve as a measure of seed selectivity. In the above example, the symbol @ should be assigned weight 0.5, so that the weight of seed is equal to 2.5 and this weight can be considered as the weight of the seed.

A subset seed automaton was proposed in [23] that recognizes the set of all alignments matched by a seed. This automaton has $O(w^{2r})$ states regardless of the size of the alignment alphabet. It has been shown that its transition table can be constructed in time like construction time for spaced seeds with Aho-Corasick automaton but results in a smaller number of states in practice. (The automaton size is smaller). Different experiments confirm the practical efficiency of the subset seed, both at the level of computing sensitivity for designing good seeds, as well as using those seeds for DNA similarity search.

2.8 Applications of Seeds in Software Programs

In spite of many extensions of spaced seeds, multiple spaced seeds are the most widely used, due to their high sensitivity, simplicity, and efficiency.

There are a number of algorithms and associated software programs for read mapping that use multiple spaced seeds. For instance, MAQ [28], SToRM [15], BFAST [16], PerM [11], SHRiMP [33], ZOOM [29] use spaced seeding technique requiring one or several hits per read.

The next generation sequencing technologies are generating billions of short reads daily. Resequencing need fast softwares to map sequencing reads to a reference genome. The analysis of next generation sequencing data requires the mapping of short reads back to a reference genome, allowing a few mismatches and indels.

Seed-based methods for read mapping use different seeding strategies. SHRiMP [33]- the SHort Read Mapping Package is a set of algorithms and methods to map short reads to a genome, even in the presence of a large amount of polymorphism. It uses spaced seeds that can hit at any position of the read and introduces a lower bound on the number of hits within one read. In [33] algorithms were developed for the mapping of short reads to highly polymorphic genomes and also methods were proposed for the analysis of the mappings. An algorithm for mapping short reads is demonstrated in the presence of a large amount of polymorphism. By employing a fast k-mer hashing step and a simple, very efficient implementation of the Smith-Waterman algorithm, the SHRiMP method conducts a full alignment of each read to all areas of the genome that are potentially homologous. Actually, multiple seeds are used to determine if a good match exists where we require a predetermined number of seeds from a read to match

within a window of the genome. MAQ [28] uses six light-weight seeds allowed to hit in the initial part of the read. ZOOM [29] proposes to use a small number (4 – 6) of spaced seeds each applying at a fixed position, to ensure a lossless search with respect to a given number of mismatches. Indeed, the idea of spaced seed has been extended to use different spaced seeds at several designated positions of the read. Thus, a spaced seed becomes the combination of its pattern and the read position where it is applied. For example, a seed 0001110100000000 is the seed 11101 applied at the fourth position of the read with length 16. A spaced seed has the same length as the read. Therefore, it is only used once to index a read. Multiple spaced seeds method was used to design different seeds on different positions of a read. This significantly reduced the number of indexes per read required to achieve 100% sensitivity, resulting less memory consumption and fewer hits. Consequently, the mapping speed is greatly improved.

In the lossless framework, PerM [11] proposes to use periodic seeds to save on the index size. Periodic spaced seeds are used to significantly improve mapping efficiency for large reference genomes. The data structure in PerM requires only 4.5 bytes per base to index the human genome, allowing entire genomes to be loaded to memory, while multiple processors simultaneously map reads to the reference. Weight maximized periodic seeds offer full sensitivity for up to three mismatches and high sensitivity for four and five mismatches while minimizing the number random hits per query, significantly speeding up the running time. BFAST method is based on creating flexible, efficient whole genome indexes to rapidly map reads to candidate alignment locations, with arbitrary multiple independent indexes allowed to achieve robustness against read errors and sequence variants. BFAST uses 10 seeds of weight 22 for reads of 50 bp to be mapped on the human genome.

Chapter 3

Algorithms for Computing Seeds

In this chapter, we will discuss about the hardness of designing multiple spaced seeds and then we describe the existing software programs for computing seeds. Given the fact that spaced seeds are so widely used in applications, one would imagine that many algorithms and software programs for computing seeds have been written. That is not the case due to two main reasons. First, the problem of finding optimal seeds is hard, as explained in the first section on this chapter. Second, finding good heuristic algorithms does not seem to be easy. In fact, there are only two software programs, Mandala [3, 36] and Iedera [23, 24]. We shall describe both in this chapter. Additionally, some authors of software programs using seeds constructed their own seeds and we shall describe those as well.

We should make it clear that we focus on designing multiple spaced seeds only, since single seeds can be computed by exhaustive search; see [12]. The exhaustive approach is computationally infeasible for multiple seeds.

3.1 Hardness of the Problem

Computing optimal multiple spaced seeds was proved to be NP-hard. Indeed, [26] shows that, given k seeds, computing the hit probability under the uniform distribution is NP-hard. The problem of finding even one optimal seed is NP-hard. When the homologous region is uniform, that is, a Bernoulli sequence generated with probability p, there have been many exponential time algorithms to compute the hit probability of a given spaced seed [26, 19, 3, 4]. In [27] it has been shown that computing sensitivity of a spaced seed over a uniform region is NP-hard.

Finding optimal (multiple) spaced seeds is NP-hard but even finding good ones is very difficult. Therefore heuristic algorithms are used to find good sets of seeds. Exhaustive search involves two exponential-time steps:

- There are exponentially many seeds to be evaluated based on their sensitivities and
- Computing the sensitivity of each takes exponential time as well.

Several approaches like [3, 36] tried to alleviate the second exponential problem by approximating the sensitivity. For the former, the number of seeds to be considered has been reduced by various heuristics like [12, 37] but their algorithms were still exponential.

Heuristic algorithms that were proposed were all exponential in theory and slow in practice. The algorithm of [18] is the only one that computes good multiple spaced seeds in polynomial time and it will be the focus of our thesis. Therefore, we do not present this algorithms here, instead, it will be thoroughly discussed in Chapter 4.

3.2 Mandala

Mandala is a software tool for seed design that was introduced in [3] and then improved by [36] where the cost of designing multiple seeds was greatly reduced.

3.2.1 Mandala's Problem

Designing seeds, even for simple probabilistic models of biosequence similarities, is computationally challenging. Mandala's probability model is a Markov model. Here is the problem that was investigated in [3, 36]:

Problem: Let M be a Markov model that generates aligned pairs of biosequences, and let the parameters w (weight), s (maximum seed length), and n (number of seeds) be given. Find a set Π of n seeds, each of weight w and length at most s, that maximizes the probability that at least one seed from Π matches an alignment chosen at random from M.

The measure of goodness for seeds used in [3] is sensitivity which is computationally hard. In this new model, a similarity is modeled by a kth-order Markov process M that gives the probability that the next bit seen will be 1 (i.e., a matching pair of bases) given the values of the previous k bits. The zeroth-order marginal probabilities of M correspond to the similarity's overall degree of conservation, while its higher-order marginals can reflect specific patterns of conservation. To illustrate, similarities in coding sequence often exhibit a pattern of two matches followed by a mismatch or a 110 pattern, corresponding to conservation of the underlying protein with silent mutations at third base positions of codons.

Actually the problem is to find a set of seeds that maximize the detection probability under the model M. An algorithm was given in [3] to compute detection probabilities for sets of seeds in Markov models. This algorithm uses dynamic programming on a finite automaton. Mandala uses the local search method for seed selection described in the following subsection.

3.2.2 Local Search Method

We represent a seed as a set of all match positions $\pi = \{x_1, \ldots, x_w\}$ and let it be the current seed, with all $x_i \leq s$ (s is the maximum seed length). To avoid generating shifted versions of the same seed, we fix $x_1 = 0$. The local neighborhood of π is the set of all seeds π' that differ from π in exactly one of $\{x_2, \ldots, x_w\}$, with the differing position chosen from among the unused set $\{1, \ldots, s-1\} - \pi$.

With this neighborhood definition and the probability calculation discussed in [3] as an evaluation function, a hill climbing with random restart is performed in the seed space to find a near-optimal seed. In order to design a set of simultaneous seeds Π, the neighborhood definition is extended to include all sets Π' in which one seed $\pi'_i \in \Pi'$ differs from the corresponding $\pi_i \in \Pi$ in a single position.

The evaluation algorithm is fast for small lengths but its cost grows exponentially as the length increases for fixed weight. The improvements on the computational cost, proposed in [36], are discussed in the next subsection.

3.2.3 Greedy Covering Algorithm

In [36], a new method is proposed that relies on efficient incremental computation of the probability that an alignment contains a match to a seed π, given that it has already failed to match any of the seeds in a set Π. This approach greedily covers the highest probability alignments of M with seeds that match them. This new method reduce the cost of designing multiple seeds compared to the local search algorithm of [3].

Let M be the probabilistic alignment model. A sample from M is a bit string α of length l, with 1's where the aligned sequences match and 0's where they do not. The event that a seed matches an alignment α is denoted as $E_\pi(\alpha)$ and its complementary event is denoted as $\overline{E_\pi}(\alpha)$. A seed set Π matches α if one of its seeds matches α. Similarly, we denote the event that Π matches α by $E_\Pi(\alpha)$ and its complementary event by $\overline{E_\Pi}(\alpha)$.

The greedy covering algorithm of multi-seed design is as follows. As mentioned, Mandala's local search method starts with a set of seeds Π and finds the best seed set in the neighborhood of Π to perform a hill climbing approach. This procedure is done iteratively until no further local improvement can be done. To speed up this algorithm, a greedy covering heuristic for choosing seed sets was developed in [36]. Given a partial seed set Π_0 of size $n' < n$, (n is the number of seeds), a set Π of size $n' + 1$ is formed by choosing the next seed π to maximize the conditional match probability $Pr(E_\pi|\overline{E_{\Pi_0}})$. Each step of the heuristic attempts to cover the highest-probability alignments not already matched by some seed in the current partial set. Starting from a single locally optimal seed, after $n - 1$ iterations of greedy covering a seed set of size n will be produced. Greedy covering algorithm is faster than Mandala's local search because most of seed set evaluations are performed on partial sets of size $< n$, while local search always evaluates sets of full size n. Also, in this new approach, each covering step optimizes only a single seed.

3.3 Iedera

Iedera is a program to select and design subset seeds. The theoretical concepts behind this program were proposed in [23, 24]. As mentioned in 2.7.4, spaced seeds and transition constrained seeds can be perfectly represented in the subset seed model.

In [23] a general framework for computing sensitivity is proposed. It allows one to compute the seed sensitivity for different definitions of seeds and different alignment models. This approach is based on a finite automata representation of the set of target alignments and the set of alignments matched by a seed, as well as on a representation of the probabilistic model of alignments as a finite-state transducer. The main part of [23] is a finite automaton that recognizes the set of alignments matched by a given subset seed. This automaton is called a subset seed automaton. The efficiency of the whole algorithm depends on the size of this subset seed automaton. Note that [3] also constructs an automaton, based on the Aho-Corasick automaton.

In [23] it was shown that size of this subset seed automaton has $O(w2^{r-w})$ states (w is the number of #'s and r is the number of other symbols). In comparison with Aho-Corasick automaton that has $O(w|A|^{r-w})$ states, the subset seed automaton is more space efficient. This automaton construction is implemented in full generality in the Iedera software package. To design spaced seeds or subset seeds, regarding to the model that is going to be used, a probability transducer is constructed. The seeds are enumerated exhaustively and the sensitivities are computed by the subset seed automaton. The best set will be reported. Consequently, the Iedera software is very slow for designing multiple seeds. It is not competitive already with Mandala. Hence, we will not use it for comparing.

3.4 PatternHunter

The seeds of the PatternHunter II software, [26], were computed using a greedy algorithm. Enumerating all possible sets and evaluating them with the Sensitivity algorithm of Section 1.6 is not feasible due to the exponential number of possible seed sets. Hence, PatternHunter proposes a method to construct good set of seeds greedily. The method is as follows.

It computes the first seed s_1 that maximizes the hit probability of $\{s_1\}$. Then, fixing s_1, the second seed is computed in a way that maximizes the hit probability of $\{s_1, s_2\}$. This procedure is continued until the desired number of seeds or the desired hit probability is reached.

The set that is generated by the above algorithm is not necessarily optimal. All possible seed sets are not investigated since every time a partial set is fixed. As mentioned in [26], it took about 12 CPU days for a Pentium IV 3GHz to compute a set of 16 weight 11 seeds that each length could not be longer than 21.

3.5 BFAST

BFAST is an alignment tool for large scale genome resequencing. The novel contribution of BFAST is the candidate alignment locations (CAL) search step, where a list of CALs is tabulated for each read with the goal to include the true (or correct) location within the CALs. BFAST uses multiple indexes of the reference to increase sensitivity of alignment. An index is defined by a spaced seed (or mask), a string of 0s and 1s that start and end with a 1, that define the bases in the read considered during the lookup in the index. We will briefly explain how BFAST compute seeds (masks) in order to index the reads to proper positions. In this context, we will use the term mask instead of seed as it is used in [16].

For mask design problem, simple random search strategies have been developed. In the simplest strategy, a global random sampling search is performed. The total number of masks in the set, S, and the key size k, are fixed. This strategy allows a search range of mask widths (seed length), extending from $w = k$ to some upper bound. The search is initialized with some given set of masks, which is typically taken to be a single mask, the most compact mask $M = 111\ldots111$ of width $w = k$. The remaining masks $S - 1$ of the set are then sampled at random, as follows: for each the strategy chooses a width, w, at random from the allowed range of widths, and then chooses the mask layout, from the $w - 2$ choose $k - 2$ ways of distributing the k ones of the mask in the w positions, insisting that the first and last position be one. Given a

full mask set, the accuracy table is evaluated. The strategy repeats this global random sampling of mask sets for a specified numbers of samples, and retains the mask set that has the "greatest" accuracy. The result is an optimized S-mask set.

Chapter 4

SpEED

As mentioned in chapter 2, heuristic algorithms for computing seeds were all exponential in theory and slow in practice until the algorithm of [18] was proposed. It computes good multiple spaced seeds in polynomial time. The novelty of this algorithm is due to a completely different approach with respect to the previous methods. It proposes the overlap complexity idea which will be discussed in this chapter. Also in this chapter, after introducing the overlap complexity, we will talk about the limitation of algorithm proposed in [18] that it does not provide seeds lengths for the general case. Therefore, our approach is to propose a heuristic algorithm to find a set of length for a set of seeds that we want to design. In section 4.3, we will propose a modified polynomial time algorithm that generates seeds, which uses the algorithm that finds good seed lengths.

4.1 Overlap Complexity

In this section, we introduce the overlap complexity measure, proposed in [18]. This measure has turned out to be well correlated with sensitivity but it is much easier to compute: it takes polynomial time instead of the exponential time required for sensitivity. Therefore, it can be used instead of sensitivity in computations. Using it, we can have a polynomial time algorithm to compute seeds.

4.1.1 Definition

Spaced seeds are represented as words on alphabet $\{1, *\}$ where 1's represent required matches and *'s represent don't care positions. We use an example to show intuitively why overlapping hits of a seed are undesirable. As shown in Figure 4.1, two seeds are used to detect an alignment. The alignment is detected by one hit of a good seed while a bad seed wastes three hits to detect the same alignment.

Hence, the sensitivity of a seed appears to be inversely proportional with the number of overlapping hits, since the expected number of hits is the same (For a seed of length ℓ and weight w, the number of hits in a region of length N is $(L - \ell + 1)p^w$, so it does not depend on the shape of the seed.) Therefore, good seeds have a low number of overlapping hits. Indeed, [27, 3] show that uniformly spaced seeds, that is, seeds that consist of a string $11\ldots1**\ldots*$

Figure 4.1: An example showing the intuition behind overlap complexity; a local alignment is detected by one hit of a good seed whereas a bad seed "wastes" three hits to detect the same alignment.

repeated a number of time (as the bad seed in Figure 4.1), are likely to be among the least sensitive choices for seeded alignment.

A measure is defined that is independent of similarity level p. For two seeds s_1 and s_2, the number of pairs of 1's aligned together when a copy of s_2 shifted by i positions is aligned against s_1 is denoted by $\sigma[i]$. The shift i takes values from $1 - |s_2|$ to $|s_1| - 1$, where a negative shift means s_2 starts first. Precisely, if we denote

$$t_1 = *^{|s_2|-1} s_1 *^{|s_2|-1}$$
$$t_{2,i} = *^{|s_2|-1+i} s_2 *^{|s_1|-i-1}, \text{ for } 1 - |s_2| \leq i \leq |s_1| - 1$$

then

$$\sigma[i] = card\{1 \leq j \leq |s_1| + 2|s_2| - 2,\ t_1[j] = t_{2,i}[j] = 1\}$$

The overlap complexity for two seeds is defined as

$$OC(s_1, s_2) = \sum_{i=1-|s_2|}^{|s_1|-1} 2^{\sigma[i]} \tag{4.1}$$

To illustrate the overlap complexity of two seeds, consider the following example. For two seeds, $s_1 = 11**1*1$, $s_2 = 1*11$, Figure 4.2 shows s_1 aligned against copies of s_2 shifted by i position where $1 - |s_2| \leq i \leq |s_1| - 1$.

The overlap complexity measure is symmetric, that is $OC(s_1, s_2) = OC(s_2, s_1)$. For a multiple seed $S = \{s_1, s_2, \ldots, s_k\}$ the overlap complexity is defined as

$$OC(S) = \sum_{1 \leq i \leq j \leq k} OC(s_i, s_j) \tag{4.2}$$

In [18] it has been shown that the overlap complexity is, experimentally, very well correlated with sensitivity for single seeds. That means, seeds with low overlap complexity have high sensitivity.

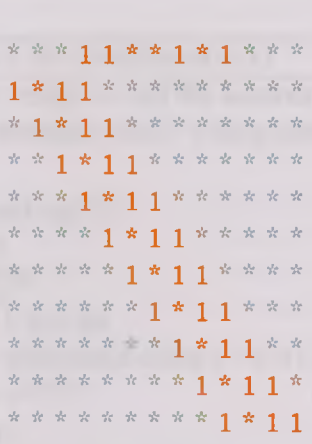

Figure 4.2: An example of overlap complexity of two seeds: $OC(11**1*1, 1*11) = \sum_{i=-3}^{6} 2^{\sigma[i]} = 25$

4.1.2 Polynomial Time Algorithm

Finding optimal seeds by trying all seeds of a given weight (and length) and selecting the best is computationally very expensive. Actually, it has been shown by [26] to be NP-hard for an arbitrary distribution.

In [18], after introducing the overlap complexity measure, an algorithm is presented to compute good seeds. This algorithm does not need to consider exponentially many seeds. It starts from a fixed set of seeds and repeatedly modifies it to improve its overlap complexity. Each improvement consists of swapping a 1 with a * as long as the overlap complexity improves. Moreover, a swap is chosen that produces the greatest improvement. A flip function is necessary to flip a 1 with a * and vice versa: flip(s, i, j) is meant to flip positions i and j in the seed s. For example, flip$(1*11*11, 3, 5) = 1**1111$. Figure 4.3 shows the algorithm MULTIPLESEEDS described in [18]. Initial seeds are seeds of form $*^{l_i-w}1^w$ that are consecutive seeds and have very low sensitivity. By swapping, these seeds are improved. As an example, PatternHunter's seed is obtained by performing only 4 swaps in the algorithm MULTIPLESEEDS(11, 18); see Figure 4.4.

Seeds can be improved dramatically by swapping as it is shown in [18]. This algorithm works fine when the lengths of the seeds are given. However, in practice usually only the weight and number of seeds are known. Solving the problem of finding good length of the seeds is one of the main goals of our research and we shall start discussing it in the next section. In [18] a very simple choice is adopted. The minimum seeds length is chosen as $\lceil \frac{4w}{3} \rceil$ and maximum seed length is 25. Although this choice works well for $w = 11$, $k = 16$ (the PatternHunter II case), it is not necessarily a good choice for other cases.

At the end of this section, we discuss the time complexity of the MULTIPLESEEDS algorithm that is presented in [18]. Steps 1-6 takes $O(kw)$ time. To perform a swap, all possibilities for the triple (r, i, j) in step 9 are considered, that is, $\sum_{i=1}^{k} w(l_i - w)$. For each, we compute the new overlap complexity in $O(l_r \sum_{i=1}^{k} l_i)$ time since the overlap complexity of two seeds is computed in time the product of their lengths and here we need only to update the pairs containing the seed s_r. If an upper bound for seeds length is set to $L = max_{1 \le i \le k} l_i$, then the time complexity of

Algorithm 1 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

1: m =round-up$(\frac{4w}{3})$
2: $M = 25$
3: $h = \frac{2(M-m)}{k}$
4: **for** $i = 1$ to k **do**
5: $\quad l_i \leftarrow \min(\text{round-up}(m + i \times h), M)$
6: $\quad s_i \leftarrow *^{l_i - w}1^w$
7: **end for**
8: $S \leftarrow \{s_1, s_2, ..., s_k\}$
9: swaps $\leftarrow 0$
10: **while** $((\exists\ r, i, j$ with $OC(\{s_1, ..., s_{r_1},$ flip $(s_r, i, j), s_{r+1}, ..., s_k\}) <$ $OC(S))$ **and** (swaps $\leq k \times w$)) **do**
11: $\quad$ choose a triple (r, i, j) that reduces $OC(S)$ the most
12: $\quad S \leftarrow \{s_1, ..., s_{r_1},$ flip $(s_r, i, j), s_{r+1}, ..., s_k\}$
13: $\quad$ swaps $\leftarrow$ swaps $+1$
14: **end while**
15: **return** (S)

Figure 4.3: The MULTIPLESEEDS algorithm

the MULTIPLESEEDS algorithm is $O(k^3L^2w^2(L-w))$.

intermediate seeds	pairs swapped
* * * * * * * 1 1 1 1 1 1 1 1 1 1 1	(1, 12)
1 * * * * * * 1 1 1 1 * 1 1 1 1 1 1	(3, 15)
1 * 1 * * * * 1 1 1 1 * 1 1 * 1 1 1	(2, 9)
1 1 1 * * * * 1 * 1 1 * 1 1 * 1 1 1	(5, 11)
1 1 1 * 1 * * 1 * 1 * * 1 1 * 1 1 1	

Figure 4.4: Intermediate seeds computed by MULTIPLESEEDS(11, 18) to find PatternHunter's seed 111*1**1*1**11*111. The flipped positions are given in the right column..

4.2 Finding Good Lengths

As we mentioned in the previous sections, the MULTIPLESEEDS algorithm works only for fixed lengths of seed. This algorithm requires the lengths of the seeds to be given as the overlap complexity does not allow comparison of multiple seeds of different lengths. The procedure for finding these length in [18] is very limited. Comprehensive testing is needed to determine

good seed lengths given the number of seeds, their weight, the length of the hit region, and the similarity level. We give in this section heuristic algorithms to find good lengths for seeds.

4.2.1 Reducing the Number of Lengths to be Guessed

Some of the software programs based on spaced seeds use very large sets of seeds (BFAST uses 10 and PatternHunter II uses 16). Therefore the number of possibilities for these lengths is very large and it makes it infeasible to do some relevant testing. Our first step is to reduce this search space. To start with, we shall try to infer all the seed lengths from the minimum and maximum of the lengths. This is done by the MAKELENGTHS algorithm, presented in Figure 4.5.

Algorithm 2 MAKELENGTH(min, max, k)

- given: min minimum seed length, max which is maximum seed length and k is the number of seeds
- returns: a set of length l for k seeds

1: $temp \leftarrow max$, $l[0] \leftarrow min$, $l[k-1] \leftarrow max$, $cnt[l[0]] \leftarrow 1, cnt[l[k-1]] \leftarrow 1$
2: isReachedToEnd $\leftarrow$ **false**
3: **for** $i = 1$ to $k-1$ **do**
4: **if** isReachedToEnd = **false then**
5: $l[i] \leftarrow \lceil (l[i-1] + max)/2 \rceil$
6: **else**
7: $l[i] \leftarrow temp$
8: **end if**
9: **if** $cnt[l[i]] < \lceil \frac{k}{2^{max-l[i]+1}} \rceil$ **then**
10: $cnt[l[i]] \leftarrow cnt[l[i]] + 1$
11: **else if** $cnt[l[i]] = \lceil \frac{k}{2^{max-l[i]+1}} \rceil$ **then**
12: isReachedToEnd $\leftarrow$ **true**
13: $l[i] \leftarrow l[i] - 1$
14: $temp \leftarrow l[i]$
15: $cnt[l[i]] \leftarrow cnt[l[i]] + 1$
16: **else**
17: isReachedToEnd $\leftarrow$ **true**
18: $cnt[l[i]] \leftarrow cnt[l[i]] + 1$
19: **end if**
20: **end for**
21: Sort(l) { sort l array so that we have k elements in increasing order}
22: **return** (l)

Figure 4.5: The MAKELENGTH algorithm

A few clarifications are in order. The algorithm generates first one length with minimum length and one with maximum length. Then it continues by choosing the third length in the middle of that interval, the fourth in the middle between the third and the maximum, the fifth in the middle between the fourth and the maximum, etc. If there are many seed lengths to be

generated and the [*min..max*] interval is not very large, the above procedure may eventually generate many seeds of length equal to *max*. To prevent this, we store the number of seeds of each length in the *cnt* array and, if the number of seeds of length equal to *max* becomes half of the total number of seeds, then we stop using the *max* length and move on to $max-1$. Here we shall generate at most a quarter of the total number of seeds and, if that is reached, we move to $max-2$, and so on. The basis for these upper bounds for the number of seeds of a given length is that the number of seeds of length ℓ and weight w is $\binom{\ell}{w}$. Therefore, if ℓ is roughly twice as large as w, there will be approximately twice as many seeds of length $\ell+1$ than of length ℓ.

For example, let $min = 12$, $max = 20$, and $k = 10$. To generate 10 lengths between 12 and 20, using the algorithm we first pick 12 and 20 as the first lengths. Then we find the floor of average between them which is $\lceil \frac{12+20}{2} \rceil = 16$. Hence, the next length will be 16. Similarly, the next length will be $\lceil \frac{16+20}{2} \rceil = 18$ and the fifth length is $\lceil \frac{18+20}{2} \rceil = 19$. After that length $\lceil \frac{19+20}{2} \rceil = 20$ is picked. In this case, we can only have up to $\lceil \frac{k}{2^1} \rceil = 5$ lengths equal to 20. Therefore, we will have these lengths: 12, 16, 18, 19, 20, 20, 20 ,20 , 20. Now, we have one more length to generate. We have to go back, to one length smaller which is 19. We can have up to $\lceil \frac{k}{2^2} \rceil = 3$ lengths equal to 19 but we do not need more since we have reached to 10 lengths that are 12, 16, 18, 19, 19, 20, 20, 20 ,20 , 20.

Using this algorithm, we can modify MULTIPLESEEDS algorithm in a way that it uses the length set that MAKELENGTH generated. Figure 4.6 shows the modified algorithm that is called COMPUTESEEDSWITHMINMAX.

Algorithm 3 COMPUTESEEDSWITHMINMAX(*min*, *max*, *w*, *k*)

- given: the *min* minimum length, *max* maximum length, weight *w* and the number of seeds *k*
- returns: a multiple seed S with k seeds of weight w and high sensitivity

```
1: MAKELENGTH(min, max)
2: for i = 1 to k do
3:     s_i ← *^{l_i - w} 1^w
4: end for
5: S ← {s_1, s_2, ..., s_k}
6: swaps ← 0
7: while ((∃ r, i, j with OC({s_1, ..., s_{r_1}, flip (s_r, i, j), s_{r+1}, ..., s_k}) <
   OC(S)) and (swaps ≤ k × w)) do
8:     choose a triple (r, i, j) that reduces OC(S) the most
9:     S ← {s_1, ..., s_{r_1}, flip (s_r, i, j), s_{r+1}, ..., s_k}
10:    swaps ← swaps +1
11: end while
12: return (S)
```

Figure 4.6: The COMPUTESEEDSWITHMINMAX algorithm

Once we have proper minimum and maximum length we can generate lengths between them using the algorithm presented. Now, the question is how do we find proper minimum and

maximum lengths and it will be answered in the next section.

4.2.2 Finding Proper Min and Max

In this section, we will propose an algorithm to compute the appropriate minimum and maximum length for the MAKELENGTH algorithm. The proposed algorithm, MINMAX, is given in Figure 4.7.

The goal of this algorithm is to find a pair of values (min, max) which, preferably, maximizes the sensitivity of the seeds obtained by the COMPUTESEEDSWITHMINMAX algorithm. Such values will be computed for selected input parameters and then interpolated in the next section to produce good guesses for all cases. It should be noted therefore that the MINMAX algorithm is used only as preprocessing for our main seed computing algorithm to be presented later.

Therefore, the complexity of this algorithm is not essential however, as this algorithm involves repeated computation of sensitivity, which is exponential, we have to build it carefully so that it is feasible. One unfeasible choice is to try all possible combinations of *min* and *max* values. We shall therefore starts with $min = w+2, max = w*2$, which are selected heuristically. We shall search then for values (min, max) so that the sensitivity resulting from this choice is, if not the global maximum, then at least a local one. Under the assumption that it is true, a binary search becomes the obvious choice however, computing sensitivity requires exponential time and space and therefore we cannot simply jump to trying very long seeds. Therefore, we shall advance carefully, by increasing the current *max* only by one unit (assuming the sensitivity increases that way) until the sensitivity stops increasing. After that, we start adjusting the *min* (one unit at the time) until we find its local maximum. We then switch back and forth between adjusting *max* and *min* until the local maximum is found, that is, increasing or decreasing either *min* or *max* by one unit would cause sensitivity to decrease.

The algorithm is divided into several procedures to increase readability. The procedure IDENTIFYMOVE tells identifies the direction (increase or decrease) in which we need to adjust *min* or *max* (depending which one we currently considering) in order to increase the sensitivity. The procedure SINGLEMOVE adjusts the *min* or *max* according to the moving direction indicated by IDENTIFYMOVE.

Note also that we always check that *min* and *max* do not go below w or above N, respectively. These checks were omitted from the pseudocode for clarity.

4.3 The Engineered Algorithm

4.3.1 Preprocessing and Analysis of Data

Using the MINMAX algorithm, we did comprehensive testing to find proper (min, max) pairs for a wide range of of the parameters k, w, N, p, using especially values that are most common in practice. Our testing was perfomed on Sharcnet[1] and the (min, max) pairs were computed for the following parameters:

- $k = 2, w = 10, 12, 14, \ldots, 22, N = 35, 50, 75, 100, p = 0.9$

[1] www.sharcnet.ca/

Algorithm 4 MINMAX(S, w, k, p, N)

- given: the set of seeds S, weight w, the number of seeds k and similarity p, region length N
- returns: best (min, max) pair

```
1:  min ← w + 2
2:  max ← 2w
3:  sen ← SENSITIVITY(COMPUTESEEDSWITHMINMAX(min, max), p, N)
4:  repeat
5:    repeat
6:      newMaxFound ← 0
7:      senDecBy1 ← SENSITIVITY(COMPUTESEEDSWITHMINMAX(min, max − 1), p, N)
8:      senIncBy1 ← SENSITIVITY(COMPUTESEEDSWITHMINMAX(min, max + 1), p, N)
9:      move ← IDENTIFYMOVE(sen, senIncBy1, senDecBy1)
10:     newMaxFound ← max(newMaxFound, move)
        SINGLEMOVE(move, max, sen, senIncBy1 , senDecBy1)
11:   until (move = 0)
12:   repeat
13:     newMinFound ← 0
14:     senDecBy1 ← SENSITIVITY(COMPUTESEEDSWITHMINMAX(min − 1, max), p, N)
15:     senIncBy1 ← SENSITIVITY(COMPUTESEEDSWITHMINMAX(min + 1, max), p, N)
16:     move ← IDENTIFYMOVE(sen, senIncBy1, senDecBy1)
17:     newMinFound ← max(newMinFound, move)
        SINGLEMOVE(move, min, sen, senIncBy1 , senDecBy1)
18:   until (move = 0)
19: until ((newMaxFound = 0) and (newMinFound = 0))
20: return (min, max)
```

Figure 4.7: The MINMAX algorithm

- $k = 3, w = 10, 12, 14, \ldots, 22, N = 35, 50, 75, 100, p = 0.9$
- $k = 4, w = 10, 12, 14, \ldots, 22, N = 35, 50, 75, 100, p = 0.9$
- $k = 10, w = 10, 12, 14, \ldots, 22, N = 35, 50, 75, 100, p = 0.9$

Therefore, 112 pairs of (min, max) are computed. Through this testing, our goal was to give a good approximation for (min, max) pairs. For each k, we did two interpolations one for min and one for max. Therefore, eight interpolations are performed.

To have a polynomial interpolation one method is to fit a single polynomial through all data points. In this case, for N points, one needs to fit a polynomial of degree $N - 1$. Problems arise when there are many data points, as then higher order polynomials are needed. However, higher order polynomials may exhibit oscillations, and thus may be a poor approximation of the function. Another method is to fit piecewise polynomials (splines) through the data points. This is a better choice when there are many points. All (piecewise) polynomials are of the same degree, and the best choice in 1 dimension is the cubic spline (degree 3 piecewise polynomials).

Algorithm 5 IDENTIFYMOVE(sen, senInc , senDec)

- given: sensitivity, increased sensitivity, decreased sensitivity
- returns: move number

```
1: move ← 0
2: maxSen ← maximum(senDec, senInc)
3: if maxSen = senDec and maxSen > sen then
4:     move ← 1
5: else if maxSen = senInc and maxSen > sen then
6:     move ← 2
7: else
8:     move ← 0
9: end if
10: return move
```

Figure 4.8: The IDENTIFYMOVE algorithm

Algorithm 6 SINGLEMOVE(move, m, sen, senIncBy1 , senDecBy1)

- given: move number, m can be minimum seed length or maximum seed length, sensitivity, increased sensitivity, decreased sensitivity

```
1: if move = 1 then
2:     sen ← senDecBy1
3:     m ← m − 1
4: end if
5: if move = 2 then
6:     sen ← senIncBy1
7:     m ← m + 1
8: end if
```

Figure 4.9: The SINGLEMOVE algorithm

For our problem, since we have exact values of *min* and *max*, interpolation is a better choice than least-squares approximation. Also, since we have many data points, splines are preferred to single polynomial interpolation. We used cubic spline interpolation. The MATLAB function is "interp2" with the option of the method "spline" (for cubic spline). We plotted the approximating function in 2 dimensions in MATLAB with "surf" (for 3-D shaded surface plot). The plots obtained from our analysis are shown in Figures 4.10, 4.11, 4.12, and 4.13.

Based on the analysis we did on the mentioned points we can infer *min* or *max* knowing k, w and N. The plots contain the values as obtained from the spline, which are usually not integers. We rounded the values to the nearest integers and put them in arrays. In other words, we compute an array for each plot. Each array is a 2 dimensional array with 13 rows and 66 columns that can cover $w \in \{10, 11, \ldots, 22\}$ and $N \in \{35, 36, \ldots, 100\}$. As an example, for computing good *min* length, seedLength_min_k4[13][66] array is computed. If we want to use that in our

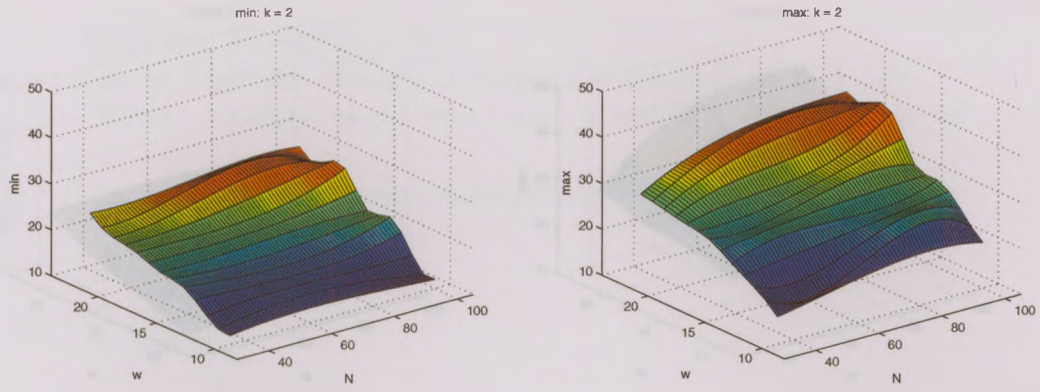

Figure 4.10: The *min* and *max* values for $k = 2$

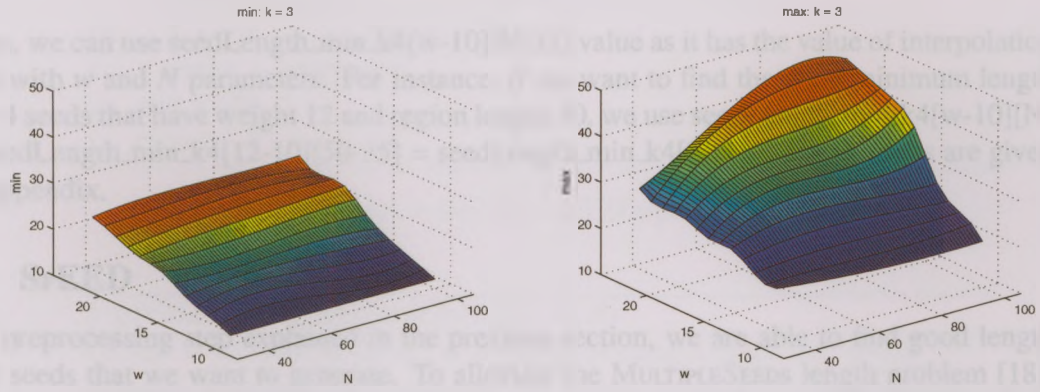

Figure 4.11: The *min* and *max* values for $k = 3$

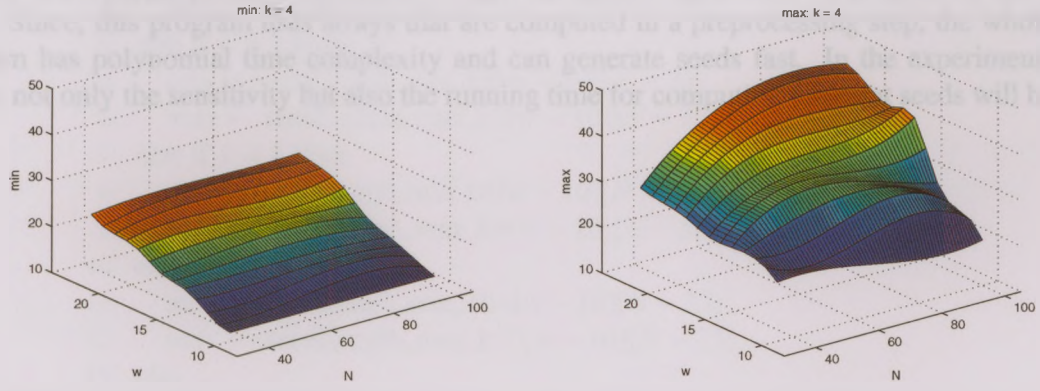

Figure 4.12: The *min* and *max* values for $k = 4$

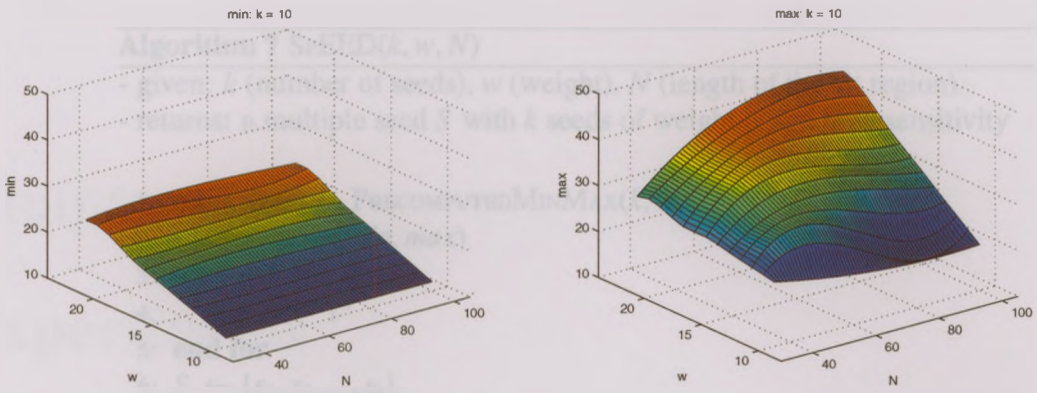

Figure 4.13: The *min* and *max* values for $k = 10$

program, we can use seedLength_min_k4[w-10][N-35] value as it has the value of interpolation for *min* with w and N parameters. For instance, if we want to find the good minimum length for $k = 4$ seeds that have weight 12 and region length 50, we use seedLength_min_k4[w-10][N-35]= seedLength_min_k4[12-10][50-35] = seedLength_min_k4[2][15]. These arrays are given in the appendix.

4.3.2 SPEED

By the preprocessing step explained in the previous section, we are able to find good length sets for seeds that we want to generate. To alleviate the MULTIPLESEEDS length problem [18], we use our arrays to find good lengths and then apply overlap complexity measure. Therefore, we propose a new algorithm that is named SPEED and is shown in Figure 4.14.

Actually, since most software programs are using 2, 3, 4 or 10 seeds, and the usual ranges for w and N are $\{10, 11, \ldots, 22\}$ and $\{35, 36, \ldots, 100\}$ respectively, we did our analysis for these values. Since, this program uses arrays that are computed in a preprocessing step, the whole algorithm has polynomial time complexity and can generate seeds fast. In the experiments chapter, not only the sensitivity but also the running time for computing different seeds will be given.

Algorithm 7 SpEED(k, w, N)

- given: k (number of seeds), w (weight), N (length of the hit region)
- returns: a multiple seed S with k seeds of weight w and high sensitivity

1: $(min, max) \leftarrow$ PrecomputedMinMax(k, w, N)
2: MakeLength(min, max)
3: **for** $i = 1$ to k **do**
4: $\quad s_i \leftarrow *^{l_i - w}1^w$
5: **end for**
6: $S \leftarrow \{s_1, s_2, ..., s_k\}$
7: swaps $\leftarrow 0$
8: **while** $((\exists \quad r, i, j$ with $OC(\{s_1, ..., s_{r_1},$ flip $(s_r, i, j), s_{r+1}, ..., s_k\}) <$ $OC(S))$ **and** (swaps $\leq k \times w$)) **do**
9: $\quad$ choose a triple (r, i, j) that reduces $OC(S)$ the most
10: $\quad S \leftarrow \{s_1, ..., s_{r_1},$ flip $(s_r, i, j), s_{r+1}, ..., s_k\}$
11: $\quad$ swaps $\leftarrow$ swaps $+1$
12: **end while**
13: **return** (S)

Figure 4.14: The SpEED algorithm

Algorithm 8 PrecomputedMinMax(k, w, N)

- given: k (number of seeds), w (weight), N (length of the hit region)
- returns: (min, max) as computed above or $(w + 2, 2w)$ if outside range

1: **if** $k = 2$ **then**
2: $\quad min \leftarrow$ seedLength_min_k2$[w - 10][N - 35]$
3: $\quad max \leftarrow$ seedLength_max_k2$[w - 10][N - 35]$
4: **else if** $k = 3$ **then**
5: $\quad min \leftarrow$ seedLength_min_k3$[w - 10][N - 35]$
6: $\quad max \leftarrow$ seedLength_max_k3$[w - 10][N - 35]$
7: **else if** $k = 4$ **then**
8: $\quad min \leftarrow$ seedLength_min_k4$[w - 10][N - 35]$
9: $\quad max \leftarrow$ seedLength_max_k4$[w - 10][N - 35]$
10: **else if** $k = 10$ **then**
11: $\quad min \leftarrow$ seedLength_min_104$[w - 10][N - 35]$
12: $\quad max \leftarrow$ seedLength_max_k10$[w - 10][N - 35]$
13: **else**
14: $\quad min \leftarrow w + 2, max \leftarrow 2w$
15: **end if**
16: **return** (min, max)

Figure 4.15: The PrecomputedMinMax function

Chapter 5

Experiments

This chapter includes our experiments. Our goal is to compute seeds with the SpEED algorithm that takes as input parameters k, w, N and p in order to improve the seeds that have computed or used by different software programs. Since Iedera software is not competitive for computing multiple spaced seeds, we compared our algorithm with Mandala based on two measures, sensitivity and runtime. Both are of crucial importance for algorithms that compute seeds. We tried to improve existing seeds that are used for instance in read mapping programs by computing seeds with the same parameters k, w, N and p. To show the efficiency of our algorithm, we computed seeds with those parameters also by Mandala software. Our results show that our seeds are always more sensitive than Mandala's and are computed much faster. Besides, our seeds improve the original seeds. Therefore, these seeds can be used to improve the programs that use them.

Mandala suffers from a deficiency of not producing length sets. The maximum seed length should be given to Mandala by the user. Therefore, we used the original seeds maximum length. Indeed, our work deals with this problem as mentioned in the previous chapter. Hence, one can compute seeds with our program much easier than Mandala since Mandala needs maximum seeds length and it should be tested for different values to see what would be the best maximum length (for each test case). Also, since Mandala takes exponential time to run, this testing would be very time-consuming or maybe infeasible altogether. Although for the sake of comparison, we used the original seeds maximum length for Mandala, our program is superior in this aspect due to its ability to compute good seed lengths.

In this chapter, we will also compute seeds to improve those used by various programs, namely, PatternHunter II, BFAST, SHRiMP, PerM and SToRM. We run all the programs on Sharcnet[1].

5.1 PatternHunter II

The PatternHunter parameters are $k = 16, w = 11, N = 64$ and $p = 0.7$. We computed seeds for these parameters but with different similarity levels both with our program and with Mandala to compare our seeds with Mandala's. For our program, since PatternHunter has 16 seeds, we computed the proper (min, max) pair by the MinMax algorithm proposed in the previous

[1] www.sharcnet.ca/

chapter. Actually, we did not compute *min* and *max* for 16 seeds of different weight and region length because PatternHunter is a special case and usually softwares do not use 16 seeds. Therefore, we only computed the proper (min, max) for $k = 16, w = 11, N = 64$ parameters to be used in our experiments. We used maximum seed length of 21 as it is the PatternHunter's maximum seed length. Table 5.1 includes our results. It consists of the sensitivities of PatternHunter seeds, Mandala's seeds and our seeds (that we call it SpEED seeds) for different similarity levels. It also contains the comparison of runtime of Mandala's program and our program for different similarity levels.

	sensitivity			time (sec)	
similarity	PatternHunter	Mandala	SpEED	Mandala	SpEED
70%	0.924114	0.922232	0.929587	4038.81	20.456
75%	0.984289	0.983969	0.985904	3138.23	20.484
80%	0.998449	0.998387	0.998663	7063.78	20.472
85%	0.999951	0.999949	0.999960	2748.51	20.464
90%	1.000000	1.000000	1.000000	4299.92	20.46
95%	1.000000	1.000000	1.000000	1873.16	20.468

Table 5.1: The sensitivity of our multiple seeds compared to that of PatternHunter and Mandala. Last two columns show the running time of Mandala and our program SpEED. The seed parameters that programs use are $k = 16, w = 11, N = 64$.

From Table 5.1 we can understand that our program computes seeds that are more sensitive than PatternHunter's seeds while Mandala does not. Our seeds are more sensitive than Mandala's seeds and it takes only about 21 seconds to compute these seeds while Mandala has different runtime for different similarity levels. Mandala's runtime is on the average more than an hour. This runtime will be much longer if Mandala's maximum length is set to a little larger value since Mandala has exponential time complexity while our program has polynomial time complexity. This is actually quite inconvenient. If we set maximum length to 100, Mandala tries to compute seeds with that length. In summary, we can easily conclude that our program is more efficient in computing seeds than Mandala and it also improves PatternHunter's seeds.

5.2 BFAST

BFAST parameters are $k = 10, w = 22, N = 50$ and $p = 0.95$ ([16]). Similar to the experiment discussed in Section 5.1, an experiment is done for BFAST seeds. We computed seeds for BFAST parameters k, w and N for different similarity levels both with our program and with Mandala to compare our seeds with Mandala's. Maximum seed length of 40 is used for Mandala as it is the BFAST's maximum seed length. Table 5.2 includes the results. It consists of the sensitivities of BFAST seeds, Mandala's seeds and SpEED seeds for different similarity levels. Moreover, this table contains the comparison of runtime of Mandala's program and our program for different similarity levels.

Table 5.2 shows the speed advantage of SpEED program over Mandala software. As it can be seen in the table, Mandala program did not produce any results after a day while our program

	sensitivity			time (sec)	
similarity	BFAST	Mandala	SpEED	Mandala	SpEED
80%	0.293775	-	0.30598	> 1 day	21.284
85%	0.586907	-	0.60340	> 1 day	21.292
90%	0.873359	-	0.88377	> 1 day	21.296
95%	0.992249	-	0.99353	> 1 day	21.296

Table 5.2: The sensitivity of our multiple seeds compared to that of BFAST and Mandala. Last two columns show the running time of Mandala and SpEED. The seed parameters that programs use are $k = 10, w = 22, N = 50$.

takes only about 21 seconds to run for each similarity level. This is due to the exponential time complexity of Mandala. Hence, Mandala program can be too time consuming for some cases like this. In such cases, Mandala is not a good tool to compute seeds.

Further to the time issue, our seeds are more sensitive than BFAST seeds for all similarity levels. For some similarities, like 80%, 85% and 90% our improvement is over a 1% which is a notable increase.

5.3 SHRiMP

SHRiMP is a software package for aligning genomic reads against a target genome. It uses spaced seeds to rapidly and accurately identify candidate mapping locations for each read. Different sets of spaced seeds are used in SHRiMP. However, it has a 4 spaced seeds of weight 12 as default.

SHRiMP parameters are $k = 4$, $w \in \{10, 11, 12, 16, 18\}$, $N \in \{35, 50\}$ and $p = 0.95$. Therefore, in order to improve each set, we computed seeds for (k, w, N) of each SHRiMP seeds for different similarity levels. We did this for Mandala too to compare our seeds with Mandala's as we did for other softwares. We will consider each set here and give the results. For each pair (w, N), we give the results in a separate table.

5.3.1 SHRiMP - weight 10

This section contains comparison of SHRiMP seeds of weight 10 and region length {35, 50} with seeds computed by our program and Mandala for those parameters. We computed seeds for both cases with our program and Mandala and compare the sensitivities and runtime for different similarity levels. Maximum seed length of 21 is used for Mandala as it is the SHRiMP's maximum seed length for weight 10. Therefore, we give two tables and compare the results of $N = 35$ in Table 5.3 and $N = 50$ in Table 5.4.

From these two tables, we can conclude the time efficiency of our method as well as its high sensitivity. Our method improves SHRiMP seeds, in all cases. In some cases the improvement are more obvious. As an exemplification, for similarity level 80% the difference is more significant. In comparison with Mandala, our seeds are always more sensitive than Mandala's. Indeed, for SHRiMP, Mandala can also improve its seeds. Hence, it works well for SHRiMP

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.896379	0.90318	0.90534	53.504	0.076
85%	0.972424	0.97436	0.97643	35.86	0.076
90%	0.996749	0.99704	0.99752	23.28	0.076
95%	0.999942	0.99995	0.99997	58.908	0.072

Table 5.3: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 10, N = 35$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.973159	0.97740	0.97766	17.464	0.076
85%	0.996613	0.99709	0.99745	28.436	0.076
90%	0.99988	0.99991	0.99992	38.348	0.076
95%	1.000000	1.00000	1.00000	29.876	0.076

Table 5.4: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 10, N = 50$.

parameters. Besides, we can see that the longer the region length, the greater the computed sensitivities.

5.3.2 SHRiMP - weight 11

Like what we did in the previous section, we perform our experiments for the SHRiMP seeds of weight 11 and region length {35, 50}. The maximum seed length of 23 is used for Mandala as it is the SHRiMP's maximum seed length for weight 11. Therefore, we will give Table 5.5 for $N = 35$ and Table 5.6 for $N = 50$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.825898	0.83608	0.83656	45.324	0.104
85%	0.944193	0.94843	0.95033	119.724	0.104
90%	0.991537	0.99199	0.99317	188.796	0.1
95%	0.999773	0.99985	0.99985	198.016	0.104

Table 5.5: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 11, N = 35$.

The results given in Tables 5.5 and 5.6 confirm that both our program and Mandala can improve the original SHRiMP seeds of weight 11 for $N = 35$ and $N = 50$. However, our program is more time efficient. It is also more sensitive in all of the cases. Also, comparing the

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.941141	0.94537	0.94824	72.856	0.092
85%	0.990145	0.99116	0.99198	32.932	0.088
90%	0.999484	0.99955	0.99963	125.176	0.092
95%	0.999998	1.00000	1.00000	44.764	0.092

Table 5.6: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 11, N = 50$.

results of these two tables with previous two tables show that a larger weight leads to smaller sensitivity.

5.3.3 SHRiMP - weight 12

The maximum seed length of 25 is used for Mandala as it is the SHRiMP-default's maximum seed length for weight 12. The results are shown in Table 5.7 for $N = 35$ and Table 5.8 for $N = 50$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.742772	0.74861	0.75395	161.124	0.12
85%	0.904162	0.90606	0.91171	210.844	0.12
90%	0.982291	0.98382	0.98477	66.608	0.12
95%	0.999365	0.99929	0.99953	64.532	0.12

Table 5.7: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 12, N = 35$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.893037	0.90089	0.90383	347.608	0.12
85%	0.977253	0.97972	0.98102	785.784	0.12
90%	0.99833	0.99862	0.99877	613.776	0.12
95%	0.99999	1.00000	1.00000	191.416	0.12

Table 5.8: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 12, N = 50$.

5.3.4 SHRiMP - weight 16

The maximum seed length of 30 is used for Mandala as it is the SHRiMP's maximum seed length for weight 16. The results are shown in Table 5.9 for $N = 35$ and Table 5.10 for $N = 50$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.391545	0.39304	0.40180	137.104	0.144
85%	0.646024	0.64794	0.65489	202.36	0.144
90%	0.877337	0.87649	0.88143	519.456	0.14
95%	0.987896	0.98677	0.98840	323.828	0.144

Table 5.9: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 16, N = 35$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.601401	0.59989	0.60546	1167.8	0.536
85%	0.840995	0.84309	0.84568	1254.73	0.532
90%	0.971676	0.97011	0.97355	2282.13	0.528
95%	0.99926	0.99914	0.99936	2822.96	0.532

Table 5.10: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 16, N = 50$.

5.3.5 SHRiMP - weight 18

The maximum seed length of 29 is used for Mandala as it is the SHRiMP's maximum seed length for weight 18. The results are shown in Table 5.11 for $N = 35$ and Table 5.12 for $N = 50$.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SpEED	Mandala	SpEED
80%	0.259008	0.259345	0.26146	26.892	0.256
85%	0.499013	0.498443	0.503639	39.08	0.256
90%	0.780301	0.773827	0.785239	50.804	0.256
95%	0.967227	0.965637	0.969015	67.264	0.256

Table 5.11: Comparing SpEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 18, N = 35$.

In all SHRiMP cases, our program performs better than Mandala with respect to both sensitivity and, especially, time. Also, the original seeds are improved. Our program can improve the original SHRiMP set as before.

	sensitivity			time (sec)	
similarity	SHRiMP	Mandala	SPEED	Mandala	SPEED
80%	0.433537	0.438502	0.444024	320.468	0.82
85%	0.711961	0.720055	0.728247	294.34	0.82
90%	0.925652	0.92504	0.935701	233.752	0.82
95%	0.996299	0.996634	0.997376	195.1	0.82

Table 5.12: Comparing SPEED with Mandala and SHRiMP for SHRiMP parameters of $k = 4, w = 18, N = 50$.

5.4 PerM

PerM is a short-read sequence alignment software that indexes the genome with periodic spaced seeds. We tried to improve PerM-F3-S20 seed family used in PerM software. PerM Parameters are $k = 2, w = 12, p = 0.95$ and $N \in \{35, 50\}$. Similar to the experiments discussed in previous sections , we computed seeds (for both cases of N) with our program and Mandala and compare the sensitivities and run time for different similarities. Maximum seed length of 24 is used for Mandala as it is the PerM's maximum seed length. Tables 5.13, 5.14 include our results for $N = 35$ and $N = 50$ respectively.

	sensitivity			time (sec)	
similarity	PerM	Mandala	SPEED	Mandala	SPEED
80%	0.542551	0.642179	0.643396	12.644	0.008
85%	0.759705	0.839183	0.84001	5.928	0.008
90%	0.924188	0.957713	0.960723	9.58	0.008
95%	0.992938	0.997255	0.997669	18.028	0.008

Table 5.13: Comparing SPEED with Mandala and PerM for PerM parameters of $k = 2, w = 12, N = 35$.

	sensitivity			time (sec)	
similarity	PerM	Mandala	SPEED	Mandala	SPEED
80%	0.748822	0.814538	0.818663	14.02	0.004
85%	0.910681	0.949099	0.950202	12.5	0.008
90%	0.984731	0.994331	0.994753	14.632	0.008
95%	0.999496	0.999928	0.999943	7.44	0.008

Table 5.14: Comparing SPEED with Mandala and PerM for PerM parameters of $k = 2, w = 12, N = 50$.

Tables 5.13 and 5.14 show that both our program and Mandala dramatically improve original PerM seed. This is mainly because PerM seeds are periodic and have low sensitivities.

Again our program is more time efficient than Mandala and generates better and more sensitive seeds.

5.5 SToRM

SToRM is a read mapping tool that uses a three weight 12 seeds. We tried to improve 3-Lossy-12 seed family. SToRM parameter are $k = 3, w = 12, p = 0.95$ and $N \in \{35, 50\}$. Similar to the previous sections , we computed seeds (for both cases of N) with our program and Mandala and compare the sensitivities and run time for different similarities. Maximum seed length of 19 is used for Mandala as it is the SToRM's maximum seed length. Our results for $N = 35$ and $N = 50$ are shown in Tables 5.15, 5.16 respectively.

	sensitivity			time (sec)	
similarity	SToRM	Mandala	SpEED	Mandala	SpEED
80%	0.700365	0.706986	0.713519	1.064	0.04
85%	0.876455	0.87205	0.887065	1.36	0.036
90%	0.972965	0.971455	0.977287	1.276	0.036
95%	0.998644	0.998396	0.999051	0.852	0.04

Table 5.15: Comparing SpEED with Mandala and SToRM for SToRM parameters of $k = 3, w = 12, N = 35$.

	sensitivity			time (sec)	
similarity	SToRM	Mandala	SpEED	Mandala	SpEED
80%	0.860248	0.862456	0.874631	1.42	0.052
85%	0.966088	0.966836	0.971637	1.24	0.052
90%	0.997003	0.997198	0.997745	1.452	0.052
95%	0.999976	0.999979	0.999985	1.94	0.052

Table 5.16: Comparing SpEED with Mandala and SToRM for SToRM parameters of $k = 3, w = 12, N = 50$.

The results of this section again show that our program notably increase sensitivities of SToRM seeds and Mandala seeds. Our runtime is also more efficient than Mandala, as always.

In a nutshell, SpEED performs very well in computing highly sensitive multiple spaced seeds. It improves seeds that are currently used in different softwares. Our results show that our seeds are always more sensitive than Mandala's and are computed much faster. This runtime advantage over Mandala can be especially seen for the BFAST case for which Mandala has not produced any seeds after a day while our program has generated highly sensitive seeds in about 21 seconds.

Chapter 6

Evaluation of Overlap Complexity

Our SpEED algorithm not only produces the best seeds but it does so with much greater speed than any other algorithm. It is the only one able to produce large sets of long seeds. Therefore, the natural question arises: How much can it be improved? Or, differently put, how far are its computed seed from optimal?

A natural answer to this question would be a theoretical result comparing the sensitivity of the seed computed by SpEED against the optimal. Such a result seems to be very difficult to prove. In addition, anything less than an optimal bound on the sensitivity of the seeds of SpEED would be useless since the difference to the optimal may be very small. Therefore, we shall attempt in this section to perform some meaningful exhaustive testing. In order to achieve something nontrivial, we need a very fast implementation of the sensitivity function.

6.1 Limitations on Exhaustive Search

In exhaustive testing, all possible seeds with the given length set, k, w, N and p are computed and sensitivity of each seed set is computed by the Sensitivity algorithm. Therefore, the most important contribution here is a fast implementation of the sensitivity function so that exhaustive testing becomes possible.

In addition, we need to evaluate our work before starting the computations. The number of possible k-seed sets, of weight w region length N and similarity p is

$$\prod_{i=1}^{k} \binom{\ell_i - 2}{w - 2}$$

where ℓ_i is the length of ith seed. This is because each seed must start and end with a 1 and the remaining $w - 2$ 1's can be arranged in any possible way on the $\ell_i - 2$ remaining positions.

This number grows very fast with all parameters and we should pick test cases that are feasible to compute. For each test case we computed this value to estimate the running time of the exhaustive search and see if the case is feasible or not. Also, we tried to have a significant number of seeds in our tests, so that we can see how well the overlap complexity measure works for multiple spaced seeds.

The fast implementation of sensitivity is given in the next section, followed by pour tests at the end of the chapter.

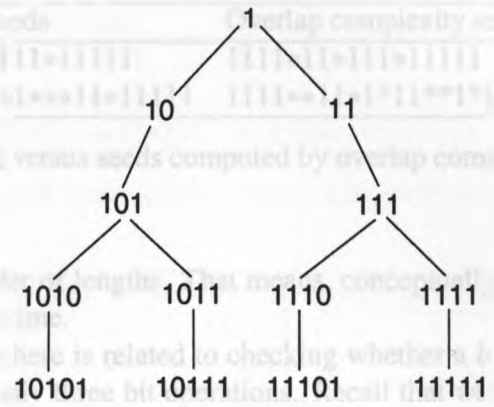

Figure 6.1: The elements of B^r for the seed $1 * 1 * 1$.

6.2 Fast Computation of Sensitivity

In this section, we explain our fast implementation for the Sensitivity function given in Figure 2.2. Instead of giving a long and hard to read pseudocode, we shall explain the basic ideas of our approach.

Recall that we had a set $A = \{s_1, s_2, \ldots, s_k\}$ of seeds and that B denoted the set of all binary strings not hit by A but compatible with some $s \in A$. For example, for the seed $s = 1 * 1 * 1$, we have the set $B = \{1, 01, 11, 101, 111, 0101, 0111, 1101, 1111, 10101, 10111, 11101, 11111\}$.

As our first improvement, we shall store all binary strings, seeds or elements of B, as integers. For seeds it does not matter but for elements of B it is actually simpler to store their reversals (b^r denotes the reversal of b) since they always starts with a 1 and the length is no longer necessary. An important observation is that B is suffix closed, which means B^r is prefix closed. Therefore, we can store it as a tree, where each node c has at most two children, $c0$ and $c1$ (this c is actually a b^r); see Figure 6.1.

The most time consuming part of the entire algorithm is the computation of the set B. Our second improvement is the computation of the set B before hand, together with direct computations of $B(0b)$ from $0b$. We shall also store all the elements of B in one array, as arrays are much faster than the pointers in a tree. In order to allocate memory for it, we need to have a good upper bound on its size. If we define B_{size} as a function from seeds to integers by

$$\begin{aligned} B_{\text{size}}(1) &= 1 \\ B_{\text{size}}(1s) &= B_{\text{size}}(s) \\ B_{\text{size}}(*s) &= 2B_{\text{size}}(s) \end{aligned}$$

then a good (over)estimate for the size of B is

$$\sum_{i=1}^{k} B_{\text{size}}(s_i) \,.$$

An entry in our array for an element b of B will contain the integer value of b^r, the positions of its children (if they exist, or -1 otherwise), the suffix link (position of $B(b^r)^r$), and information whether it is hit or not by some $s \in A$. The array is built by adding all elements of B of the

Optimal seeds	Overlap complexity seeds
111*11*1111*11111	1111*11*111*11111
11111**1*1***11*11111	1111**11*1*11**1*1111

Table 6.1: Optimal seed versus seeds computed by overlap complexity for 2 seeds

same length, in increasing order of lengths. That means, conceptually, we build the tree from top to bottom, one level at the time.

An essential improvement here is related to checking whether a $b \in B$ is compatible with a seed s. This is done using only three bit operations. Recall that we work with the reversals of all the strings. We right shift the seed by $|s| - |b|$ positions and bit AND it with the bit complement of b. The obtained string contains no 1's (which means it equals 0 as an integer) if and only if b is compatible with a.

The obtained sensitivity function is much faster than a naive implementation. It turned out to be much faster than Mandala's as well. It will enable us to perform some nontrivial testing in the next section.

6.3 Tests

We performed three tests: one for $k = 2$, one for $k = 3$ and one for $k = 4$. For each test case we fix the ℓ_i's. We did exhaustive search and found the optimal solution. We set N to 35 and find a value for p that will lead to sensitivity around 80% so that differences between optimal sensitivity and sensitivity of overlap complexity based algorithm is increased. We also generate seeds with the same length sets by overlap complexity simply applying overlap complexity to find seeds with given length set. This can be done by changing the MULTIPLESEEDS algorithm so that it allows fixed lengths. Here are the results of our three tests:

- Parameters: $k = 2, w = 14, N = 35, p = 0.88, l_0 = 17$ and $l_1 = 21$.

optimal sensitivity:	0.828460
our sensitivity:	0.821946

  The difference between sensitivities of our seeds and optimal seeds is about 0.65% which is a small difference and suggest that overlap complexity measure works well in computing these two seeds. Optimal seeds and our seeds are given in Table 6.1.

- Parameters: $k = 3, w = 10, N = 35, p = 0.78, l_0 = 12$ and $l_1 = 14, l_2 = 16$.

optimal sensitivity:	0.818325
our sensitivity:	0.814159

  The difference between sensitivities of our seeds and optimal seeds is about 0.42% which is a small difference even smaller than the previous test case and suggest that overlap complexity measure works well in computing these three seeds. The optimal seeds and our seeds are given in Table 6.2.

Optimal seeds	Overlap complexity seeds
111*11*11111	1111*111*111
111**111*1*111	111*11**1*1111
1111*1****11*111	111*1**1*1**1111

Table 6.2: Optimal seed versus seeds computed by overlap complexity for 3 seeds

Optimal seeds	Overlap complexity seeds
1*111*11	11*1*111
111**11*1	111**1*11
11**1*1*11	11*11**1*1
11*1****111	111***1**11

Table 6.3: Optimal seed versus seeds computed by overlap complexity for 4 seeds

- Parameters: $k = 4, w = 6, N = 35, p = 0.6, l_0 = 8$ and $l_0 = 8, l_1 = 9, l_2 = 10, l_3 = 11$.

optimal sensitivity:	0.849525
our sensitivity:	0.844622

The difference between sensitivities of our seeds and optimal seeds is about 0.50% which is a small difference and suggest that overlap complexity measure works well in computing these four seeds. The optimal seeds and our seeds are given in Table 6.3.

As it can be seen from Tables 6.1, 6.2 and 6.3, the sensitivities of our seeds are very close to the optimal sensitivities in all three cases (less than 1%). Although we did not perform exhaustive testing for larger test cases since they would be very time-consuming and for some cases infeasible, these three test cases are just a few examples that suggest the superiority and efficiency of overlap complexity measure.

Chapter 7

Conclusion

We have succeeded to engineer the overlap-complexity based algorithm so that it computes seeds that are better than any other ones, while being computed orders of magnitude faster. Considering the fact that increasingly many software programs for biological applications use multiple spaced seeds, our software program will be a very useful tool for creating fast the best seeds available. Further research remains to be done in order to adapt the overlap complexity idea to models other than Bernoulli, such as Markov.

Bibliography

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D., (1990), Basic local alignment search tool, *J. Mol. Biol.* **215** 403 – 410.

[2] Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D.J., (1997), Gapped Blast and Psi-Blast: a new generation of protein database search programs, *Nucleic Acids Res.* **25** 3389 – 3402.

[3] Buhler, J., Keich, U., and Sun, Y., (2003), Designing seeds for similarity search in genomic DNA, in: *Proc. of RECOMB03*, ACM Press, 67 – 75.

[4] Brejova,B., Brown, D., and Vinar, T., (2004), Optimal spaced seeds for homologous coding regions. *J. Bioinf. and Comp. Biol.* **1** 595 – 610.

[5] Brejova, B., Brown, D. G., Vinar, T., Vector seeds: An extension to spaced seeds, (2005), *J. Comput. Syst. Sci.* **70**(3) 364 – 380.

[6] Brejova, B., Evidence combination in hidden Markov models for gene prediction., (2005), PhD Thesis. University of Waterloo, Canada.

[7] Brown, D. G., Optimizing multiple seeds for protein homology search, (2005), *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **2**(1) 29 – 38.

[8] Burkhardt, S., and Karkkiainen, J., (2001), Better filtering with gapped q-grams. *Proc. of CPM01*, Lecture Notes in Comput. Sci. **2089**, Springer, 73 – 85.

[9] Califano A, Rigoutsos I (1993) Flash: a fast look-up algorithm for string homology. *Computer Vision and Pattern Recognition*, 1993 Proceedings CVPR 93, 1993 IEEE Computer Society Conference on. 353 – 359.

[10] Chang, J., Raychaudhuri, S. and Altman, R., (2001), Including biological literature improves homology search. *Pac. Symp. Biocomput.* **6** 374 – 383.

[11] Y. Chen, T. Souaiaia, and T. Chen, (2009), PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds, *Bioinformatics* **25**(19) 2514 – 2521.

[12] Choi, K.P., Zeng, F., and Zhang, L., (2004), Good Spaced Seeds for Homology Search, *Bioinformatics* **20** 1053 – 1059.

[13] K. Choi and L. Zhang, "Sensitivity analysis and efficient method for identifying optimal spaced seeds," Journal of Computer and System Sciences, vol. 68, pp. 2240, 2004.

[14] Delcher,A.L., Kasif,S., Fleischmann,R.D., Peterson,J., White,O. and Salzberg,S.L. (1999) Alignment of whole genomes. *Nucleic Acids Res.* **27** 2369 – 2376.

[15] Girdea M., Noe L., and G. Kucherov, Read mapping tool for AB SOLiD data, (2009), in *Proceedings of the 9th International Workshopon Algorithms in Bioinformatics (WABI 09)*, Philadelphia, Pa, USA.

[16] Homer N., Merriman B., Nelson SF, (2009), BFAST: an alignment tool for large scale genome resequencing. *PLoS ONE* **4**(11) e7767.

[17] Huang, X., Miller, W., (1991) A time-efficient, linear-space local similarity algorithm, *Adv. Appl. Math.* **12** 337 – 357.

[18] Ilie L., Ilie S., (2007), Multiple spaced seeds for homology search. Bioinformatics **23** 2969 – 2977.

[19] Keich, U., Li, M., Ma, B., and Tromp, J., (2004), On spaced seeds for similarity search, *Discrete Appl. Math.* 3, 253 – 263.

[20] Kent, W. J., (2002) BLATthe BLAST-like alignment tool. *Genome Research* **12**(4) 656 – 664.

[21] Kucherov, G., Noe, L., and Ponty, Y., (2004), Estimating seed sensitivity on homogeneous alignments, in: *Proc. IEEE 4th Symp. on Bioinformatics and Bioengineering*, Taiwan, 387 – 394.

[22] Kucherov G., Noe L., and Roytberg M., Multiseed lossless filtration, (2005), *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **2** (1) 51 – 61.

[23] Kucherov, G., Noe, L., Roytberg, M., (2006) A unifying framework for seed sensitivity and its application to subset seeds. *J. Bioinf. Comput. Biology* **4** 553 – 569.

[24] G. Kucherov, L. Noe, M. Roytberg, (2007) Subset seed automaton, *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA 2007)*, Lecture Notes in Comput. Sci. **4783** 180 – 191.

[25] Kurtz, S., Schleiermacher, C., (1999) REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics* **15**(5) 426 – 427.

[26] Li M, Ma B, Kisman D, Tromp J, (2004), PatternHunter II: highly sensitive and fast homology search. *J. Bioinform Comput Biol* **2** 417 – 439.

[27] Li, M., Ma, B., and Zhang L. (2006) Superiority and complexity of the spaced seeds. Proc. SODA'06, 444453. Ma, B., Tromp, J., and Li, M. 2002. PatternHunter: faster and more sensitive homology search. *Bioinformatics* **18** 440 – 445.

[28] Li H., Ruan J., and Durbin R., (2008),Mapping short DNA sequencing reads and calling variants using mapping quality scores, *Genome Research* **18**(11) 1851 – 1858.

[29] Lin H., Zhang Z., Zhang M. Q., Ma B., and Li M.,(2008), ZOOM! Zillions of oligos mapped, *Bioinformatics* **24**(21) 2431 – 2437.

[30] Lipman, DJ., Pearson, WR (1985). "Rapid and sensitive protein similarity searches". *Science* **227** (4693): 1435 – 1441.

[31] Ma, B., Tromp, J., and Li, M., (2002), PatternHunter: faster and more sensitive homology search, *Bioinformatics* **18** 440 – 445.

[32] Noe L., and Kucherov G., (2004) Improved hit criteria for DNA local alignment. *BMC Bioinformatics* **5** 149 – 149.

[33] Rumble S.M., P. Lacroute, Dalca A. V., Fiume M., Sidow A., and Brudno M., (2009), SHRiMP: accurate mapping of short color-space reads. *PLoS Comput. Biol.* **5** e1000386.

[34] Smith, T.F., and Waterman, M.S., (1981), Identification of common molecular subsequences, *J. Mol. Biol.* **147** 195 – 197.

[35] States, D., SENSEI website: `http://stateslab.wustl.edu/software/sensei/`

[36] Sun, Y. and Buhler, J. 2005. Designing multiple simultaneous seeds for DNA similarity search. *J. Comput. Biol.* **12** 847 – 861.

[37] Yang, I.-H.,Wang, S.-H., Chen, H.-H., Huang, P.-H., and Chao, K.-M., (2004) Efficient methods for generating optimal single and multiple spaced seeds, *Proc. of IEEE 4th Symp. on Bioinformatics and Bioengineering*, Taiwan, 411 – 418.

[38] Zhang,Z., Schwartz,S., Wagner,L. and Miller,W. (2000) A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.* **7** 203 – 214.

Appendix A

Lists of Arrays

```
int seedLength_min_k2[13][66] = {
 12,12,12,12,12,12,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,
 12,12,13,13,13,13,13,13,13,13,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,13,13,13,13,13,13,13,13,13,13,12,12,12,12,12,12,12,12,12,12,12,12,12,
 14,14,14,14,14,14,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
 16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
 18,18,18,18,18,18,18,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,
 19,19,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,
 19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,
 20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,
 21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,
 22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,
 22,22,22,22,22,22,22,22,22,22,22,23,23,23,23,23,23,23,23,24,24,24,24,24,24,25,25,25,25,25,25,26,26,26,26,26,26,27,27,27,27,27,27,27,27,28,28,28,28,28,28,28,28,28,28,28,28,28,27,27,27,27,27,26,26,26,
 23,23,23,23,23,23,23,23,23,23,23,23,24,24,24,24,24,24,25,25,25,25,25,25,26,26,26,26,26,27,27,27,27,27,28,28,28,28,28,28,28,28,29,29,29,29,29,29,29,29,29,28,28,28,28,28,28,27,27,27,27,26,26,25,25,
 25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,26,26,26,26,26,26,26,26,26,26,26,26,27,27,27,27,27,27,27,27,27,27,27,27,27,27,28,28,28,28,28,28,28,28,28,27,27,27,27,27,27,27,
};

int seedLength_max_k2[13][66] = {
 16,16,16,16,16,17,17,17,17,17,17,17,18,18,18,18,18,18,18,19,19,19,19,19,19,19,20,20,20,20,20,20,20,20,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,20,20,20,20,
 18,18,17,17,17,17,16,16,16,16,16,16,16,16,16,17,17,17,17,17,18,18,18,18,19,19,19,20,20,20,21,21,21,22,22,22,23,23,23,24,24,24,25,25,25,25,25,26,26,26,26,26,26,26,26,26,26,25,25,25,24,24,23,23,22,
 20,20,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,20,20,20,20,20,21,21,21,21,22,22,22,23,23,23,23,24,24,24,25,25,25,25,25,26,26,26,26,26,27,27,27,27,27,27,27,27,27,27,27,27,26,26,26,26,25,25,
 21,22,22,22,22,22,22,22,22,22,23,23,23,23,23,23,23,23,23,24,24,24,24,24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,26,26,26,26,26,26,26,26,26,26,27,27,27,27,27,27,27,27,27,28,28,
 23,23,24,24,24,24,25,25,25,25,25,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,27,27,27,27,27,27,27,27,28,28,28,28,29,29,
 25,25,25,25,25,25,25,26,26,26,26,26,26,26,26,26,27,27,27,27,27,27,27,27,27,27,27,27,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,29,29,
 26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,27,27,27,27,27,27,27,28,28,28,28,28,28,29,29,29,29,30,30,30,30,30,30,30,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,30,30,30,
 27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,28,28,28,28,28,28,28,28,28,29,29,29,29,29,29,30,30,30,30,30,30,31,31,31,31,31,32,32,32,32,32,32,32,33,33,33,33,33,34,34,34,34,34,34,34,
 27,27,27,28,28,28,28,28,28,28,28,29,29,29,29,29,29,29,29,29,29,29,29,30,30,30,30,30,30,30,30,30,30,30,30,31,31,31,31,31,31,32,32,32,32,32,32,32,33,33,33,33,34,34,34,35,35,35,36,36,36,37,37,38,38,39,
 27,28,28,28,29,29,29,30,30,30,30,30,31,31,31,31,31,31,32,32,32,32,32,32,32,32,32,32,33,33,33,33,33,33,33,33,33,33,33,34,34,34,34,34,34,34,34,34,35,35,35,35,35,36,36,36,36,37,37,37,38,38,38,39,39,40,40,
 28,28,29,29,30,30,30,31,31,31,32,32,32,32,33,33,33,33,34,34,34,34,35,35,35,35,35,35,35,36,36,36,36,36,36,36,36,36,37,37,37,37,37,37,37,37,37,38,38,38,38,38,38,38,38,38,38,38,38,39,39,39,39,39,39,
 29,29,29,30,30,30,31,31,31,32,32,32,32,33,33,33,33,34,34,34,34,35,35,35,35,36,36,36,36,37,37,37,37,37,37,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,38,
 29,29,30,30,30,31,31,31,32,32,32,32,33,33,33,33,33,34,34,34,34,34,35,35,35,35,35,36,36,36,36,36,36,36,36,36,37,37,37,37,37,37,37,38,38,38,38,38,38,38,38,38,38,38,38,38,38,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,
};
```

```
int seedLength_min_k3[13][66] = {
 12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,
 13,13,14,14,14,14,14,14,14,14,14,14,15,15,15,15,15,15,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,13,13,13,13,13,13,13,13,13,13,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,13,13,13,13,
 14,14,15,15,15,15,15,15,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,15,15,15,15,15,15,15,15,14,14,14,14,14,14,14,14,14,14,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,14,14,14,14,14,
 15,15,15,15,16,16,16,16,16,16,16,16,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,16,16,16,16,16,16,16,16,16,16,16,16,16,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
 16,16,16,16,16,16,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
 18,18,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,17,17,17,17,
 19,19,19,19,19,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,
 20,20,20,19,19,19,19,19,19,19,19,19,19,19,19,19,19,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,
 20,20,20,20,20,20,20,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,23,23,23,23,23,23,23,23,23,23,23,
 21,21,21,21,21,21,21,22,22,22,22,22,22,22,22,22,22,22,22,22,22,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,
 22,22,22,22,22,22,22,22,23,23,23,23,23,23,23,23,23,23,23,23,23,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,23,23,23,23,23,23,23,23,23,23,
 23,23,23,23,23,23,23,23,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,
 24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,
};

int seedLength_max_k3[13][66] = {
 22,22,22,22,22,22,22,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,
 22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,23,23,23,
 23,23,23,23,23,23,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,
 25,25,25,25,25,26,26,26,26,26,26,26,26,26,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,27,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,25,25,25,25,25,25,25,25,25,25,
 26,26,27,27,27,27,28,28,28,28,28,29,29,29,29,29,29,29,29,29,29,29,29,29,29,29,30,30,30,30,29,29,29,29,29,29,29,29,29,29,29,29,29,29,29,29,29,29,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,
 26,27,27,27,28,28,29,29,29,30,30,30,30,30,30,31,31,31,31,31,31,31,31,31,32,32,32,32,32,32,32,32,32,32,32,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,31,32,32,32,32,32,32,33,33,33,
 26,27,27,28,28,29,29,30,30,30,31,31,31,32,32,32,32,32,32,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,34,34,34,34,34,34,35,35,35,35,36,36,37,37,38,38,
 27,28,28,29,29,30,30,31,31,31,32,32,32,32,33,33,33,33,33,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,35,35,35,35,35,35,35,36,36,36,36,37,37,37,38,38,39,39,40,40,41,
 28,29,29,30,30,31,31,32,32,32,33,33,33,34,34,34,34,34,35,35,35,35,35,35,35,35,35,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,37,37,37,37,37,37,38,38,38,38,39,39,39,40,40,41,41,41,42,
 28,29,30,30,31,31,32,32,33,33,34,34,35,35,35,36,36,36,36,37,37,37,37,37,38,38,38,38,38,38,38,38,38,38,38,39,39,39,39,39,39,39,39,39,39,39,39,39,40,40,40,40,40,40,40,41,41,41,41,42,42,42,43,43,
 28,29,30,30,31,31,32,32,33,34,34,35,35,36,36,37,37,37,38,38,38,38,39,39,39,40,40,40,40,40,40,41,41,41,41,41,41,41,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,43,43,43,43,43,43,43,43,43,43,43,43,44,44,44,44,
 28,29,30,30,31,31,32,32,33,34,34,35,35,36,36,37,37,37,38,38,38,39,39,39,40,40,40,40,41,41,41,42,42,42,43,43,43,43,44,44,44,44,44,44,44,44,45,45,45,45,45,45,45,45,45,45,45,46,46,46,46,46,46,45,45,45,45,45,45,45,45,
 30,30,31,31,31,31,32,32,33,33,33,34,34,34,35,35,36,36,36,36,37,37,37,38,38,38,39,39,39,40,40,40,40,41,41,42,42,42,43,43,43,43,44,44,44,44,44,45,45,45,46,46,46,46,47,47,47,47,47,47,48,48,48,48,48,48,48,48,48,48,48,48,48,48,47,47,47,47,46,46,
};
```

```
int seedLength_min_k4[13][66] = {
 12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,
 13,12,12,12,12,12,12,12,12,12,12,12,13,13,13,13,13,13,13,13,13,13,13,14,14,14,14,14,14,14,14,14,14,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14,14,14,14,14,13,13,13,
 14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,15,15,15,15,15,15,15,15,15,15,15,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
 16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
 17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,
 17,17,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,
 18,18,18,18,18,18,18,18,19,19,19,19,19,19,19,19,19,19,19,19,19,19,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,19,19,19,19,19,19,19,19,18,18,18,18,
 19,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,
 21,21,21,21,21,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,22,22,
 22,22,22,22,22,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,23,23,23,23,23,23,
 22,22,22,23,23,23,23,23,23,23,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,
 23,23,23,23,23,23,24,24,24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,24,24,24,
 24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,
};

 int seedLength_max_k4[13][66] = {
 23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,22,22,22,22,22,22,22,21,21,21,21,21,21,21,20,20,20,20,20,20,20,20,20,20,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,20,20,20,20,
 25,25,24,24,24,24,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,24,24,24,24,24,25,25,25,25,25,26,26,26,26,26,27,27,27,27,27,27,28,28,28,28,28,28,28,28,28,28,28,28,27,27,27,27,26,26,26,25,25,24,24,23,
 26,26,26,26,25,25,25,25,25,25,25,25,26,26,26,26,26,26,27,27,27,27,27,28,28,28,28,29,29,29,29,29,30,30,30,30,30,31,31,31,31,31,31,31,31,31,31,31,31,31,30,30,30,30,29,29,28,28,27,27,26,25,25,24,
 26,26,27,27,27,28,28,28,28,29,29,29,29,30,30,30,30,31,31,31,31,31,31,32,32,32,32,32,32,32,32,32,32,33,33,33,33,33,33,33,33,33,32,32,32,32,32,32,32,32,31,31,31,31,30,30,30,29,29,29,28,28,27,27,26,26,25,25,
 26,27,27,28,29,29,30,30,31,31,31,32,32,32,33,33,33,33,33,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,33,33,33,33,33,33,33,33,32,32,32,32,32,31,31,31,31,31,30,30,30,30,29,29,29,29,28,28,28,28,
 26,27,27,28,29,29,30,30,31,31,32,32,32,33,33,33,34,34,34,34,34,34,34,35,35,35,35,35,35,35,34,34,34,34,34,34,34,34,34,34,34,34,34,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,34,34,34,34,34,
 26,27,27,28,29,29,30,30,31,31,31,32,32,32,33,33,33,33,33,34,34,34,34,34,34,34,34,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,35,36,36,36,36,36,36,37,37,37,37,38,38,38,39,39,40,40,
 27,28,28,29,30,30,31,31,32,32,32,33,33,34,34,34,34,35,35,35,35,35,36,36,36,36,36,36,36,36,36,37,37,37,37,37,37,37,37,37,37,37,37,37,37,37,37,38,38,38,38,39,39,39,39,40,40,40,41,41,
 28,29,29,30,31,31,32,33,33,34,34,34,35,35,36,36,36,37,37,37,37,38,38,38,38,38,38,38,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,40,40,40,40,
 29,29,30,31,32,32,33,33,34,35,35,36,36,36,37,37,38,38,38,39,39,39,40,40,40,40,40,41,41,41,41,41,41,41,41,41,42,42,42,42,42,42,42,42,41,41,41,41,41,41,41,41,41,41,41,41,41,41,40,40,40,
 29,30,30,31,32,33,33,34,34,35,35,36,36,37,37,38,38,38,39,39,40,40,40,41,41,41,42,42,42,42,43,43,43,43,43,43,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,43,43,43,43,43,43,42,42,42,
 29,30,31,31,32,33,33,34,35,35,36,36,37,37,38,38,39,39,40,40,41,41,41,42,42,43,43,43,43,44,44,44,44,44,45,45,45,45,45,45,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,45,45,45,45,45,45,44,44,44,
 30,31,31,32,33,33,34,35,35,36,36,37,38,38,39,39,39,40,40,41,41,42,42,42,43,43,43,44,44,44,44,44,45,45,45,45,45,45,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,45,45,45,45,
};
```

```
int seedLength_min_k10[13][66] = {
 12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,
 13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,
 14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
 15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
 16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
 17,17,17,17,17,17,17,17,17,17,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,
 18,18,18,18,18,18,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,18,18,18,18,18,18,18,18,18,18,18,
 19,19,19,19,19,19,19,19,19,19,19,19,19,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,19,19,19,19,19,19,
 20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,20,20,20,20,20,
 21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,21,
 23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,23,23,23,23,23,
 24,24,24,24,24,24,24,24,24,25,25,25,25,25,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,
 24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,24,24,24,24,24,24,24,24,24,24,24,
};


int seedLength_max_k10[13][66] = {
 24,24,24,24,23,23,23,23,23,23,23,22,22,22,22,22,22,22,22,21,21,21,21,21,21,21,21,21,21,21,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,
 25,25,26,26,27,27,27,28,28,28,28,28,28,28,28,28,28,28,28,28,28,27,27,27,26,26,26,25,25,25,24,24,24,23,23,22,22,22,21,21,21,20,20,20,19,19,19,19,19,18,18,18,18,18,18,18,19,19,19,19,20,20,20,21,21,22,
 25,26,27,27,28,28,29,29,30,30,30,31,31,31,31,31,31,31,31,31,31,31,30,30,30,30,29,29,29,28,28,28,28,27,27,27,26,26,25,25,25,24,24,24,23,23,23,22,22,22,21,21,21,21,21,21,21,21,21,22,22,22,23,23,23,24,
 25,26,27,27,28,29,29,29,30,30,31,31,31,31,31,32,32,32,32,32,32,32,32,32,32,32,32,32,31,31,31,31,31,30,30,30,30,30,29,29,29,29,28,28,28,28,27,27,27,27,27,26,26,26,26,26,26,26,25,25,25,25,25,25,26,26,26,26,
 26,27,27,28,28,29,29,30,30,30,31,31,31,32,32,32,32,32,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,32,32,32,32,32,32,31,31,31,31,31,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,
 27,27,28,29,29,30,30,31,31,31,32,32,33,33,33,33,34,34,34,34,34,34,35,35,35,35,35,35,35,35,35,35,35,35,35,34,34,34,34,34,34,34,34,33,33,33,33,33,33,33,32,32,32,32,32,32,31,31,31,31,31,30,30,30,30,30,
 27,28,29,29,30,31,31,32,32,33,33,34,34,34,35,35,35,35,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,35,35,35,35,35,35,34,34,34,34,34,34,33,33,33,33,33,33,33,32,32,32,32,32,32,32,32,32,32,
 27,28,29,29,30,31,32,32,33,33,34,34,35,35,35,35,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,35,35,35,35,35,35,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,34,
 27,28,29,29,30,31,31,32,33,33,34,34,35,35,36,36,36,37,37,37,38,38,38,38,38,38,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,39,38,38,38,38,38,38,38,38,38,37,37,37,37,37,37,37,36,36,36,36,36,
 28,29,29,30,30,31,32,32,33,33,34,34,35,35,36,36,36,37,37,37,38,38,38,39,39,39,39,40,40,40,40,40,40,40,40,41,41,41,41,41,41,41,41,41,41,41,41,40,40,40,40,40,40,40,40,40,40,39,39,39,39,39,39,38,
 29,30,30,31,31,32,32,33,33,34,34,34,35,35,36,36,36,37,37,37,38,38,38,39,39,39,39,40,40,40,40,40,40,41,41,41,41,41,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,41,41,41,41,
 30,30,31,31,32,32,32,33,33,34,34,35,35,35,36,36,37,37,37,38,38,38,39,39,39,40,40,40,40,41,41,41,41,42,42,42,42,42,43,43,43,43,43,43,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,43,43,43,43,
 29,30,30,31,31,32,33,33,34,34,35,35,36,36,37,37,37,38,38,39,39,39,40,40,40,40,41,41,41,41,42,42,42,42,43,43,43,43,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,44,
};
```