2010

MULTIPLE SPACED SEEDS FOR OLIGONUCLEOTIDE DESIGN

Shima Khoshraftar
Western University

MULTIPLE SPACED SEEDS FOR OLIGONUCLEOTIDE DESIGN

(Spine title: Multiple spaced seeds for oligonucleotide design)
(Thesis format: Monograph)

by

Shima Khoshraftar

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. Mark Daley

.....
Dr. Mahmoud El-Sakka

.....
Dr. Jagath Samarabandu

The thesis by

Shima Khoshraftar

entitled:

Multiple spaced seeds for oligonucleotide design

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

.....
Date

.....
Chair of the Thesis Examination Board

Abstract

An oligonucleotide is a small piece of DNA or RNA molecule, which is designed to hybridize with a unique position in a target sequence. DNA oligonucleotides have many applications such as gene identification, PCR (polymerase chain reaction) amplification, or DNA microarrays. One of the crucial issues in designing good oligonucleotide is to minimize the chance of cross-hybridization. Various heuristic algorithms are used to filter out the unsuitable regions before checking for cross-hybridization. The most successful ones are based on seeds because of their efficiency and ability to tolerate mismatches. The quality of the seeds is essential in this process. We present a sound framework for evaluating seed quality for oligonucleotide design and show that multiple spaced seeds are expected to provide the best discrimination between oligos and non-oligos.

Keywords: DNA, oligonucleotides, multiple spaced seeds, discriminability, efficiency

Acknowlegements

I would like to deeply thank my supervisors, Dr. Lucian Ilie and Dr. Silvana Ilie, who have supported me throughout my thesis. In particular, I am grateful to Dr. Lucian Ilie for all his valuable guidance and encouragement at the various stages of the research.
Special thanks goes to my parents for their endless love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An oligonucleotide is a small piece of DNA or RNA molecule, which is designed to hybridize with a unique position in a target sequence. In this way the target sequence can be uniquely identified using the oligonucleotide as a probe. DNA oligonucleotides have many applications such as gene identification, PCR (polymerase chain reaction) amplification, or DNA microarrays.

Many software programs have been written to construct good DNA oligonucleotides, such as ProbeSelect [9], PROBESEL [7], ProMide [17], OligoArray [20, 21], ArrayOligoSelector [2], OligoWiz [14], ROSO [18], GoArrays [19], and ProDesign [4]. One of the crucial issues in designing good oligonucleotides is to minimize the chance of cross-hybridization. Essentially, these programs apply heuristic algorithms to filter out the unsuitable regions before checking for cross-hybridization.

The underlying algorithms of these software programs are based on one or more of the following tools: suffix trees, suffix arrays, sequence alignments, seeds. Whereas suffix trees and suffix arrays are well known indexes for exact text search, efficient counterparts do not exist for approximate search and heuristic algorithms are used instead.

Seeds were made highly popular by BLAST, the most widely used software program in bioinformatics. Instead of the quadratic dynamic programming exact algorithm of Smith-Waterman [23], which is infeasible for long sequences, BLAST searches for 11 contiguous matches between the sequences as an indicator of potential local similarity. PatternHunter [11] uses a spaced seed, that is, the 11 matches are not consecutive but separated by don't care positions. The sensitivity of this spaced seed is significantly higher than that of BLAST's contiguous seed. Many software programs use a combination of several spaced seeds as the chances to find the similarities increase very much. In fact, multiple spaced seeds quickly became the state-of-the-art in similarity search in biological applications due to their great efficiency and flexibility. The most successful DNA oligonucleotide design algorithms are based on seeds. The quality of the seeds is essential in this process.

To our best knowledge, there is only one study, due to [3], attempting to evaluate the quality of the seeds for oligonucleotide design. We describe very clearly that their approach is flawed. Essentially, the F-score statistical measure for a test's accuracy is inappropriately used and also the approach combines two measures in a way that cannot be meaningfully interpreted.

We present a sound framework for evaluating seed quality for oligonucleotide design. We show that multiple spaced seeds are expected to provide the best discrimination between oligos

and non-oligos. However, the increased discrimination power comes at the price of decreased efficiency. It is interesting to notice that spaced seeds are both more powerful at discrimination but also more efficient than contiguous seeds. They are followed very closely by transition-constrained seeds [15]. After that, increasing the number of seeds increases discriminability but decreases efficiency.

The thesis is organized as follows. In Chapter 2 the basic notions needed in the thesis, such as DNA, oligonucleotides, alignments, seeds, as well as few basic notions from statistics are introduced. Current programs for designing oligonucleotides are briefly described in Chapter 3. In Chapter 4 we explain the problems with the approach of [3] and explain why, intuitively, multiple spaced seeds are expected to be good candidates. We also give a fast algorithm for constructing good seeds and then present our new approach for seed evaluation. We performed several tests in Chapter 5 which confirm the findings we described above. The thesis concludes in Chapter 6 with a few remarks about the importance of our contribution and further research that can now be done to improve the existing software programs for designing oligonucleotides.

Chapter 2

Basic Notions

This chapter contains the basic notions that are needed throughout the thesis. We introduce DNA, oligonucleotides, sequence alignments, a detailed introduction to spaced seeds, and a few basic concepts from statistics.

2.1 DNA

DNA is a nucleic acid present in the cells of all living organisms. It encodes the genetic material which determines what an organism will develop into and what an organism functions are. DNA structure is composed of two strands made of four types of nucleobases Adenine (abbreviated A), Cytosine (C), Guanine (G) and Thymine (T), of which A and T are complementary, as are C and G. The two DNA strands are held together in the shape of a double helix by hydrogen bonds between complementary bases; see Figure 2.1.

Figure 2.1: DNA double helix

2.2 Oligonucleotides

An *oligonucleotide* (or *oligo*) is a piece of DNA or RNA molecule that can be either short or long. Short oligos typically consist of 20 to 25 bases and long ones 50 or 70 bases. The main

Figure 2.2: Three oligos of size 35bp; oligo1 is perfectly matching the target DNA: probe-target identity = 100%, continuous stretch = 35bp; oligo 2 and oligo 3 have the same probe-target identity = 85% but the continuous stretch is very different: 25bp for oligo 2 and only 7bp for oligo 3.

property of oligos is their ability to readily bind to their complementary sequence. However, given a set of genomic sequences called target sequences or master sequences, the challenge is to select oligos, called probes, that only hybridize to the intended target sequence and not to any other sequence. In other words, oligos must have high specificity in detecting target sequences. There are many parameters that affect probe specificity; here are the most important ones:

- Probe-target identity, the ratio of complementary nucleotides between oligo and target DNA,
- Continuous stretch, the longest continuous sequence of complementary base pairs, and
- Hybridization free energy, the resulting energy from the structure formed by binding a probe and its target. The structure with minimum free energy is desirable because it tends to be more stable.

Figure 2.2 contains several examples of oligos binding to their targets with varying probe-target identity and maximum continuous stretch.

Hybridization free energy ΔG is an important factor in oligo selection as it measures the stability of the binding formed between oligo and a sequence. Usually a threshold is set for binding free energy and oligos with lower free energy than the threshold will be selected. Oligos with low binding free energy with target sequence and high binding free energy with non-target sequences are desirable because it indicates they tend to form a more stable binding with their targets rather than non-targets which contributes to reduction in cross-hybridization. One of the standard methods for computing binding free energy is the standard dynamic programming approach based on thermodynamic parameters. Based on this approach, the optimal alignment between oligo and sequence is computed because it is usually the same as the lowest energy structure. Further information about alignments is presented at the end of this section. Given the optimal alignment, different algorithms account for the effects of alignment mismatches, matches by different thermodynamic parameters. The most common thermodynamic parameters are the SantaLucia parameters [22]. There are also some heuristic approaches which calculate the free energy faster than dynamic programming approach.

Hybridization melting temperature T_m is another parameter applied in some oligo design programs. Melting temperature between an oligo and a sequence is the temperature in which

50% of the oligo population are single stranded and 50% are duplexed. The more stable a hybridization is, the higher temperature is necessary to break the hydrogen bonds between the bases (causes of the oligo target binding). Most of the oligo design software apply a nearest-neighbor model using SantaLucia parameters [22] for T_m estimation.

$$T_m = \frac{\Delta H}{\Delta S + Rln(c/4)} - 273.15, \tag{2.1}$$

where ΔH and ΔS are the enthalpy and entropy for oligo and sequence helix formation, R is the universal gas constant (1.99 cal/K.mol), and c is the total molar concentration of strands which is not generally known and is set to different constants by different algorithms.

As explained, both T_m and ΔG yield information about the stability of the oligo target helix structure. It is recommended that, since T_m and ΔG together can provide much more qualitative thermodynamic understanding of helix structure formed between oligo and target than either of the two values alone, oligo design programs consider the two values together.

Furthermore, the probe must be unique, even with an allowable number of mismatches, to only one target. Otherwise the probe will hybridize with more than one sequence and is not considered as an appropriate probe in applications.

Oligos have many uses in molecular biology and medicine. One of the most powerful applications of oligos in genomic research is in microarrays. A DNA or RNA microarray consists of arrayed series of thousands of oligos. Thus, they can probe a large number of targets simultaneously, resulting in significant reduction in time and cost. DNA microarrays have enabled scientists to carry out research on previously intractable problems and to uncover novel gene targets which might be underlying genetic causes of many human diseases. Biological defense and environmental monitoring are among the applications of microarrays in biomedical science.

The microarray experiment can be conducted at even higher throughput and greater convenience using technological advances. However, several careful consideration must be taken into account in microarray design to produce useful data. Particularly, they should rely on probe specificity and probe sensitivity, and a unifying algorithm that interprets the multiple signals coming from array probes. Probes must be unique to the target gene and exhibit high specificity to be able to distinguish between closely related target sequences. Probe sensitivity to targets is also important because it must be able to detect targets. In addition, uniformity is required in order that probes function optimally under the same melting temperature and other experiment conditions.

2.3 Sequence alignments

A sequence alignment is an arrangement of two biological sequences to identify their regions of similarity. The similarity between sequences may indicate their functional and structural relationships. To define alignment formally, consider two sequences X and Y of length n and m respectively. An alignment maps X and Y into new strings X' and Y' of the same length which

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

Table 2.1: Matrix for the local alignment between X=GATACTTG and Y = ATAGTCTA. The backtrack path is specified by red.

differ from X and Y in that they may contain space characters. The score of an alignment is

$$\sum_{i=1}^{L} s(X'[i], Y'[i]), \tag{2.2}$$

where $X'[i]$ denotes the ith character of X', $s(X'[i], Y'[i])$ is the score of aligning the two characters $X'[i]$ and $Y'[i]$, and L is the common length of X' and Y'. An *optimal* alignment of two sequences is an alignment with the highest alignment score.

Sequences can be aligned together either in a global or local manner. A global alignment is an alignment in which all of the characters in both sequences participate in the alignment. However, local alignment algorithms compare segments of all possible lengths instead of considering the total sequence and detect similar regions between two sequences. The best known such algorithm is due to Smith-Waterman [23] and is based on a dynamic programming technique. The algorithm builds a matrix which contains in its (i, j) entry the highest score of a local alignment ending at position i in X and position j in Y. The highest score of a local alignment is given by the highest value in the matrix and the alignment itself is obtained by tracing back in the matrix starting from the position with the highest value.

The idea of constructing the matrix is as follows. Put $X = x_1, ..x_n$ and $Y = y_1, ..., y_m$ and let $T_{i,j}$ be the matrix, where $1 \leq i \leq n$ and $1 \leq j \leq m$. Then matrix is filled recurrently as follows:

$$\begin{aligned} &T_{0,j} = 0, T_{i,0} = 0, \text{ for all } i, j \\ &T_{i,j} = \max\{0, T_{i-1,j-1} + s(x_i, y_j), T_{i,j-1} + s(-, y_j), T_{i-1,j} + s(x_i, -)\} \end{aligned} \tag{2.3}$$

where $s(x_i, -)$ is the score of aligning the character x_i with a space character. As an example, let X = GATACTTG, Y = ATAGTCTA and the scores be $s(x_i, y_j) = 2$ if $x_i = y_j$, $s(x_i, y_j) = -1$ if $x_i \neq y_j$, and $s(x_i, -) = s(-, y_j) = -2$. The matrix for the alignment is shown in Table 2.1. To obtain the local alignment, the algorithm starts tracing from the entry with highest score to get to an entry value 0. It goes back to the biggest value among those in positions $(i - 1, j)$, $(i, j - 1)$, and $(i - 1, j - 1)$ that produced the maximum in the recurrence. The backtrack path is specified with red characters in the matrix and the corresponding alignment is given in Figure 2.3.

```
Column  0 1 2 3 4 5 6 7 8
X       G A T A C T - T G
Y         A T A G T C T A
```

Figure 2.3: A local alignment between X=GATACTTG and Y = ATAGTCTA which detects the similar region in two sequences. The alignment starts from column 1 to 7.

Let the lengths of sequences be n and m. Then, building the matrix and backtracking it to find the optimal alignment takes $O(nm)$ time. Because of the quadratic time complexity of the dynamic programming algorithm, seeds are used in many alignment applications in practice. Further information about seeds and how they work will be given in next section.

2.4 Seeds

The dynamic programming algorithm of Smith-Waterman [23] can find all alignments but the quadratic time complexity makes it impossible to use it for large sequences. Therefore, researchers searched for faster solutions. BLAST [1] the most widely used software for alignment search introduced the following very simple idea. If two sequences have a fairly long substring in common, then they might be actually similar. The default length for the common substring is 11. Put differently, the two sequences need to have 11 consecutive positions that are identical, or 11 matches, and it is represented by the BLAST seed 11111111111; a 1 stands for a match.

The main idea is that it is easier to look for exact matches than for approximate ones. However, Ma et al. [11] noticed that searching for 11 matches that are not consecutive has a higher chance of finding actual alignments. Their seed is

$$111*1**1*1**11*111\,,$$

where 1's stand for matches as before and *'s mean don't care positions. That is, when searching for a potential alignment, only positions corresponding to 1's are checked, whereas those corresponding to *'s are ignored. Such a seed is called *spaced* as opposed to the *contiguous* seed of BLAST. The new seed, implemented in the PatternHunter software, [11], is much more sensitive, in the sense that it has a much higher chance of finding actual alignments.

The number of 1's in a seed is called the *weight* of the seed whereas the total number of symbols is the *length*.

The sensitivity is, intuitively, a measure of a seed's ability to detect similar segments between biological sequences. In order to define the sensitivity precisely, we need a formal model.

First, given two aligned sequences, we define the *similarity level*, p, as the probability of having a match between the two sequences. Then we model the alignment as a Bernoulli random sequence R of 0's and 1's so that the probability of a 1 is p, the similarity level. We shall denote the length of R by N. Given a seed s, we say that s hits R at position k if aligning the end of s with position k in R causes all 1's in s to align with 1's in R. An example of a hit is shown in Figure 2.4.

R: 1011111101101101111110110
s: 111*1**1*1**11*111

Figure 2.4: An example of a hit using PatternHunter's seed; the seed s hits the random region R at position 21.

Formally, the sensitivity of s is defined as the probability that s hits R. Notice that this probability depends, besides s, on the similarity p and length of the random region N. The sensitivity can be computed by the dynamic programming algorithm of [10].

A biologically motivated variant of spaced seeds, *transition-constrained seeds*, was introduced in [15] and used in their YASS software program. Such a seed contains, in addition to 1's for matches and *'s for don't cares, the new character @ which stands for either a match or a transition, that is, a substitution A↔G or C↔T. The biological motivation for this is that transitions are more common than transversions, that is, A/G↔C/T. The seed used in YASS is

1@1**11**1*11@1 ,

The most powerful extension is obtained when several seeds are used together: *multiple spaced seeds*. The sensitivity is greatly increased. For instance, PatternHunter II, [10], uses 16 such seeds, each having 11 matches. A multiple spaced seed hits a random region R when one of its seeds does so. Therefore, the definition of sensitivity is immediately extended to the case of multiple seeds. Also the dynamic programming algorithm that computes it is extended as well.

In practice, the spaced seeds are used by computing seed hashes and then searching for those. A *seed hash* is obtained from a seed s by replacing all the 1's of s with each of the letters A, C, G, T. That means, if the seed has weight w, then it has 4^w hashes. These hashes are then searched for in both sequences to be considered. Any time the same hash is found in both sequences, the two positions constitute a potential alignment and need to be checked.

2.5 Basic statistics

Precision and recall are two widely used metrics for evaluating the performance of classification algorithms. For better understanding of this two metrics we use a classification example. Assume that we have a set of different items and a classifier categorizes the items into the ones that belong to a class and ones that do not. We want to evaluate the correctness of the classification. The required values in the evaluation are true positives, false positives, true negatives and false negatives which are defined below; see also Table 2.2:

- True Positive (TP) : The number of items correctly labeled as belonging to a class.
- False Positive (FP) : The number of items incorrectly labeled as belonging to the class.
- True Negative (TN) : The number of items correctly labeled as not belonging to the class.
- False Negative (FN) : The number of items incorrectly labeled as not belonging to the class.

		Correct Result	
		belong (positive)	not-belong (negative)
Classifier Result	belong (positive)	TP	FP
	not-belong (negative)	FN	TN

Table 2.2: TP, FP, TN, FN values for classification

For example, consider a population of 200 individuals and a disease X. Assume 70 individuals have the disease and 130 do not. A gene test is developed to identify the disease and tested on the above population. The results of the test are shown in Table 2.3.

		Actual condition	
		sick (positive)	not-sick (negative)
Gene test result	sick (positive)	68	25
	not-sick (negative)	2	105

Table 2.3: TP, FP, TN, FN values for the gene test classification algorithm.

The next step is to define two metrics, precision and recall, to evaluate the classifier.

2.5.1 Precision

Precision is defined as

$$P = \frac{TP}{FP + TP} \tag{2.4}$$

which in our classification task is the number of items correctly labeled as belonging to the class divided by the total number of elements that were classified as belonging to the class.

The precision of our gene test classifier is $\frac{68}{93} = 0.73$, which is quite low due to the large number of false positives.

2.5.2 Recall

Recall is defined as

$$R = \frac{TP}{FN + TP} \tag{2.5}$$

which is the number of items correctly labeled as belonging to the class divided by the total number of elements that actually belong to the class.

The recall of the gene test classifier is $\frac{68}{70} \approx 0.97$ which is fairly good due to the low number of false negatives.

Chapter 3

Algorithms for selecting oligos

Many studies have been done on selecting oligonucleotide probes. The focus in this studies is to design and select appropriate oligos that only hybridize with intended target sequence and not with any other sequences. One of the challenging areas in the process of oligonucleotide probe design is regarding minimization of the cross-hybridization of the probes which occurs when the probe hybridizes with an undesired target sequences. The main idea of proposed heuristic algorithms for this problem are based on preprocessing the unreliable regions to eliminate repetitive regions. The main methods and data structures that are used in the mentioned preprocessing steps are suffix trees, suffix arrays, multiple alignments and hashing algorithm using seeds. In this chapter we first present a summary of some of the main oligo design algorithms. We start by briefly introducing suffix trees and suffix array, followed by the description of those algorithms that use these data structures. In the second part, some of the major oligo design algorithms using seeds are presented.

3.1 Oligo design methods using suffix trees/suffix arrays and alignments

3.1.1 Suffix trees and suffix arrays

Since these data structures are essential to some of the algorithms we present in this chapter, we introduce them briefly here. Given a string w, a suffix of w is any string v such that there is another string u so that $w = uv$. The *suffix tree* of w is a tree whose edges are labeled by strings so that the paths from the root to its leaves spell precisely the suffixes of w. Figure 3.1 gives an example. A *generalized suffix tree* is a suffix tree that contains the suffixes of several strings.

The *suffix array* of a string w gives the lexicographically sorted list of all suffixes of w. An example is shown in Figure 3.2.

3.1.2 ProbeSelect

ProbeSelect has been introduced by Li and Stormo in [9]. Their approach for designing probes can be divided into two stages. At the beginning, a set of candidate probes is built from probes that maximize the minimum number of mismatches to every other gene in the genome. Next

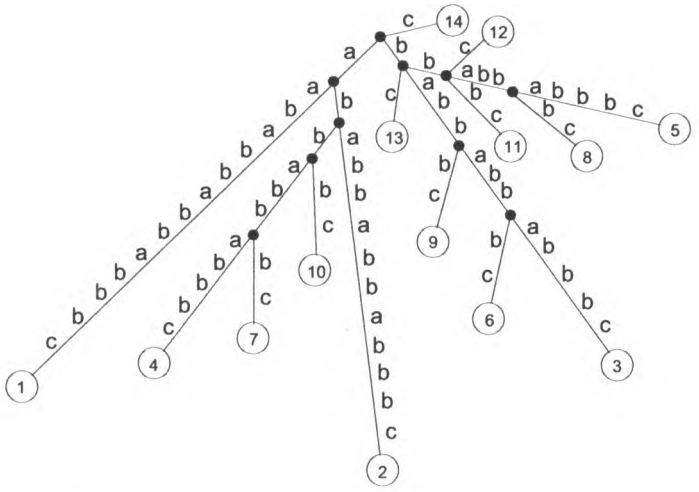

Figure 3.1: The suffix tree of the string w = aababbabbabbbc.

i	SA[i]	$\text{suff}_{\text{SA}[i]}$	LCP[i]
1	1	aababbabbabbbc	0
2	2	ababbabbabbbc	1
3	4	abbabbabbbc	2
4	7	abbabbbc	6
5	10	abbbc	3
6	3	babbabbabbbc	0
7	6	babbabbbc	7
8	9	babbbc	4
9	5	bbabbabbbc	1
10	8	bbabbbc	5
11	11	bbbc	2
12	12	bbc	2
13	13	bc	1
14	14	c	0

Figure 3.2: The suffix array of the string w = aababbabbabbbc. The second column gives the actual suffix array SA, the third gives the corresponding suffixes, whereas the last represents the size of the longest common prefixes, LCP, of adjacent suffixes; this last array often complements the suffix array in the data structure.

step is the selection of the optimal probes from the set. A probe will be selected as optimal when it hybridizes with the target sequence in an acceptable range of free energy and maximizes the difference in hybridization free energy with other non-target sequences. In order to describe their algorithm in more detail, we can divide it into seven steps.

1. Build a suffix array of the concatenation of all coding sequences of a genome
2. Build a landscape for each gene using the suffix array
   A landscape of a sequence contains the frequency of all the words of that sequence. In this case, the purpose of using the landscape is to find low frequency words in the rest of the genome which will constitute the set of unique probes.
3. Select probe candidates
   Candidate probes are chosen using their finding from testing many genes. They found that probes with the lowest frequencies at each of levels of sub-word lengths are the rarest in the rest of the genome, even with a few mismatches. Based on that they add all the probes sub-word frequencies for each probe and select ones which have the lowest values which is estimated to have the fewest approximate matches elsewhere.
4. Search for matching sequences in the whole genome
   The Myers algorithm [13] is used to find all match locations for the candidate probes in the coding regions of the genome with four or fewer mismatches for short oligos and 10 or fewer for 50 mers and 20 or fewer for 70 mers.
5. Locate match sequence positions in all genes
   By building a sorted array of all genes, a binary search of the array returns the positions of the matches in a gene.
6. Compute the free energy and melting temperature for each valid target sequence
   The free energy is calculated using the alignment of the probe sequence and its target because it is usually the lowest energy structure. However, a fast heuristic algorithm is designed to test various alternatives to the alignment in case of some exceptions.
7. Choose match sequences that have stable hybridization structures with a probe

3.1.3 PROBESEL

PROBESEL was introduced by Schliep and Kaderli in [7]. They consider only probes that are unique up to k errors and perfectly complementary to their target sequences in order to design probes that bind specifically to the target sequences. Their algorithm steps are as follows:

1. Construct a generalized suffix tree of all the target sequences and reverse complements
   Using this feature non-unique probes will be identified.
2. Remove infeasible probes by some other criteria such as:
   - Probe length: remove probes that do not satisfy the prespecified length
   - Unique probes: eliminate probes that are complementary to more than one targets

- Probe melting temperature: exclude probes which hybridize with their target in a melting temperature less than the constraint

3. Align the remaining probes with their respective target and compute the melting temperature

4. Select the temperature T and one probe for each of the target sequences
   T is a temperature threshold so that the melting temperature of all selected probes with intended target must be higher than T whereas all undesired cross-hybridizations melting temperatures must be lower than T.

The first two probe preselection steps significantly reduce the number of alignments that must be done between probes and targets.

3.1.4 ProMide

Introduced by Rahman in [17], ProMide is fast and specific for oligo lengths up to 30 bases and can handle large-scale data sets in a reasonable time. In order to design specific probes, it defines two unspecificity measures:

1. The longest common factor lcf:
   The longest substring that occurs in both an oligo and a sequence.

2. The longest common factor allowing one mismatch lcf_1:
   The longest substring that occurs in both an oligo and a sequence with at most one mismatch.

Here, the working hypothesis is that lcf can approximate the unspecificity of an oligo better than the number of mismatches between an oligo and background which the oligo specificity is checked against. A long lcf of the oligo and background shows the possibility of undesired cross-hybridization. However, this measure may be too optimistic at times. For instance, it does not exclude oligos which their lcf is low although they may have a long substring in common with background allowing one mismatch. Thus, to solve this problem the lcf_1 is also considered.

The algorithm first selects a set of candidate oligos which satisfy the specified length range or melting temperature range or both. Then, the oligos in the set will be ranked by comparing their lcf- and lcf_1- vectors which contain the length of the factors between each probe and all background sequences and incorporating some additional sequence-specific constraints. The lcf- and lcf_1- vectors are computed rapidly using a memory efficient enhanced suffix arrays.

3.2 Seed-based methods

We present below some of the main algorithms in oligo design which use seeds. Other than the ProDesign, all of them uses BLAST program.

3.2.1 OligoArray

OligoArray, developed by Rouillard et al. [20], considers several criteria to select oligonucleotide and works as follows:

- Read each sequence using a window of of size eq1ual to the length of the oligo
- Check for prohibited sequences
- Extract a substring of length equal to that of the desired oligo
- Check for the substring specificity using BLAST
- Check a set of criteria for the remaining oligos: absence of strong secondary structure, required melting temperature, and position in transcript.

OligoArray2.0, [21], is an evolved version of OligoArray. In OligoArray2.0, the computation of the specificity is based on the thermodynamics of the hybridization.

3.2.2 ArrayOligoSelector

ArrayOligoSelector, designed by Bozdech et al. [2], designs 70mer oligonucleotide probes. The program optimizes the oligo selection based on several parameters. In the first stage, four filters performs simultaneously on the sequences.

- Uniqueness filter: In order to find unique oligos which have less potential for cross-hybridization, early versions of the program used the BLASTN alignment software. However, in subsequent versions it followed by computing the theoretical binding energy between the oligo and its second best target. More stable binding with larger value shows high cross-hybridization potential. Thus, unique ones have smaller binding energies.
- User-defined filter: User can decide to avoid any kind of sequence patterns.
- Self-binding filter: Potential for forming secondary structure is detected by calculating the optimal alignment score between the probe sequence and its reverse complement. A high score indicates potential for forming secondary structure.
- Complexity filter: A lossless compression is done on the probe sequence. A high compression score shows low complexity of the sequence. Therefore, repeats are detected using this scores.

After passing these four filters, probes are subjected to %GC filter and 3′ end proximity ranking. Finally, for each target sequence a single probe candidate will be returned.

3.2.3 OligoWiz

OligoWiz, developed by Nielson et al. [14], takes into accounts some parameters for selecting probes and assign a score between 0 and 1 for each parameter. The score of an oligo will be the normalized sum of its parameter scores which is between 0 and 1 as well. The set of parameters which are considered in OligoWiz are:

- Cross-hybridization: Estimation of the degree of cross-hybridization is performed by computing the homology score for each probe using BLAST search between the sequence that the oligo originated from and other sequences.
- Melting temperature difference: It is required that all oligos can perform well under similar hybridization condition. One of these conditions is the melting temperature Tm. It is desired that there is minimal difference between melting temperature of all oligos. Thus, the oligos with the smallest melting temperature difference are chosen.
- Position within transcript: Oligos are assigned a score based on their position in the target sequence. For instance, oligos with target positions closer to the starting point of reverse sequence are preferred.
- Low-complexity filtering: In sequences from the same species some fragments are very common. Oligos which contain those specific fragments are considered as low-complexity oligos and will be assigned a low score.
- GATC-only score: Every sequence which contains characters different from A, C, G and T will be given score 0.

3.2.4 ROSO

The ROSO algorithm, due to Reymond et al. [18], consists of five major steps:

1. Removing identical genes, genes which have repeated or degenerated bases
2. Looking for potential cross-hybridizations using BLAST.
3. Eliminating oligonucleotides which form a stable secondary structure
4. Computing the melting temperature of each candidate oligonucleotide and selecting a set of oligonucleotides which minimizes the variability of the melting temperature
5. Selecting the optimal set of oligonucleotides on four criteria: GC rate, first and last bases (preferably a G or a C), repetitions, and free energy

The originality of ROSO algorithm comes from the separation of the specificity analysis using BLAST and the probe selection. This feature enables the user to separate the time-consuming BLAST step and select the probes efficiently.

3.2.5 GoArrays

GoArrays, Rimour et al. [19], has been developed to overcome the existing specificity limits for complex biological system. In order to measure specificity of an oligonucleotide, they follow the conditions of [8]:

1. The probe must not have more than 75% of similarity with a non-target sequence.
2. The probe must not have a stretch of more than 15 identical bases with a non-target sequence.

After performing *in silico* tests, they found out that in complex biological system long oligos (30 to 80 mers) do not satisfy condition 2 and show low specificity. Decreasing oligo length results in increase in specificity (20 to 30 mers) but it could lower the sensitivity. Therefore, they form the probes by concatenating two specific short sequences to gain both specificity and sensitivity. The algorithm works as follows:

- Read each sequence from the 3′ end to find the first specific substring
- Check substring specificity using BLAST
  An oligo is considered nonspecific if shows an alignment with 75% or more similarity or contains a stretch of 15 identical bases with non-target sequence
- Search for the second specific substring
- Add two substring using randomly chosen bases and check the specificity of the whole probe
- Check for melting temperature, secondary structure, prohibited sequences

3.2.6 ProDesign

The ProDesign program, designed by Feng and Tillier [4], uses spaced seed hashing for designing appropriate probes. The program can design both gene-specific and group-specific probes. Usage of spaced seeds allows the program to consider more mismatches between a probe and its targets.The program procedure consists of four stages described below.

At the beginning, for each spaced seed a hash table is built. The hash table consists of associated words of each seed where every 1 in the seed is replaced by any possible nucleobase A, C, G and T encoded by two bits, 00, 01, 10, and 11 respectively. The size of each seed's hash table is 4^w where w is the weight of the seed.

Each word in the hash table will be examined to see whether that word is specific to a group of sequences or not. The specificity of the word is obtained by counting its occurrences in the group sequences. Let $h_{i,j}$ mark the occurrence of the word x_i in the sequence j; it equals one if the word occurs in the sequence and zero otherwise. Furthermore, let $H_{i,k}$ mark the occurrence of the word x_i in the group k. If the value of $h_{i,j}$ is 1 for all the sequences in the group, then $H_{i,k}$ is 1. For each word in the hash table a list of groups with $H_{i,k} = 1$ is built. A word is specific to a group if $H_{i,k} = 1$ only for that group.

w_1: ACCTA
w_2: TACGG
sequence: ...TCATTACCTACGGAGACC...

Figure 3.3: The words w_1 and w_2 two overlapping group specific words. The new group specific word, in red, is the overlapping substring of two words.

w_1: ACCTA
w_2: TACGG
sequence: ...ACCCTACCTACTACGGAGATG...

Figure 3.4: The words w_1 and w_2 are two group specific words separated by a gap of length 1. The new group specific word, in red, is formed by joining them together with the gap.

After this stage, some groups have one or even more specific words and some groups may not have a word. The latter groups are considered later to be reclustered into other groups.

The specific words for each group are taken into consideration for finding probe candidates for the group. The user can specify the length of the required probes. If the group-specific word satisfies the length threshold it will be returned as the group probe. Otherwise, two or more words need to be joined together to provide the required length. In this case, ProDesign starts with selecting a random sequence in each group. Then, the position of the specific word is found by scanning the sequence. The scan is extended forward to find the second specific word. In order to be joined, two words must be either overlapping or the gaps between them must be less than 3 bases. The joined word starts from the first character of first word until the last character of the second word. Two examples of possible joined words are given in Figures 3.3 and 3.4.

The new word is then searched for in all the other sequences of the group. If found in all of them, it will be selected as a probe candidate for the group. All the probe candidates will be checked in order not to have low complexity and satisfy the melting temperature and %GC content requirements.

In the paper, they compare the ProDesign with some other software programs for oligo design and their results indicate that ProDesign have more flexibility and speed than other programs. In addition, the software coverage was more than others which means the proportion of target sequences specifically identified by the program. The greater coverage is due to the use of space seeds.

3.2.7 Seed evaluation in oligo design

The seed based algorithms are widely used in oligonucleotide design because they are fast and highly sensitive; they can work well in the presence of mismatches. A crucial aspect of all these algorithms is the seeds they use. Possibilities are many: contiguous, transition-constrained, spaced, multiple spaced, etc. Therefore, an algorithm for thoroughly evaluating all these variants is very much needed. We shall discuss the only attempt to build such a procedure in the next section, reveal its shortcomings, and propose a much better approach for evaluating seeds in oligonucleotide design.

Chapter 4

Multiple spaced seeds for oligonucleotide design

This section and the next one contain the main contribution of this thesis. We shall introduce a proper framework for comparing various types of seeds for oligonucleotide design. To start with we describe the current approach, due to [3] and explain why this approach in flawed. After that we explain why multiple spaced seeds are expected to be a good candidate. We introduce a fast algorithm for computing those and then use it to compute a variety of multiple spaced seeds. We include all these seeds and plot their sensitivity. At the end of the chapter the new approach for comparing seeds is described.

4.1 Comparison setup

We describe in this section the general setup used to compare seeds. A testing data set is created by first generating a number of (main) sequences and then generating a set of (secondary) sequences for each; of these, some are similar with the main sequences (and we shall call those *oligos*) while others are not (and will be called *non-oligos*). A small (unrealistic) example is given in Table 4.1 where one main sequence is shown together with its associated secondary sequences. (The way this data set is constructed in reality is discussed in the experiments chapter.)

The comparison of the seeds is based on the idea that they should hit oligos and should not hit non-oligos. Determining whether a seed hits a secondary sequence or not is done based on the use of seed hashes as described before: a sequence is hit if a common seed hash (according to the seed under consideration) is found between this sequence and the main one.

4.2 Efficient discriminability

To the best of our knowledge, there exists only one study, due to Chung and Park [3], aiming at comparing various seeds for oligonucleotide design. In this study, they propose three measures for evaluating seeds:

main seq.	CGGGGCAACCGCGACAACTT	
secondary sequences	CGGGGCAACCGCGACAACTT	oligo
	CGGGGCAACCGCGACGCCTT	oligo
	CGGGGCTACCGCGAAAGCTT	non-oligo
	CGGGTCCACCGCGCCAACAT	non-oligo
	TCGCGCAAACGCGACATCTC	non-oligo
	CCGGGATACCCCGACACCCG	non-oligo
	CCGGGGTAGCTCGACTCCTG	non-oligo
	CCGCGCGACATCTACAGTGC	non-oligo
	GGGAGTGCCTAGAACAGCTC	non-oligo
	CATAGGACAGGCCACGTATA	non-oligo
	CGTGCACAGGAAGTTGGCGG	non-oligo

Table 4.1: One main sequence and its associated secondary sequences, marked as oligos or non-oligos.

1. Discriminability
2. Efficiency
3. Efficient discriminability

with the last one being the most important, aiming to combine the first two. Their definitions are described below.

The discriminability of a seed is defined using the well known measures from statistics: precision P and recall (or, sensitivity) R (which we introduced in Chapter 1, equations 2.4, 2.5) of that seed in the oligo design.

Precisely, a seed aims at detecting a similarity and for that purpose *seed hashes* are constructed, where each `1` is replaced by any possible nucleobase `A`,`C`,`G`,`T`, and then searched for in the given sequences. The values of TP, FP, TN and FN are therefore defined as follows.

TP *for precision* : The number of seed hashes hitting oligos.

TP *for recall*: The number of oligos containing at least a matched seed hash.

FP: The number of seed hashes hitting non-oligos.

TN: The number of seed hashes does not hit any non-oligo.

FN: The number of oligos that do not contain any matched seed hash.

The *discriminability* is the well known F-score measure of a test's accuracy. It is defined as the weighted harmonic mean of precision and recall:

$$F_\alpha = \frac{(1+\alpha^2)PR}{\alpha^2 P + R}. \tag{4.1}$$

The parameter α is a non-negative parameter. By increasing it over 1, the weight of precision becomes higher than that of recall and makes F_α sensitive to false positives. By decreasing it, discriminability will be sensitive to false negatives. The F_α maximum value is 1 which occurs when precision and recall are 1. In this case, there is not any false negative or false positive.

In practice, the basic F-score is used, since there is no reason for choosing a particular values of α different from 1:

$$F_1 = 2\frac{PR}{P+R}. \tag{4.2}$$

However, the choice of α is a minor point here. Any F_α-score is inappropriately defined, since the definition of TP changes between P to R. Essentially, the precision and recall of different tests are used in the definition of the F-score, thus making it meaningless from statistical point of view.

Efficiency is defined as a normalization of the average rate A of the number of seed hashes in an oligo.

$$E_\beta = \frac{1}{1+\beta \log A} \tag{4.3}$$

The weight β ranges from 0 to 1 and the weight of the rate A increases and decreases with β. In practice $\beta = 1$ is used however, an immediate question raises: Why normalize it this way since normalization can be done in a variety of ways? The significance of this choice becomes apparent in the next definition, that of the main measure, efficient discriminability, below.

Efficient discriminability, the central measure in the study is obtained by multiplying the discriminability and the efficiency:

$$G_{\alpha\beta} = F_\alpha E_\beta. \tag{4.4}$$

The maximum value of $G_{\alpha\beta}$ is obtained when F_α and E_β have their maximum values. The value used in practice is obtained for $\alpha = \beta = 1$, that is:

$$G = G_{1,1} = F_1 E_1 = 2\frac{PR}{P+R}\frac{1}{1+\log A}. \tag{4.5}$$

Besides the flaws in the definitions of discriminability and efficiency, now the two measures are multiplied together, without much consideration given to what is obtained. Mixing the two measures gives results that are impossible to interpret. And that would remain true even in the case when discriminability would be correctly defined.

4.3 Different normalization

We shall make our point very clear by using several examples. We present the values of discriminability, efficiency and efficient discriminability for a number of optimal contiguous, spaced and transition-constrained seeds, where the only variable is the normalization used

	Efficient Discriminability			Efficiency			Discriminability		
Weight	Cont	Spaced	Trans	Cont	Spaced	Trans	Cont	Spaced	Trans
10	0.1915	0.2012	0.2034	0.2000	0.2117	0.2133	0.9574	0.9501	0.9533
11	0.1971	0.2110	0.2127	0.2054	0.2212	0.2229	**0.9597**	0.9538	0.9545
12	0.2008	0.2169	0.2177	0.2105	0.2265	0.2278	0.9538	0.9575	0.9557
13	0.2023	0.2248	0.2251	0.2152	0.2362	0.2375	0.9398	0.9516	0.9479
14	0.2018	0.2283	0.2280	0.2196	0.2431	0.2442	0.9189	0.9389	0.9336
15	0.2001	0.2329	0.2317	0.2240	0.2528	0.2536	0.8935	0.9212	0.9134
16	0.1897	0.2308	0.2289	0.2251	0.2562	0.2566	0.8431	0.9009	0.8921

Table 4.2: Evaluation results for efficiency normalized as $\frac{1}{1+\log_2(A)}$, where A is the average rate of seed hashes in an oligo.

for computing efficiency. Various bases of the logarithm are used in Tables 4.2, 4.3, and 4.4 whereas in Table 4.5 the normalization of A is simply done by considering

$$E = \frac{1}{A}\,. \tag{4.6}$$

The results are, expectedly, very different. In all tables, the highest efficient discriminability value in each column is shown in a box. When changing the base of the logarithm from 2 to 10, Tables 4.2 and 4.3, the contiguous seed with the highest efficient discriminability changes. For a larger base, see Table 4.4, the seeds with the highest efficient discriminability change in all three categories, contiguous, spaced and transition-constrained.

When the normalization is completely changed to the inverse of the average rate A of the number of seed hashes in oligos, Table 4.5, the entire comparison collapses, since not only the seeds with the highest efficient discriminability change, but the efficient discriminability keeps increasing with the weight of the seeds. (It peaks at weight 19 for the transition-constrained seeds but keeps growing past weight 20 for the spaced ones.)

It should be very clear by now that the entire framework of [3] for evaluating seeds for oligonucleotide design is simply not working.

In addition there are problems with their data sets. They use two data sets, one generated using their program and another consisting of real biological data. Unfortunately, the biological data is not available and the authors did not respond to our e-mails. Therefore, we cannot discuss the real data at all. The synthetic data is available because their software for producing it is public. We have used it to generate the data for the experiments in the last chapter.

The conclusions of the seed evaluation results in [3] are:

- Spaced seeds are generally preferred to the other seeds.
- Transition seeds works slightly lower than spaced ones.
- Spaced seed or transition seed of weight 15 ~ 18 are recommended for 50mer oligos.
- Multiple seeds are not good at oligo design.

	Efficient Discriminability			Efficiency			Discriminability		
Weight	Cont	Spaced	Trans	Cont	Spaced	Trans	Cont	Spaced	Trans
10	0.4344	0.4481	0.4518	0.4537	0.4716	0.4739	0.9574	0.9501	0.9533
11	0.4434	0.4631	0.4656	0.4620	0.4855	0.4879	**0.9597**	0.9538	0.9545
12	0.4481	0.4722	0.4731	0.4697	0.4932	0.4950	0.9538	0.9575	0.9557
13	0.4480	0.4823	0.4820	0.4767	0.5068	0.5085	0.9398	0.9516	0.9479
14	0.4440	0.4847	0.4833	0.4832	0.5162	0.5177	0.9189	0.9389	0.9336
15	0.4373	0.4875	0.4844	0.4894	0.5292	0.5303	0.8935	0.9212	0.9134
16	0.4140	0.4807	0.4766	0.4910	0.5336	0.5342	0.8431	0.9009	0.8921

Table 4.3: Evaluation results for efficiency normalized as $\frac{1}{1+\log_{10}(A)}$, where A is the average rate of seed hashes in an oligo.

	Efficient Discriminability			Efficiency			Discriminability		
Weight	Cont	Spaced	Trans	Cont	Spaced	Trans	Cont	Spaced	Trans
10	0.5466	0.5591	0.5631	0.5709	0.5884	0.5907	0.9574	0.9501	0.9533
11	0.5558	0.5741	0.5766	0.5791	0.6019	0.6041	**0.9597**	0.9538	0.9545
12	0.5595	0.5833	0.5839	0.5866	0.6092	0.6109	0.9538	0.9575	0.9557
13	0.5577	0.5920	0.5912	0.5934	0.6221	0.6237	0.9398	0.9516	0.9479
14	0.5510	0.5924	0.5903	0.5996	0.6309	0.6323	0.9189	0.9389	0.9336
15	0.5411	0.5923	0.5882	0.6056	0.6430	0.6440	0.8935	0.9212	0.9134
16	0.5119	0.5829	0.5777	0.6072	0.6470	0.6475	0.8431	0.9009	0.8921

Table 4.4: Evaluation results for efficiency normalized as $\frac{1}{1+\log_{40}(A)}$, where A is the average rate of seed hashes in an oligo.

	Efficient Discriminability			Efficiency			Discriminability		
Weight	Cont	Spaced	Trans	Cont	Spaced	Trans	Cont	Spaced	Trans
10	0.0599	0.0720	0.0740	0.0625	0.0758	0.0776	0.9574	0.9501	0.9533
11	0.0657	0.0831	0.0851	0.0685	0.0872	0.0892	**0.9597**	0.9538	0.9545
12	0.0709	0.0898	0.0912	0.0743	0.0938	0.0954	0.9538	0.9575	0.9557
13	0.0751	0.1012	0.1024	0.0799	0.1064	0.1080	0.9398	0.9516	0.9479
14	0.0783	0.1085	0.1093	0.0852	0.1156	0.1171	0.9189	0.9389	0.9336
15	0.0809	0.1188	0.1188	0.0905	0.1289	0.1301	0.8935	0.9212	0.9134
16	0.0775	0.1204	0.1198	0.0919	0.1336	0.1343	0.8431	0.9009	0.8921

Table 4.5: Evaluation results for efficiency normalized as $\frac{1}{A}$, where A is the average rate of seed hashes in an oligo.

However, these conclusions are claimed to have been made on both the real and synthetic data but [3] gives only the results for the real data. The real data itself being unavailable, we could not check it but we generated the synthetic data and their own conclusions do not fit the results. The best indication of this fact is that the highest discriminability values in the Tables 4.2,4.3, 4.4 and Table 4.5 is obtained for contiguous seeds, in contradiction with the common knowledge that both spaced and transition-constrained seed are better. (The claimed results of [3] for the real data confirm the common knowledge.)

4.4 A good candidate: multiple spaced seeds

Searching exact occurrences of patterns in texts can be done in linear time. However, when searching a small pattern in a large database, a time complexity linear in the size of the database is too slow. Linear time in the length of the pattern is preferred. This is achieved by constructing an *index* on the database, such as a suffix tree or suffix array. Such an index takes linear time in terms of the database size to compute but the advantage is that it will be used many times, providing very fast searching times.

In biological applications, the search for patterns is never exact but approximate. Our biological mechanisms are built to allow for errors and so have to be our search algorithms in order to provide meaningful results. However, there are no approximate indexes which are time and space efficient as in the exact case. Many techniques have been developed to deal with the situation and the best of all is building indexes for multiple spaced seeds, that is, tables storing their seed hashes. It is therefore natural to consider them as good candidates for oligonucleotide design, where approximate search is essential. We should also notice that the conclusion of [3] that the multiple spaced seeds are not good for oligo design refers actually to some less powerful variants, called vector seeds and BLAT seeds. Real multiple spaced seeds were not considered at all by [3].

We go next into details as to why multiple spaced seeds are good at approximate string searching. Consider a single seed s of length ℓ and weight w (recall that the weight is the number of 1's). Consider also a random region R of length N and similarity level p, as done in the definition of sensitivity. The expected number of hits s has in R is $(N-\ell+1)p^w$, since there are $N-\ell+1$ places where s can hit and each has probability p^w. Now, if we increase the weight of a the seed by 1, then the expected number of hits becomes essentially a fraction $1/p$ of the old one. Assuming that the four bases A, C, G, T appear with equal probability, that means that the number of expected hits is only a quarter of the previous one. Less hits means less wasted ones, that is, less false positives and therefore increased specificity. However, increasing the weight of a seed also decreases the true positives, and therefore the sensitivity. In order to increase both, we can increase not only the weight but also the number of seeds. It turns out that doubling the number of seeds provides slightly better sensitivity. But doubling the number of seeds only increases the expected hits by a factor of two whereas increasing the weight by one reduces it to a quarter. Essentially, this is the main reason why multiple spaced seeds are so good.

As an example, in Figure 4.1, we plot the sensitivity values for the optimal spaced seed of weight $w = 11$, a set of two (good) spaced seeds of weight $w = 12$, a set of four spaced seeds with $w = 13$ and a set of eight spaced seeds with $w = 14$. Various levels of similarity are used

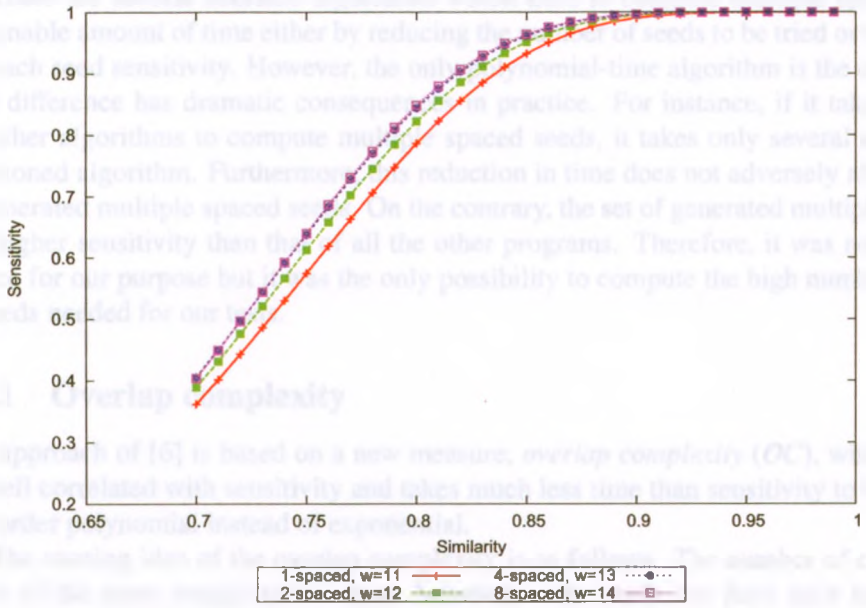

Figure 4.1: Sensitivity plots for 4 sets of spaced seeds: 1 spaced seed of weight 11, 2 spaced seeds of weight 12, 4 spaced seeds of weight 13, and 8 spaced seeds of weight 14. The sensitivity curves are higher for larger number of seeds. The last two curves are essentially identical.

to give a complete picture. As it can be seen from the graphics, the sensitivities increase with the size of the seed set, even if the weights increase as well.

One should be aware however, that more memory is required for a higher number of seeds in order to store more hash tables and this enforces an upper bound on the number of seeds that can be used.

4.5 A fast algorithm for computing multiple spaced seeds

We need good multiple seeds for our purpose but computing good multiple spaced seeds is a hard problem. There are several algorithms which give good multiple spaced seeds but the only fast approach available is due to [6]. In addition to being fast, the algorithm computes the most sensitive multiple spaced seeds.

In this section, we will fully describe the fast algorithm for computing multiple spaced seeds by [6]. We start by explaining the reasons that make computing optimal multiple spaced seeds by exhaustive search infeasible. There are two exponential steps in this process:

1. There are exponentially many seeds need to be tried.

2. Time complexity of computing each seed sensitivity is exponential as well.

There are several heuristic algorithms which tried to compute multiple spaced seeds in a reasonable amount of time either by reducing the number of seeds to be tried or by approximating each seed sensitivity. However, the only polynomial-time algorithm is the one of [6]. This time difference has dramatic consequences in practice. For instance, if it takes several days for other algorithms to compute multiple spaced seeds, it takes only several seconds for the mentioned algorithm. Furthermore, this reduction in time does not adversely affect the quality of generated multiple spaced seeds. On the contrary, the set of generated multiple spaced seeds has higher sensitivity than that of all the other programs. Therefore, it was not only the best choice for our purpose but it was the only possibility to compute the high number of large sets of seeds needed for our tests.

4.5.1 Overlap complexity

The approach of [6] is based on a new measure, *overlap complexity* (*OC*), which is shown to be well correlated with sensitivity and takes much less time than sensitivity to be computed; a low order polynomial instead of exponential.

The starting idea of the overlap complexity is as follows. The number of expected hits of seeds of the same weight is the same, however, some seeds can have their hits overlapping more than others. Consider two very simple seeds: 1111 and $1*11**1$. If 1111 has a hit then it will have another one shifted by one position if the next position is a match. This happens with probability p (the similarity level). On the other hand, the seed $1*11**1$ requires three new matches in order to have the same additional hit. That happens with probability p^3, which is considerably lower. In general the difference is even bigger due to higher weight.

What this means is that the hits of the seed that overlap less will be more uniformly distributed and hence able to hit more alignments. That means, higher sensitivity. In other words, high sensitivity is obtained by low number of overlapping hits. The complexity measure introduced to replace sensitivity, overlap complexity, *OC*, is therefore defined as follows.

Consider two seeds s_1 and s_2. The overlap complexity of the two seeds, denoted $OC(s_1, s_2)$, is defined as:

$$OC(s_1, s_2) = \sum_{i=1-|s_2|}^{|s_1|-1} 2^{\sigma[i]}, \tag{4.7}$$

where $\sigma[i]$ is the number of pairs of 1's aligned together when a copy of s_2 which is shifted by i positions, is aligned against s_1. Precisely, to define $\sigma[i]$, two variables t_1 and t_2 are defined as

$$\begin{aligned} t_1 &= *^{|s_2|-1} s_1 *^{|s_2|-1}, \\ t_{2,i} &= *^{|s_2|-1+i} s_2 *^{|s_1|-i-1}, \text{ for } 1-|s_2| \le i \le |s_1|-1, \end{aligned} \tag{4.8}$$

and then

$$\sigma[i] = \text{card}\,\{j \mid 1 \le j \le |s_1| + 2|s_2| - 2, t_1[j] = t_{2,i}[j] = 1\}. \tag{4.9}$$

For example, let $s_1 = 11*1$ and $s_2 = 1*1$. In this case, $t_1 = **11*1**$ and $t_{2,i} = *^{2+i}1*1*^{3-i}$ for $-2 \le i \le 3$. The alignments of t_1 against $t_{2,i}$ for different values of i in the range are shown in Figure 4.2. As shown in the figure, the value of *OC* is 13. Note that the measure has symmetry, that is, $OC(s_1, s_2) = OC(s_2, s_1)$, for any two given seeds s_1 and s_2.

For a multiple spaced seed $S = \{s_1, s_2, ..., s_k\}$, the overlap complexity is defined as sum of the overlap complexities of each two seeds in the set.

									$\sigma[i]$
t_1 :	*	*	1	1	*	1	*	*	
$t_{2,-2}$:	1	*	1	*	*	*	*	*	1
$t_{2,-1}$:	*	1	*	1	*	*	*	*	1
$t_{2,0}$:	*	*	1	*	1	*	*	*	1
$t_{2,1}$:	*	*	*	1	*	1	*	*	2
$t_{2,2}$:	*	*	*	*	1	*	1	*	0
$t_{2,3}$:	*	*	*	*	*	1	*	1	1

Figure 4.2: An example of the two seeds overlap complexity computation: $OC(11*1, 1*1) = \sum_{i=-2}^{3} 2^{\sigma[i]} = 13$.

It is shown in [6] that the overlap complexity is experimentally very well correlated with sensitivity for single seeds. However, the same comparison for multiple spaced seeds cannot be done since exhaustive search is infeasible in this case.

4.5.2 The polynomial-time algorithm

The heuristic algorithm based on overlap complexity described above deals with the two exponential steps as follows:

1. Sensitivity: Instead of computing the exponential-time sensitivity the polynomial-time overlap complexity is computed.

2. Number of seeds: When considering many seeds, the possible choices for their lengths are exponentially many. This is dealt with by guessing a good length set. Second, once the lengths are fixed, there are exponentially many possible seeds with the given lengths and weight. To avoid this, the algorithm starts with some fixed seeds and repeatedly swap a 1 with a * as long as the overlap complexity improves. In addition, it has a greedy approach in selecting each swap. The swap which improves the results the most will be chosen. It also set a boundary for the number of swaps in each seed which is the weight of the seeds.

The pseudocode of the algorithm is given in Figure 4.3. Note that 1 flipped is $*$ and vice versa. The procedure flip(s, i, j) flips the letters in positions i and j. For instance, flip$(1*111, 2, 4) = 111*1$. The parameter m and M are minimum and maximum bounds for the length of generated seeds. The parameter, m equals round-up$(\frac{4w}{3})$ and parameter M is set to 25 in the original algorithm. However, the algorithm gives the option to set the seed lengths range experimentally. For instance, if the needed weight for the seeds is 19, we can set $m = 21$ and $M = 32$. Then, the algorithm returns the set of seeds with given weight which has the lowest overlap complexity. In addition, we used the heuristic value $h = 1.25$ for producing seeds.

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

4.6 Computed multiple spaced seeds

Using the MULTIPLESEEDS algorithm, we generate multiple spaced seeds with 2, 4, and 8 seeds for all weights between 7 and 20. The seeds are given in Tables 4.9, 4.10, 4.11 and 4.12.

They will be tested in the next chapter against single seeds of three types: contiguous, transition-constrained and single spaced. As we mentioned before, the single seeds can be computed by exhaustive search and therefore they are the same as those in [3]. They are given in Tables 4.6, 4.7, 4.8.

Seed	Weight
1111111	7
11111111	8
111111111	9
1111111111	10
11111111111	11
111111111111	12
1111111111111	13
11111111111111	14
111111111111111	15
1111111111111111	16
11111111111111111	17
111111111111111111	18
1111111111111111111	19
11111111111111111111	20

Table 4.6: Contiguous seeds weight 7 to 20

Seed	Weight
1*1@*1*1@**11	7
11*1@*1*1@**11	8
11*1@*1*1@**111	9
11*11**@11*1*1@1	10
111*1@*1*1**1@*111	11
11@*1*11*1**11@111	12
11@*1@11**11**1*1111	13
111*1@1**1*11@*1*1111	14
1111**1@1*1*11**@1*1111	15
111@*11**11@1*1*11*1111	16
1@11*1*1*111**11*11@1111	17
1111*11**1@1*1*1@11*11111	18
111@*1*1*111@*11*11*11*1111	19
1@1*11@*1*11@*1*1*11@11111*11	20

Table 4.7: Transition seeds weight 7 to 20

Seed	Weight
1*11*1*1***11	7
11*11*1*1***11	8
11*11*1*1***111	9
11*11***11*1*111	10
111*1**1*1**11*111	11
111*1*11*1**11*111	12
111*1*11**11**1*1111	13
111*111**1*11**1*1111	14
1111**1*1*1*11**11*1111	15
1111*11**11*1*1*11*1111	16
1111*1*1*111**11*11*1111	17
1111*11**111*1*1*11*11111	18
1111*1*1*111**11*11*11*1111	19
111*11**1*111*1*1*11*11111*11	20

Table 4.8: Spaced seeds weight 7 to 20

$w = 7$	$w = 8$
11*1*1*111	111**11*1*11
111****1**1*11	111*1****1***1*11
$w = 9$	$w = 10$
111*11*1**111	1111*11*1*111
111*1****1***1*111	111*1***1**1**1111
$w = 11$	$w = 12$
111*1*11*11*111	111*1*11*11*1111
1111*1**1***1*1*111	1111*1***1**1***11*111
$w = 13$	$w = 14$
1111*11*111*1111	111*11*1*111*11111
1111*11****1*1**11*111	1111*1**11***1**1*11111
$w = 15$	$w = 16$
11111*111*1*11*1111	1111*1111*111*11111
1111*1*1**11***1**11*1111	111111*1***1*1**1**111*111
$w = 17$	$w = 18$
1111*111*1*111*11*1111	11111*1111*11*111*1111
1111*1*1**11**11***1*111111	111111**11**1*11**1*1*11111
$w = 19$	$w = 20$
11111*11111*11*111*1111	1111*111*11*11111*111111
11111*11**11***1*11**1*1*11111	111111*1*11*11*1*1**11**11111

Table 4.9: 2-spaced seeds weight 7 to 20

$w = 7$	$w = 8$
11*1*1*111	11*11**1111
111***1**1*11	111***1*1*1*11
11*1**1***1***11	11*1**1****1**111
11**1***1****1*11	11*1*1******1****1*11
$w = 9$	$w = 10$
11*11*1*1111	111**1*11*1*111
111*1***1**1*111	11*111***111*11
111*1**1****1***1*11	111*1*1***1**1**111
111***1***1****1***1*11	111*1**1*******1***1*111
$w = 11$	$w = 12$
111*11*11*1*111	1111*11*111*111
111*1**1*1***1**1111	1111***1*1*11**1111
11*1*1***111***11*11	111*1**1**1***1**1*1111
1111**1******1**1***1*111	111*1**1****1*1***1*1**111
$w = 13$	$w = 14$
1111*111*11*1111	111*11*11*111*1111
1111*1**1*1*11**1111	11111*1**1*1**11*1*111
111*1*1**11****111*111	1111*1*1****11***1**11*111
1111**1*1***1***1***1*1111	1111**11**1****1*1*1**1111
$w = 15$	$w = 16$
1111*111*1*11*11*111	1111*111*1111*11111
111*1**1*1*11***111*1111	111*111**1**1*1*11**11111
1111**11**1**11*1***11111	11111**1*111*1***1*11*111
1111**1*1***1*1**1**1***11111	11111**1*1****11**1**1*1*1111
$w = 17$	$w = 18$
11111*111*11111*1111	111*11111*111*11*11111
11111*11**1*11*1*1**11*111	111111**1**1111**1*11*1111
1111*1*1*11***11**11*11111	1111*1*1111****111*111*111
11111**11**1***1*1***1*1**11111	11111**1*1**11***1**1*1*1*11111
$w = 19$	$w = 20$
1111111*1111*111*11111	111111*1111*11111*11111
11111*1*11**1*111*1**111111	11111*1*111*1***11*11*111111
1111*111**1*11***11*11*11111	1111*111***11*1111**1*1*11111
11111**1*111****1*1*1**1*11*1111	11111*1*11*1*1**1***1**111**11111

Table 4.10: 4-spaced seeds weight 7 to 20

$w = 7$	$w = 8$
111*1*111	111*1*1111
111*1**1*11	111*11**1*11
11***1*11*11	11*1**11**111
1*11***11**11	111**1***11*11
11**1*1***111	11*1***1*1**111
11*1******1**111	11*1*1***1***1*11
11**1*****1*1**11	11*1*1****1***11*1
11*1****1*****1*11	11*1****1*****1*111
$w = 9$	$w = 10$
11*111**1111	111*111*1111
1111**1*1*1*11	1111*1*11**111
111*1*1**1**111	11*1*11**11*111
11**1*11**1**111	1111**1*1*1*1*11
11*1**11****1*111	111*11**1****1111
111**1****1*1**1*11	1111***1*1**1**1*11
11*1***1***1***1*111	111*1***1**1***1*111
11*1*1****1*****11*11	11*1**1****1*1**1*111
$w = 11$	$w = 12$
1111111*1*111	111*11111*1111
111*1**111*1111	111*111*11*1*111
111*1*11*11**111	11111***1*111*111
111*111*1****1111	11*1*1111***11*111
111*1**1**11*1*111	11*11*1**11***11111
111*1**1****1*11*111	1111*1***1***11*1*111
11*1**1***11***1*1111	1111*1****1*1**1**1111
111*11****1**1***1*111	1111**1*1*****1**11*111
$w = 13$	$w = 14$
11111*11111*111	1111*1111*111111
1111*111**1*11111	111*11111*11**1111
1111*1**111*11*111	1111**11*11*1*11111
1111**11*1*11**1111	11111*1**1*1*111*111
11*111**1**11*1*1111	111*11*11***11*1*1111
1111**1***11*1*1*1*111	11111**1*1*1*1**1**1111
11111***1*1***1*1**1111	1111*1*1**11*****111*111
1111*11****1**1***11*111	1111*1**1***1**11**1*1111

Table 4.11: 8-spaced seeds weight 7 to 14

$w = 15$	$w = 16$
1111111*111*11111	111111*111111*1111
111**11111*1*111111	111*11111*11*1*11111
1111*111**1111*1*111	1111*1**1111*111*1111
11111*1*11*1**11*1111	11111*1*11**111*11*111
11*111*1***1*1111*1111	1111*1111**11**1*11*111
11111**1*1**11***111*111	1111*11**11****111*1*1111
11111**11*1****1*1**11111	1111*1*1**1*111***1**11111
1111*1*1***11*1**1**1*1111	111111***1*1*1**1***11*1111
$w = 17$	$w = 18$
11111*111*1111*11111	111111*11111111*1111
1111*11*11111**111*111	111*1111111*11*1*11111
1111*11111***11*11*1111	1111*111*1*11**11111111
1111*1*1*11*11*1**111111	11111*1**1*1111*11*11111
111*111**1***1*11111*1111	111111*11*11***111*1*1111
11111***111*11*1*1**1**1111	1111*1***111**11*11**111111
1111*11**1*1**11***1*1*11111	1111*11*111***1*1**11**11111
11111**11***1*1***11*1*1*1111	11111**11**1*111****1*11*1111
$w = 19$	$w = 20$
111111*111*1111111111	1111111*1111111*111111
111*111111111*1*11*1111	111*11111*1111*111*11111
1111*111*11*11111**11111	111111*11*11*111*111*1111
11111*1*11*11**111111*111	11111*1**111*1*1111*111111
111111**1*1*1111**11*11111	11111**111*11111**1*11*1111
11111**111***11*11*1*11*1111	1111*111*11***1**1111*1*11111
1111*111*1*111***1*1***111111	11111*11*1***1111**1**111*1111
11111*1*1**11**1*1***111*11111	1111*111**11*1*1***1*1**1111111

4.7 Sensitivity plots

The sensitivity values for the different types of single and multiple spaced seeds which we use is essential. We have used the implementation of [6] to compute the sensitivity of all seeds except for the transition-constrained ones. The sensitivity of transition seeds was computed using Iedera [15] that is a seed design tool and can also measure sensitivity of existing seeds. The sensitivity values were computed for a wide range of similarity levels, $0.7, 0.71, ..., 0.99$. We have chosen to plot these values in such a way that that the values can be compared also visually in Figures 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9.

It is visible as well in the plots that the relation emphasized in Figure 4.1, that is, sensitivity does not decrease when simultaneously increasing the weight by one and doubling the number of seeds, is valid for all cases.

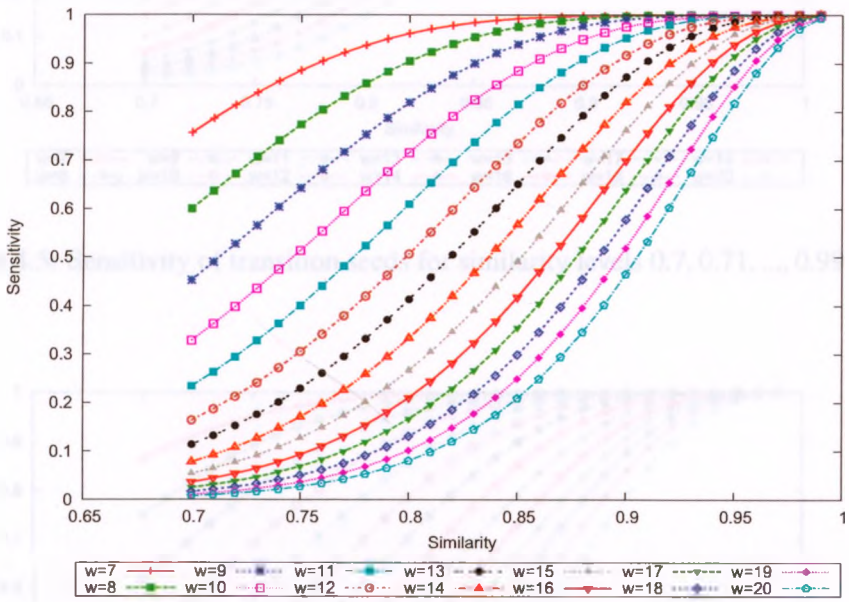

Figure 4.4: Sensitivity of contiguous seeds for similarity levels $0.7, 0.71, ..., 0.99$

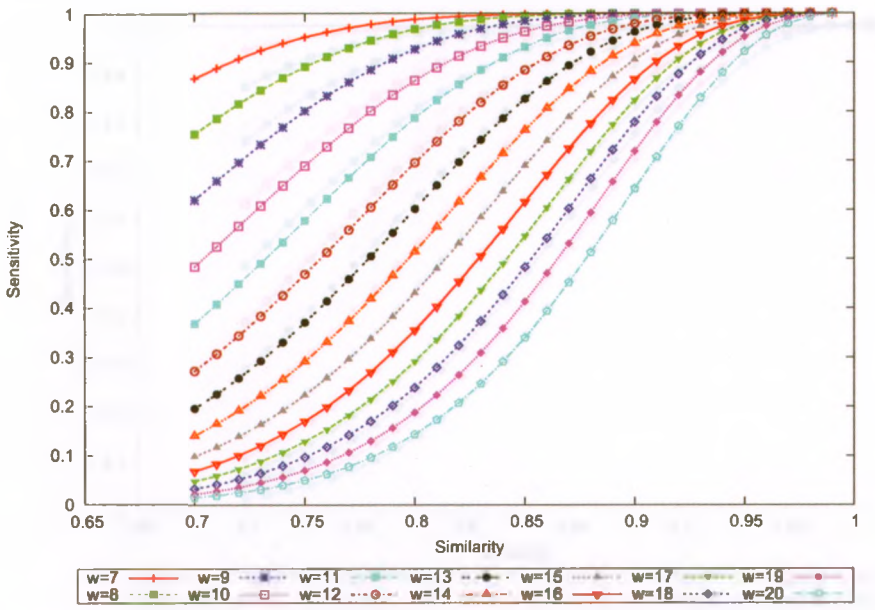

Figure 4.5: Sensitivity of transition seeds for similarity levels 0.7, 0.71, ..., 0.99

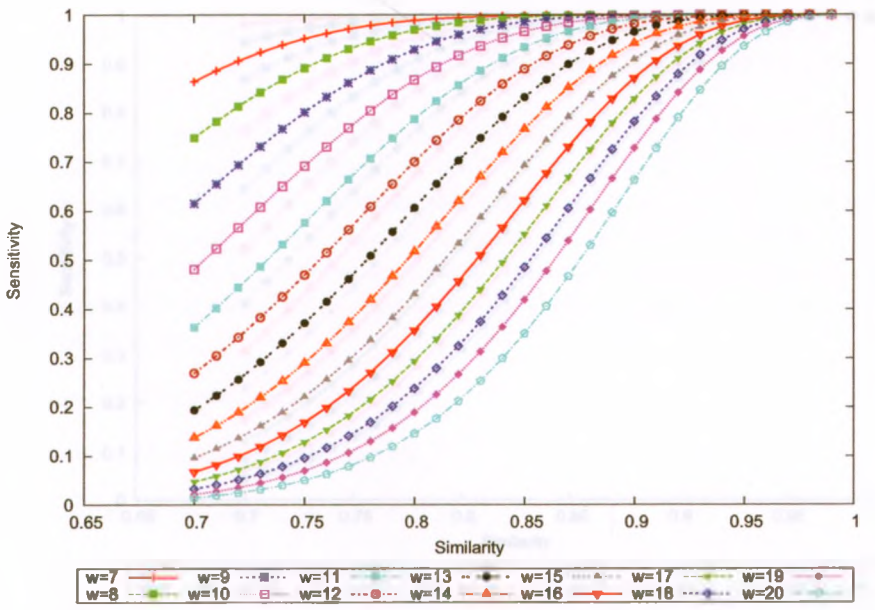

Figure 4.6: Sensitivity of 1-spaced seeds for similarity levels 0.7, 0.71, ..., 0.99

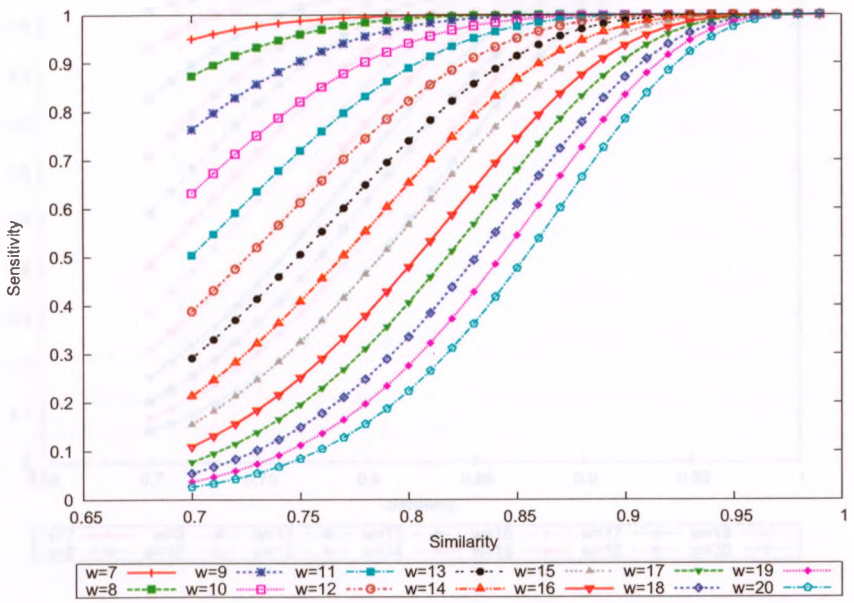

Figure 4.7: Sensitivity of 2-spaced seeds for similarity levels 0.7, 0.71, ..., 0.99

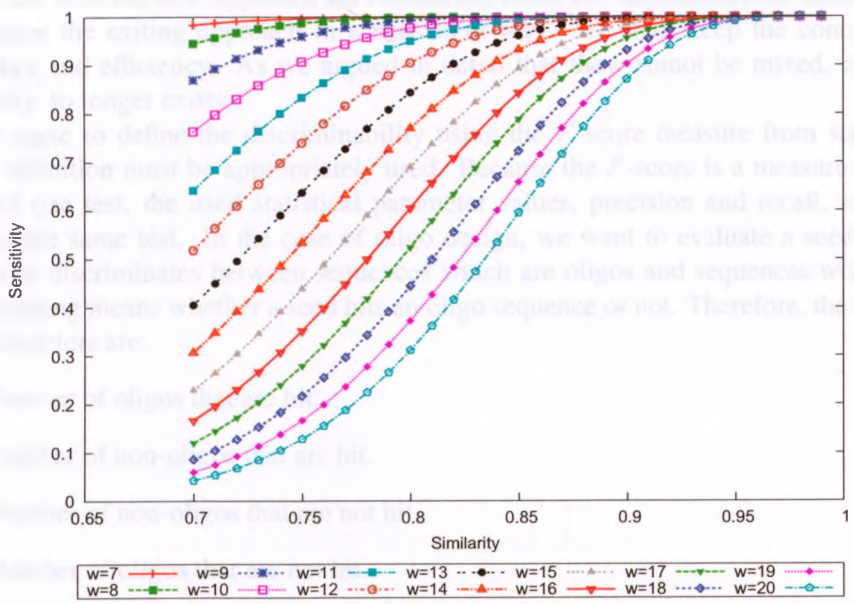

Figure 4.8: Sensitivity of 4-spaced seeds for similarity levels 0.7, 0.71, ..., 0.99

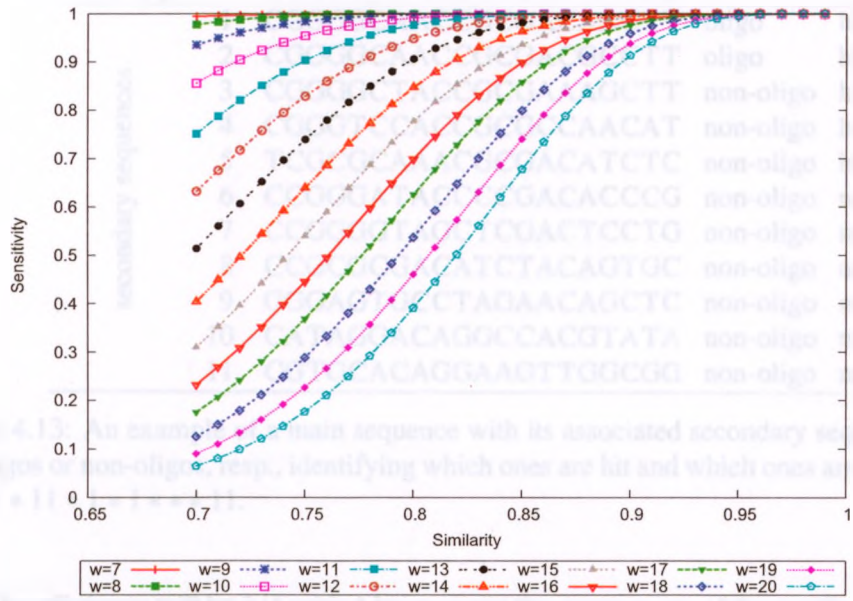

Figure 4.9: Sensitivity of 8-spaced seeds for similarity levels $0.7, 0.71, ..., 0.99$

4.8 Computing discriminability

We can describe now the new approach for comparing seeds for oligonucleotide design. We shall build upon the exiting approach in a consistent way. We shall keep the concepts of discriminability and efficiency. As we argued in detail that they cannot be mixed, efficient discriminability no longer exists.

It makes sense to define the discriminability using the F-score measure from statistics, however, its definition must be appropriately used. Because the F-score is a measure for the correctness of one test, the used statistical parameter values, precision and recall, must be computed for the same test. In the case of oligo design, we want to evaluate a seed to see how well it can discriminates between sequences which are oligos and sequences which are not. Discriminating means whether a seed hits an oligo sequence or not. Therefore, the correct statistical parameters are:

TP: Number of oligos that are hit.

FP: Number of non-oligos that are hit.

TN: Number of non-oligos that are not hit.

FN: Number of oligos that are not hit.

This will give us the correctly computed F-score which will constitute the new discriminability:

$$F = 2\frac{PR}{P+R}. \tag{4.10}$$

main seq.		CGGGGCAACCGCGACAACTT		
secondary sequences	1	CGGGGCAACCGCGACAACTT	oligo	hit
	2	CGGGGCAACCGCGACGCCTT	oligo	hit
	3	CGGGGCTACCGCGAAAGCTT	non-oligo	hit
	4	CGGGTCCACCGCGCCAACAT	non-oligo	hit
	5	TCGCGCAAACGCGACATCTC	non-oligo	hit
	6	CCGGGATACCCCGACACCCG	non-oligo	not hit
	7	CCGGGGTAGCTCGACTCCTG	non-oligo	not hit
	8	CCGCGCGACATCTACAGTGC	non-oligo	not hit
	9	GGGAGTGCCTAGAACAGCTC	non-oligo	not hit
	10	CATAGGACAGGCCACGTATA	non-oligo	not hit
	11	CGTGCACAGGAAGTTGGCGG	non-oligo	not hit

Table 4.13: An example of a main sequence with its associated secondary sequences, marked as oligos or non-oligos, resp., identifying which ones are hit and which ones are not by the seed $s = 1 * 11 * 1 * 1 * * * 11$.

The efficiency will be $1/A$ with A being again the average rate of the number of seed hashes in an oligo. It no longer matters how the definition of efficiency depends on A, as long as it is inversely proportional, since efficiency is going to be considered separately from discriminability.

For the purpose of better understanding, we describe our approach using a small example. Assume we want to evaluate the discriminability of the seed

$$s = 1 * 11 * 1 * 1 * * * 11$$

against a sample data set given in Table 4.13. The method used for generating such data set consisting of oligo and non-oligo sequences will be given in the experiments chapter. For now, it is enough to know that the data set consists of a set of target (main) sequences and a set of secondary sequences associated with each target sequence, divided into two categories: oligos and non-oligos. Our sample set in Figure 4.13 has only one main sequence given in the first line of the data set. Based on some criteria which will be discussed in next chapter, from the 11 secondary sequences, 2 are selected as oligos and the rest are non-oligos.

In order to check which of the secondary sequences are hit by the seed s, all possible hashes of s that appear in the main sequence are computed by aligning s with the main sequence in all possible positions; the letters in the main sequence which are aligned with 1's in s give the hashes. For our example, all seed hashes are given in the Table 4.14.

Then, each seed hash is searched for in all the secondary sequences. If a hash is found, we say that s hits that sequence, otherwise it does not. For instance, the first secondary sequence is hit at any position since it is identical with the main sequence. On the other hand, the fifth secondary sequence is hit exactly at one position, as shown in Figure 4.10.

For our example, the last column of Table 4.13 indicates which sequences are hit and which are not.

As it is shown in Table 4.13, the sequences 1 and 2 are the true positives because they are oligo and are hit by the s. False positives are the sequences 3, 4, 5 and true negatives are 6, 7,

	CGGGGCAACCGCGACAACTT
1.	C∗GG∗C∗A∗∗∗CG
2.	G∗GG∗A∗C∗∗∗GA
3.	G∗GC∗A∗C∗∗∗AC
4.	G∗CA∗C∗G∗∗∗CA
5.	G∗TA∗C∗C∗∗∗AC
6.	C∗AC∗G∗G∗∗∗AC
7.	A∗CC∗C∗A∗∗∗CT
8.	A∗CG∗G∗C∗∗∗TT

Table 4.14: The target sequence seed hashes

```
main sequence      CGGGGCAACCGCGACAACTT
5th secondary seq. TCGCGCAAACGCGACATCTC
                    1*11*1*1***11
```

Figure 4.10: The fifth sequence is hit exactly once by the seed 1*11*1*1***11.

8, 9, 10, 11. There are no false negatives because all the oligo sequences are hit by the seed. Having these statistical parameters, we can now compute the precision and recall for the seed s.

$$P = \frac{TP}{FP + TP} = \frac{2}{2+3} = 0.4, \quad R = \frac{TP}{FN + TP} = \frac{2}{2} = 1. \tag{4.11}$$

The seed discriminability is:

$$F = \frac{(1+1) \times 0.4 \times 1}{0.4 + 1} \approx 0.57\,. \tag{4.12}$$

4.8.1 Recall-ordered discriminability

There is another, very important, aspect of our comparison framework. The F-score estimates the accuracy of a seed to discriminate between oligos and non-oligos however, the danger is that sequences similar to a probe are missed. That is, we want the recall to be no less than a certain value, to ensure very few false negatives. Therefore, the next step of our comparison approach is using the discriminability values computed above in the right way. For that purpose, we shall fix first a lower bounds for the recall. Only seeds with the recall larger or equal to this lower bound are considered for comparison. After that we shall vary the lower bound for the recall in order to give a complete picture. The tests are done in the next chapter where the conclusions about which seeds are the best are drawn as well.

Chapter 5

Experiments

In this chapter we compare the seeds and, for this purpose, we shall test them on two data sets. In [3], a package of tools is built which consists of three programs written in Python programming language; OligoGenerator, SeedEvaluator, and SeedChooser. SeedEvaluator evaluates seeds using their three defined measures, discriminability, efficiency, and efficient discriminability, SeedChooser recommends good seeds for certain given parameters, and OligoGenerator generates a set of oligos under experimental conditions. We prepared our data sets using OligoGenerator. We explain first how OligoGenerator generates its data sets and then describe the data sets to be used in the tests. One of our data sets is similar to the one of [3], the other is new. We discuss also the differences between the former one and the similar one in [3]. After that we perform the testing and in the end discuss the results.

5.1 The OligoGenerator program

The OligoGenerator program works as follows. It gets as inputs several parameters:

- The number of target (main) sequences,
- The number of variations for each sequence (secondary sequences), and
- The length of each sequence.

The program then peforms the following steps

1. It generates the required number of target sequences following a Bernoulli model which randomly selects characters from the alphabet {A, C, G, T}.
2. For each target sequence, it produces the required number of variations (secondary sequences) which have the same length and are obtained using FASTA [16].
3. Some of the secondary sequences are selected to be oligos whereas the remaining ones are non-oligos according to the selection criteria of He *et al* [5]. In the study performed in [5], after comparing theoretical predictions and experimental hybridization, a combination of three criteria is suggested for designing 50-mer and 70-mer oligos.

   To design 50-mer oligos, the suggested combination is

(a) Identity level with target sequence is over 85%,

(b) The maximum stretch of continuous matches is at least 15bp,

(c) The hybridization free energy is less than −30 kcal/mol.

To design 70-mer oligos, the suggested combination is

(a) Identity level with target sequence is over 85%,

(b) The maximum stretch of continuous matches is at least 20bp,

(c) The hybridization free energy is less than −40 kcal/mol.

The results of [5] also suggested that relaxing one or more criteria may keep more qualified oligo candidates.

Although OligoGenerator uses the criteria of [5] for oligo selection, it does not use, by default, the exact thresholds suggested by it, which we mentioned above. It gives the user the option to specify the thresholds according to the experimental condition.

OligoGenerator aligns variations with the respective target sequence. The alignment is done using CLUSTAW [24] which is a multiple alignment tool and accepts input sequence files in FASTA format. After the alignment, the identity level and number of continuous matches of variation and its target sequence are obtained. The free energy of the alignment is computed using OligoArrayAux [12] which is a free software for calculation of the melting temperatures. Provided that the variation alignment with its target satisfies the given thresholds, the variation is hybridizable with the target under given conditions and it will be selected as an oligo for the target sequence.

5.2 Data sets

The experiments of [3] were performed on both a simulated data set and one real data sets. (We already mentioned that the real data set is unavailable so we shall focus on the simulated one.) The simulated data set was generated using OligoGenerator and consists of 100 target sequences of length 50bp. Each target sequence has been mutated into 5000 variations. The difference between the parameters used and those mentioned above from the model of [5] is that the free energy threshold was set to −40 kcal/mol instead of −30 kcal/mol.

For our experiment, we generate two data sets using OligoGenerator which have the same number of sequences and number of variations as those of the paper. One of our data sets contains sequences of length 50bp, the other 70bp. The thresholds we set for three criteria for selecting oligos in OligoGenerator are those suggested by [5]. Of all 500,000 generated sequences, the number of oligos and non-oligos was 88,570 and 411,430, resp., for the 50-mer data set 116,301 and 383,699, resp., for the 70-mer data set.

5.3 Results

For both data sets generated as above, we have computed the discriminability and efficiency values, as described in the previous chapter, for all seeds under consideration.The programming

languages used for implementation of these computations are Java and Python.

The discriminability results are given in Figure 5.1 for the 50-mer data set and in Figure 5.2 for the 70-mer data set. Recall that, as explained in the previous chapter, the abscissa indicates the lower bound for the recall and the ordinate gives the best discrimination value for a seed of the considered type.

We can see a clean ranking of the seeds in Figure 5.1, as we expected. The contiguous ones are the worst, then come the single spaced and transition seeds which are very much comparable with a slight advantage of the spaced seed. Continuing in the increasing order of discriminability, next follow the 2-spaced seeds, then 4-spaced seeds, and finally the 8-spaced seeds. There is clear improvement from 2-seeds to 4-seeds and little, but consistent, from 4-seeds to 8-seeds.

Our results appear to be very consistent as they repeat identically for the 70-mer data set, shown in Figure 5.2. The answer appears to be very clear. Multiple spaced seeds are the best. Also, the more the better. However, this impacts, as expected, the efficiency.

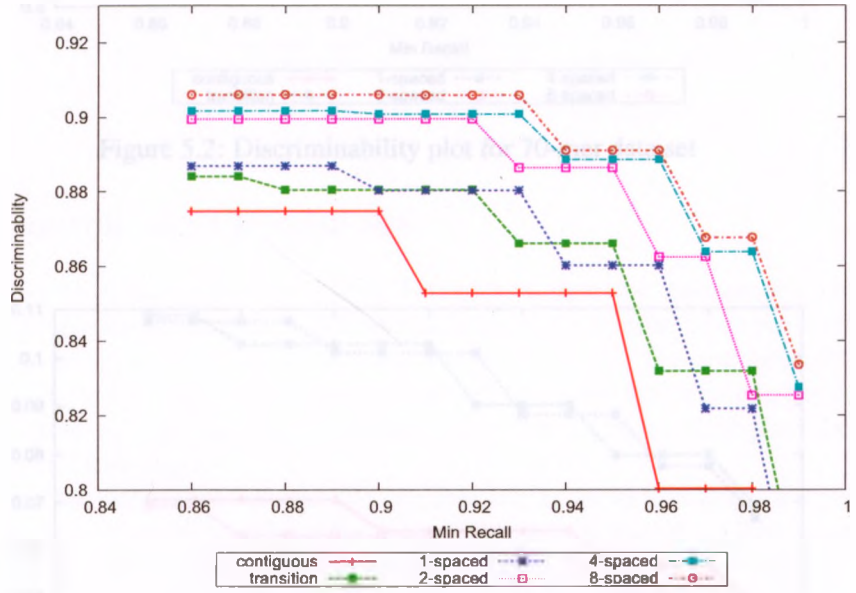

Figure 5.1: Discriminability plot for 50-mer data set

The efficiency plots, shown for the 50-mer data set in Figure 5.3 and for the 70-mer data set in Figure 5.4 are again expected. Increasing the number of seeds decreases the efficiency. Notice however that both single spaced and transition seeds are also more efficient than the contiguous ones. Between the two, the single-space seeds are slightly more efficient than the transition ones.

As explained earlier, discriminability and efficiency cannot be mixed. Taken separately, they show clearly the expected ranking. Together, they give the trade off: better discriminability comes with a price in efficiency (except when contiguous seeds are replaced by transition-constrained or single spaced seeds).

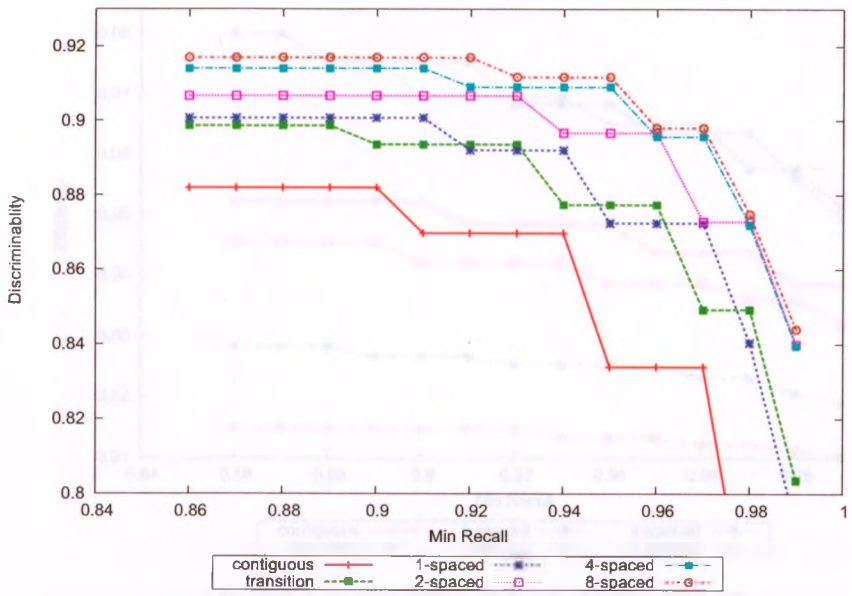

Figure 5.2: Discriminability plot for 70-mer data set

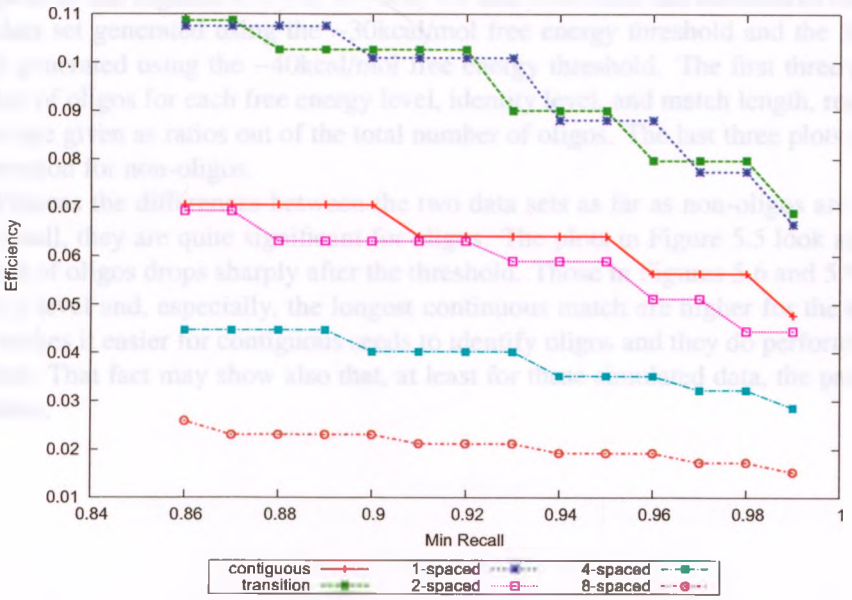

Figure 5.3: Efficiency plot for 50-mer data set

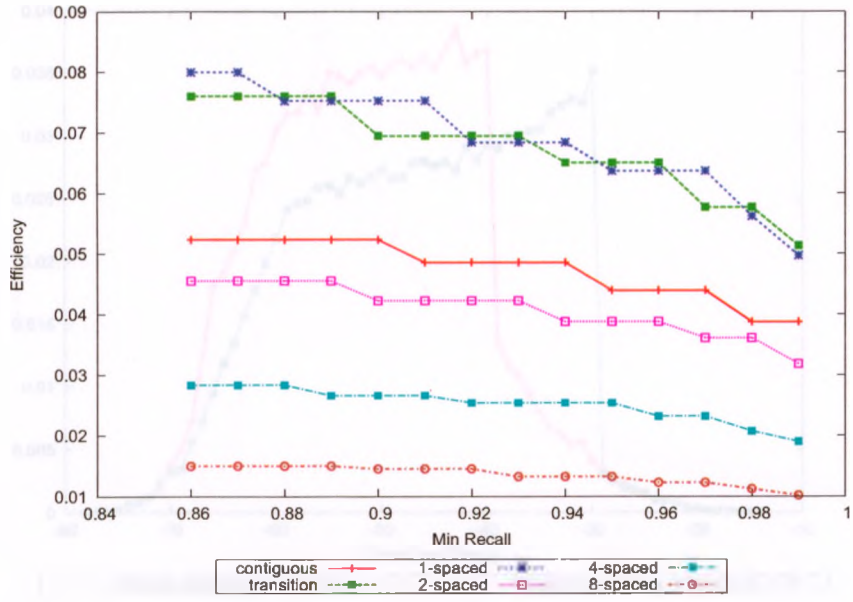

Figure 5.4: Efficiency plot for 70-mer data set

5.4 Comparing the data sets

The plots in the Figures 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10 show the differences between our 50-mer data set generated using the −30kcal/mol free energy threshold and the 50-mer data set of [3] generated using the −40kcal/mol free energy threshold. The first three plots show the number of oligos for each free energy level, identity level, and match length, respectively. The values are given as ratios out of the total number of oligos. The last three plots show the same information for non-oligos.

Whereas the differences between the two data sets as far as non-oligos are concerned are very small, they are quite significant for oligos. The plots in Figure 5.5 look as expected; the number of oligos drops sharply after the threshold. Those in Figures 5.6 and 5.7 show that the identity level and, especially, the longest continuous match are higher for the data set of [3]. That makes it easier for contiguous seeds to identify oligos and they do perform in some cases the best. That fact may show also that, at least for these simulated data, the parameters of [5] are better.

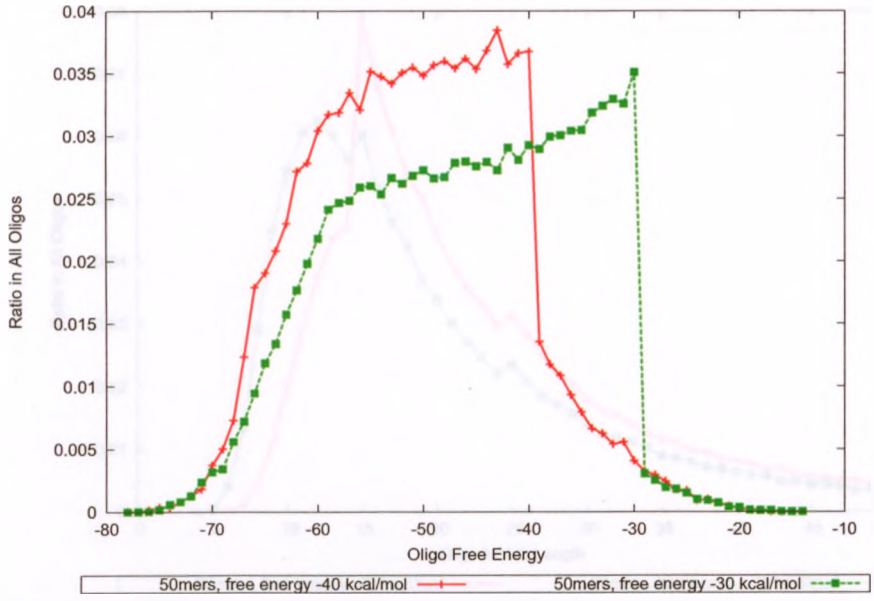

Figure 5.5: Ratio of oligos with various free energy levels for the 50-mer data set with −30kcal/mol free energy threshold and the 50-mer data set with −40kcal/mol threshold.

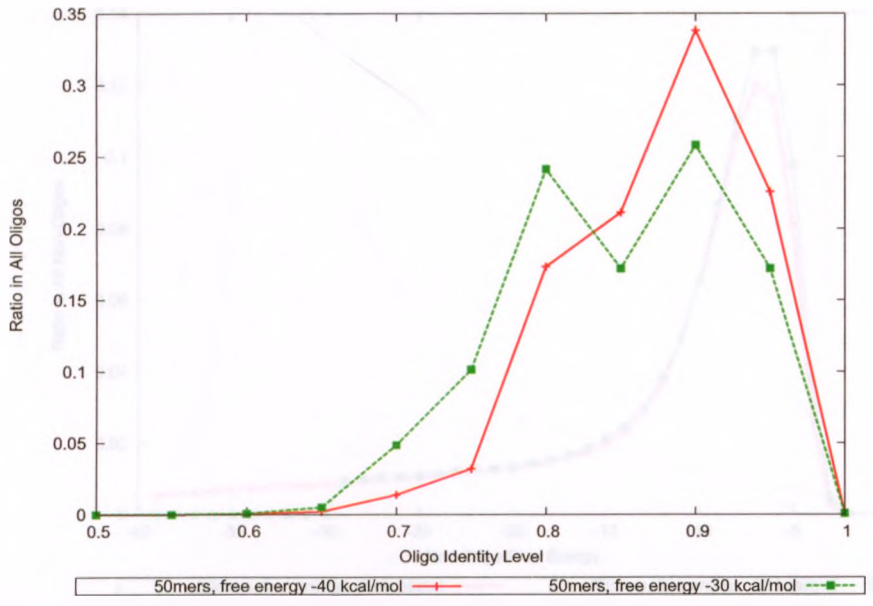

Figure 5.6: Ratio of oligos with various identity levels for the 50-mer data set with −30kcal/mol free energy threshold and the 50-mer data set with −40kcal/mol threshold.

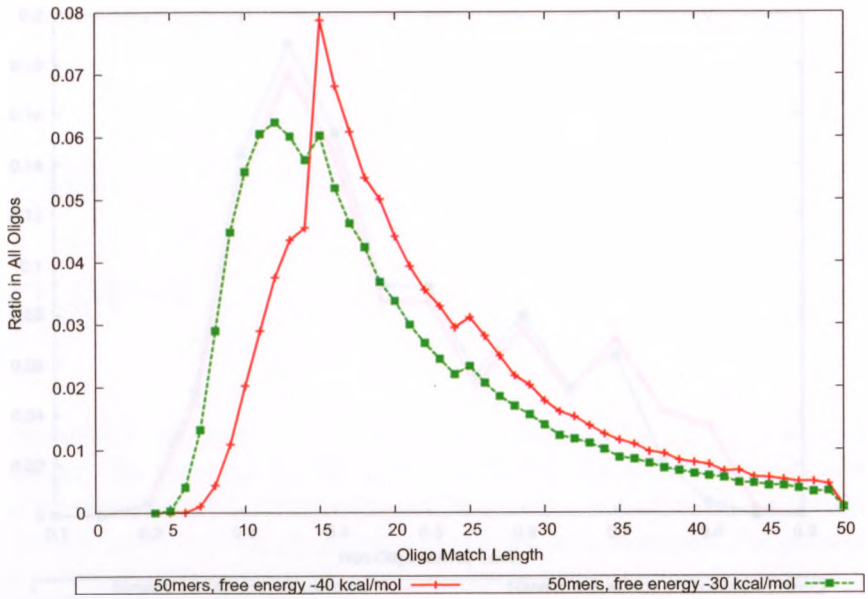

Figure 5.7: Ratio of oligos with various match lengths for the 50-mer data set with −30kcal/mol free energy threshold and the 50-mer data set with −40kcal/mol threshold.

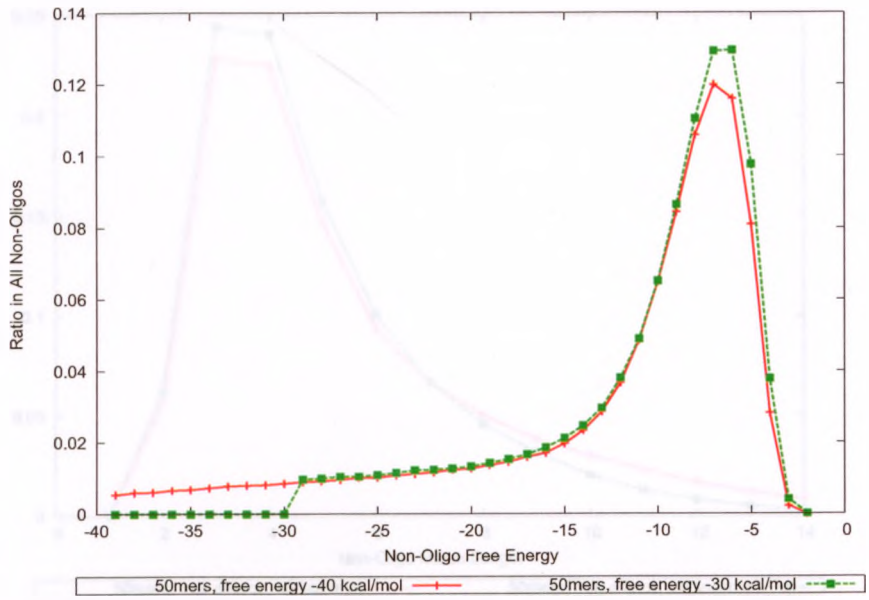

Figure 5.8: Ratio of non-oligos with various free energy levels for the 50-mer data set with −30kcal/mol free energy threshold and the 50-mer data set with −40kcal/mol threshold.

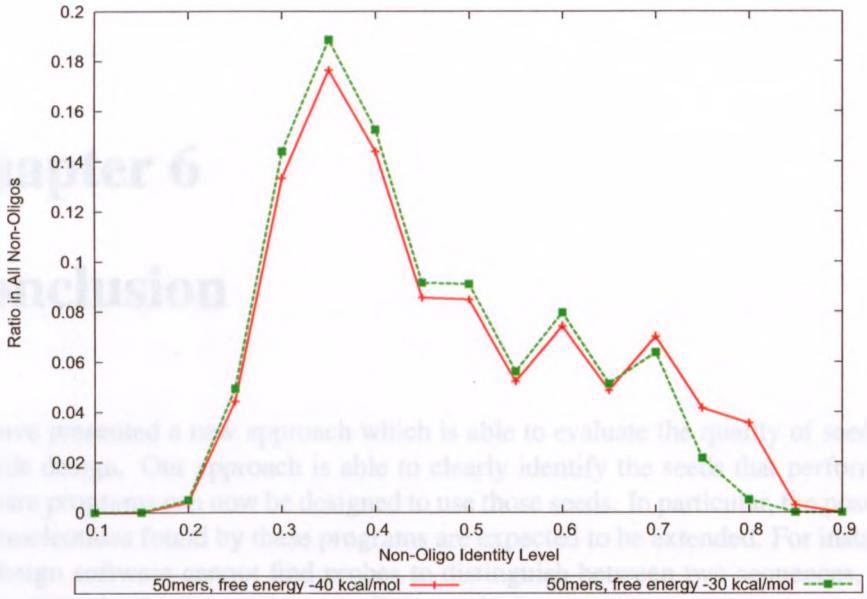

Figure 5.9: Ratio of non-oligos with various identity levels for the 50-mer data set with −30kcal/mol free energy threshold and the 50-mer data set with −40kcal/mol threshold.

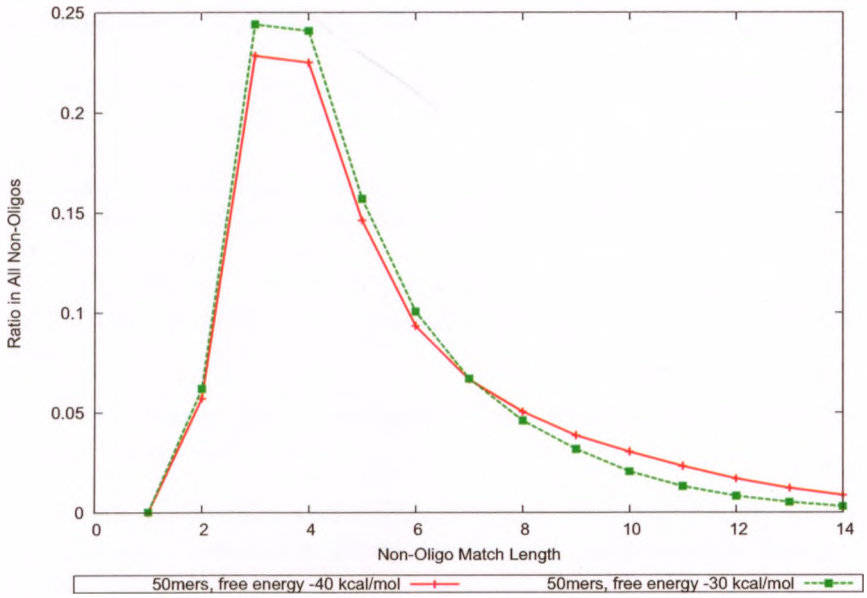

Figure 5.10: Ratio of non-oligos with various match lengths for the 50-mer data set with −30kcal/mol free energy threshold and the 50-mer data set with −40kcal/mol threshold.

Chapter 6

Conclusion

We have presented a new approach which is able to evaluate the quality of seeds for oligonucleotide design. Our approach is able to clearly identify the seeds that perform well. Better software programs can now be designed to use those seeds. In particular, the possibilities of the oligonucleotides found by these programs are expected to be extended. For instance, when the ProDesign software cannot find probes to distinguish between two sequences, the sequences are considered to be too similar to be distinguished and appropriately clustered. Such clustering may be greatly reduced by the increased sensitivity of the multiple spaced seeds we propose.

Bibliography

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215, 1990.

[2] Z. Bozdech, J. Zhu, M.P. Joachimiak, F.E. Cohen, B. Pulliam, and J.L. DeRisi. Expression profiling of the schizont and trophozoite stages of plasmodium falciparum with a long-oligonucleotide microarray. *Genome Biol.*, 4:R9, 2003.

[3] W.H. Chung and S.B. Park. An empirical study of choosing efficient discriminative seeds for oligonucleotide design. *BMC Genomics*, 10(Suppl 3):S3, 2009.

[4] S. Feng and E.R.M. Tillier. A fast and flexible approach to oligonucleotide probe design for genomes and gene families. *Bioinformatics*, 23(10):1195 – 1202, 2007.

[5] Z. He, L. Wu, X. Li, M. Fields, and J. Zhou. Empirical establishment of oligonucleotide probe design criteria. *Appl Environ Microbiol*, 71(7):3753–3760, 2005.

[6] L. Ilie and S. Ilie. Multiple spaced seeds for homology search. *Bioinformatics*, 23(22):2969 – 2977, 2007.

[7] L. Kaderali and A. Schliep. Selecting signature oligonucleotides to identify organisms using dna arrays. *Bioinformatics*, 18(10):1340–1349, 2002.

[8] M. Kane, T. Jatkoe, C. Stumpf, J. Lu, J. Thomas, and S. Madore. Assessment of the sensitivity and specificity of oligonucleotide (50 mer) microarrays. *Nucleic Acids Res*, 28, 2000.

[9] F. Li and G.D. Stormo. Selection of optimal dna oligos for gene expression arrays. *Bioinformatics*, 17:1067–1076, 2001.

[10] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter ii: Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 2(3):417 – 440, 2004.

[11] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18, 2002.

[12] N.R. Markham and M. Zuker. Dinamelt web server for nucleic acid melting prediction. *Nucleic Acids Res*, 33(3):W577 – W581, 2005.

[13] E. W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J ACM*, 46, 1998.

[14] H.B. Nielsen, R. Wernersson, and S. Knudsen. Design of oligonucleotides for microarrays and perspectives for design of multi-transcriptome arrays. *Nucleic Acids Res*, 31:3491–3496, 2003.

[15] L. Noe and G. Kucherov. Yass: enhancing the sensitivity of dna similarity search. *Nucleic Acids Research*, 33, 2005.

[16] W. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms. *Genomics*, 11, 1991.

[17] S. Rahman. Fast large scale oligonucleotide selection using the longest common factor approach. *Journal of Bioinformatics and Computational Biology*, 1(2):343 – 361., 2003.

[18] N. Reymond, H. Charles, L. Duret, F. Calevro, G. Beslon, and J.M. Fayard. Roso: optimizing oligonucleotide probes for microarrays. *Bioinformatics*, 20:271–273, 2004.

[19] S. Rimour, D. Hill, C. Militon, and P. Peyret. Goarrays: highly dynamic and efficient microarray probe design. *Bioinformatics*, 21, 2004.

[20] J.M. Rouillard, C.J. Herbert, and M. Zuker. Oligoarray: genome-scale oligonucleotide design for microarrays. *Bioinformatics*, 18(3):486–487, 2002.

[21] J.M. Rouillard, M. Zuker, and E. Gulari. Oligoarray 2.0: design of oligonucleotide probes for dna microarrays using a thermodynamic approach. *Nucleic Acids Res*, 31, 2003.

[22] J.J. SantaLucia. A unified view of polymer,dumbbel,and oligonucleotide dna nearest-neighbor thermodynamics. *Proc. Natl Acad. Sci. USA*, 95, 1998.

[23] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Nucleic Acids Res*, 147, 1981.

[24] J.D. Thompson, D.G. Higgins, and T.J. Gibson. Clustaw: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(2):4673 – 4680, 1994.