Designing oligonucleotides for DNA microarrays

Diploma Thesis
Department of Computer Science
ETH Zurich

Designing Oligonucleotides for DNA Microarrays

Thomas Brunner

Advisors:
Prof. Dr. Gaston Gonnet
Dr. Peter von Rohr

May 12, 2003 – September 11, 2003

Abstract

The microarray technology has attracted a tremendous amount of interest recently because it allows to monitor a large number of genes in parallel in a single experiment. While a considerable amount of research has been done in the field of microarray image analysis, relatively little work has been spent on oligonucleotide design. The selection of the oligonucleotides which will be put on the microarray stands at the beginning of every experiment. Because the hybridization and melting process is at the core of the microarray technology, we first conceive a model to predict the melting temperatures between oligonucleotides and the DNA in the sample. For the actual oligonucleotide selection, different strategies need to be pursued depending on the type of experiment. In principle we always start with a large pool of candidate oligonucleotides and then narrow down this set until we have the desired number of oligonucleotides. We present two complementary approaches. The first approach uses suffix arrays to efficiently select oligonucleotides which are as specific as possible to a specified gene. The chosen algorithm is optimized to minimize memory and time requirements. The second approach can be used to produce characteristic fingerprints or signatures of a gene. Fingerprinting only works if we restrict ourselves to a particular family of genes, for example from a single organism. Our fingerprinting algorithm uses the concept of entropy from information theory. This method attempts to maximize the information content of the chosen oligonucleotides. We conclude our work by analyzing the running times of the algorithms and a description of our implementation in DARWIN.

Zusammenfassung

Die Microarray-Technologie ist in letzter Zeit auf enormes Interesse in der Forschergemeinde gestossen, weil sie es zum ersten Mal ermöglicht, in einem einzigen Experiment eine grosse Anzahl von Genen gleichzeitig zu untersuchen. Während auf dem Gebiet der Microarray-Bildanalyse schon recht viel Forschung betrieben wurde, gibt es über das Design der Oligonukleotiden relativ wenig Material. Die Auswahl der Oligonukleotiden steht immer am Beginn eines Microarray-Experiments. Weil der Schmelz- und Hybridisierungsvorgang den Kern der Microarray-Technologie ausmachen, entwickelten wir zuerst ein Modell um die Schmelztemperaturen zwischen den Oligonukleotiden und der untersuchten DNA zu bestimmen. Für das eigentliche Auswahlverfahren gibt es verschiedene Möglichkeiten, wobei je nach Art des Experiments die eine oder andere Möglichkeit näher liegt. Das Prinzip ist dabei immer dasselbe. Wir beginnen immer mit einer grossen Auswahl von Kandidatenoligonukleotiden und filtern dann ungeeignete Kandidaten heraus, bis wir die gewünschte Anzahl von Oligonukleotiden haben. Wir stellen zwei Ansätze vor, die sich ergänzen. Der erste Ansatz benutzt Suffix-Array-Datenstrukturen um möglichst effizient Oligonukleotide auszuwählen, die spezifisch für ein bestimmtes Gen sind. Der vorgestellte Algorithmus minimiert Speicher- und Zeitanforderungen. Der zweite Ansatz generiert charakteristische Fingerabdrücke von Genen. Dies kann aber nur funktionieren, wenn wir uns auf eine bestimmte Familie von Genen beschränken, z.B. von einem einzelnen Organismus. Der vorgstellte Algorithmus benutzt dabei das Konzept der Entropie aus der Informationstheorie. Diese Methode versucht den Informationsgehalt der ausgewählten Oligonukleotiden zu maximieren. Wir schliessen die Arbeit mit einer Untersuchung der Laufzeiten der beiden Algorithmen und einem Überblick über unsere Implementation in DARWIN ab.

Contents

Chapter 1

Introduction

1.1 DNA Microarrays

It is today widely believed that in all living organisms thousands of genes and their products such as RNA or proteins function in a complex and orchestrated way that creates the mystery of life. However, traditionally research in molecular biology has worked on a "one gene in one experiment" basis which meant that throughput has been very limited and that it was hard if not impossible to obtain the "whole picture" of a gene function. In the past few years, DNA microarrays have attracted a tremendous amount of interest among biologists. For the first time this technology promises to monitor the whole genome in a single experiment which provides a much better insight into the complicated interactions between the genes in a genome of an organism. Terminologies that describe this technology that are being used include, but are not limited to: *Biochip, DNA chip, DNA microarray, genome chip* and *gene array.* Affymetrix Inc.[1] owns a registered trademark, *GeneChip*, which refers to its patented and proprietary DNA array technology. Figure 1.1 shows such a GeneChip.

Base-pairing (i.e. A-T and G-C for DNA; A-U and G-C for RNA) or hybridization is the key principle behind microarrays. A microarray is a plastic or glass slide containing a large number of so-called spots. Each spot consists of many copies of a known DNA sequence, also called a *probe.* The spot size is typically less than 200 microns in diameter and one microarray can contain thousands of spots. The probes in the spots are then allowed to hybridize with the DNA under examination which we will call the *target.* Whenever the Watson-Crick complementary sequence of a probe is present in the target DNA, that DNA will bind (hybridize) to the probe. After a while, all unhybridized sequences are washed off the chip and the amount of hybridized sequences at each spot can be measured. DNA microarrays are fabricated by high-speed robotics. There are two variants of the DNA microarray technology in terms of the properties of the probes:

cDNA probes. Each spot consists of probe cDNA (complementary DNA, 500–5000 bases long) immobilized to a solid surface such as glass. The probes are put on the slide by robots and are immobilized for example by using an amino linker. The slide can be reused by replacing the probes.

[1]`http://www.affymetrix.com`

Figure 1.1: Affymetrix GeneChip probe array. Image courtesy of Affymetrix.

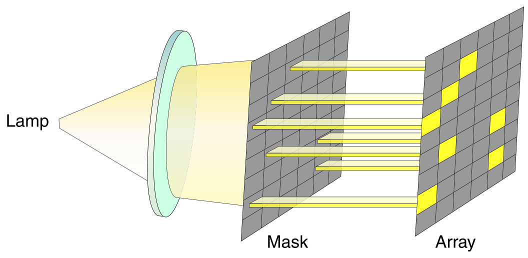

Figure 1.2: GeneChip probe arrays are manufactured through a combination of photolithography and combinational chemistry. Image courtesy of Affymetrix.

Oligonucleotide probes. Each spot consists of oligonucleotide (20–80 bases long) or peptide nucleic acid (PNA) probes synthesized either *in situ* (on-chip) or by conventional synthesis followed by on-chip immobilization like it is done with the cDNA probes. Affymetrix uses a proprietary in situ technique where the oligonucleotides are synthesized layer by layer using photolithography. This process is similar to the one used in computer chip fabrication. Figure 1.2 depicts the process. Microarrays with on-chip synthesized probes cannot be reused with different probes.

Furthermore, we will distinguish three major fields of application for the DNA microarray technology:

Identification and detection of DNA sequences. The goal of these experiments is usually the identification or detection of a particular gene or gene mutation in the target.

Comparative gene expression experiments. In these experiments the expression levels of certain genes in the target are compared against the expression levels in a reference sample, for example to determine the impact of certain external factors. The target DNA is obtained by reverse-transcribing mRNA (messenger RNA) from the cell nucleus of the target organism. We thus only deal with coding regions of the DNA.

Classification of genes. Here we use microarrays to obtain signatures or fingerprints of genes or gene families to classify them without the need to completely sequence them.

There are several steps in the design and implementation of DNA microarray experiments as the following overview shows:

Probe design The first step is to determine the probes that will be put on the chip. Since all the following steps will depend on the quality of the probes, this step is important in order to maximize the information gained from an experiment. A simple and straightforward solution would be to put the whole genome of the organism under examination on the chip, but in most cases this is not feasible but also not necessary. Various effects such as cross-hybridization or melting temperature need to be considered.

Chip fabrication After the first step which was a purely theoretical exercise, the chosen probes have to be synthesized and then immobilized on the chip. As already mention, the synthesis can be done directly on-chip or in a separate step. Figure 1.3 shows the relation between the microarray, spots and oligonucleotides. If we want use cDNA we first have to reverse-transcribe the mRNA.

Target labeling In order to later detect the target DNA which has hybridized with the probes on the microarray, we must label them with a reporter molecule that identifies their presence. This is usually done with fluorescent tags. Depending on the microarray used, it is sometimes also possible to use two samples in one experiment which is particularly useful for comparative gene expression experiments. In that case we use two different fluorescent tags so that we are able to distinguish the two samples. Note that the number of fluorescent molecules that label each target sequence is unknown since it depends on its length and sequence composition. Fluorescent intensities can therefore not be quantitatively compared. However, results from two experiments where the same number of labeling molecules are added to the same target are still comparable.

Assay The target is now allowed to hybridize with the probes. It is important that also strands which are not perfectly complementary in the Watson-Crick sense may hybridize. Figure 1.4 shows the hybridization process.

Scanning After the target samples which did not hybridize have been washed off the slide, the intensities of the reporter molecules can be measured using a scanner. Figure 1.5 shows such a scan. It should be noted that the colors on these images are false colors and are just for illustration. The actual fluorescent tags do not show their colors unless stimulated with a specific frequency of light by a laser. Even then, the colors are not directly observed, rather the wavelength of the emitted light is used to tune a detector which measures the fluorescence.

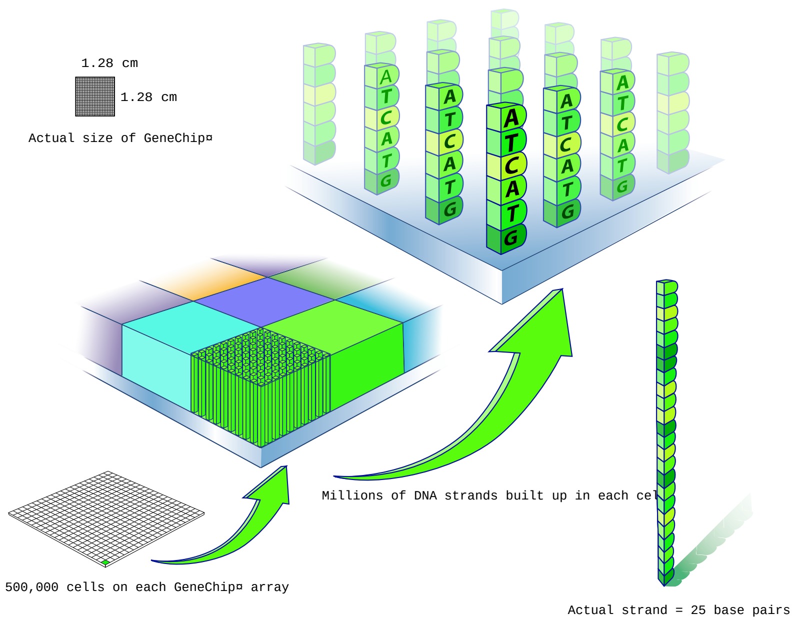
1.28 cm
1.28 cm
Actual size of GeneChip¤
A T C A T G
Millions of DNA strands built up in each cel
500,000 cells on each GeneChip¤ array
Actual strand = 25 base pairs

Figure 1.3: A single spot (cell in Affymetrix terminology) with oligonucleotides of length 25 on a microarray. Image courtesy of Affymetrix.

Interpretation The final product of such a microarray experiment is the aforementioned scanned image. While in principle one can now simply measure intensities, there are a number of problems which need to be considered. Besides the actual intensities, the scanned image also contains different kinds of noise, for example from irregular spots, dust on the slide or non-specific hybridization. Even after overcoming these detection and calibration problems, there are other issues, such as inherently lower concentrations of some of the target sequences. A considerable amount of research has been done in the field of microarray analysis. [Sch03] provides a good overview.

While there are numerous publications about microarray analysis, relatively little material is available about the probe design and selection.

1.2 Thesis Objectives and Purpose

The goal of this thesis is to investigate ways how to design suitable oligonucleotide molecules for DNA microarray experiments and to implement the found methods in *DARWIN*[2]. In order to design oligonucleotides, one must understand the interactions between the microarray and the sample under examination and then conceive a model of those processes. In this thesis we will look at two approaches to these issues, the first approach is primarily for experiments where the goal is to identify or detect genes and the second approach allows to create fingerprints or signatures of genes. Both methods will be implemented in

[2]For more information about DARWIN see `http://cbrg.inf.ethz.ch/Darwin/`

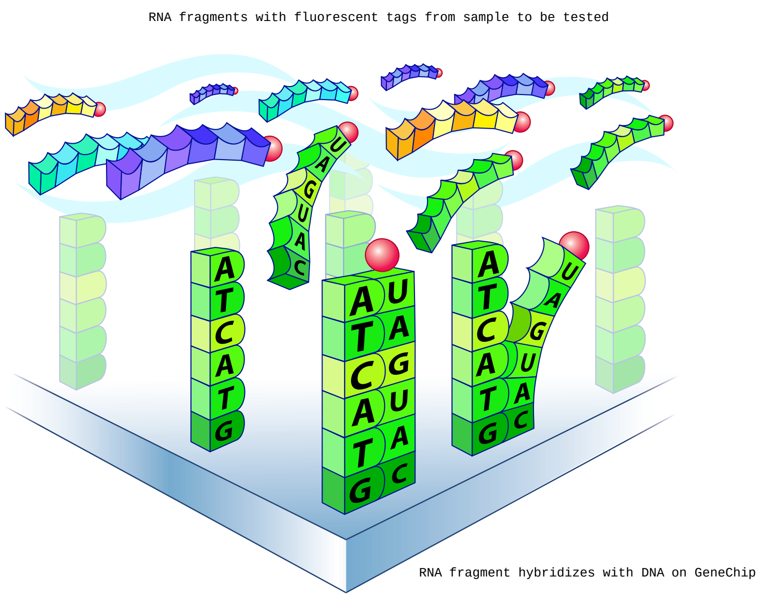
RNA fragments with fluorescent tags from sample to be tested
RNA fragment hybridizes with DNA on GeneChip
ATCATG
AUTACGCAUUTAGGC
ATCATG
UAGUAC
ATCATGUAGUAC

Figure 1.4: Depiction of the hybridization between probes and target. Here the target is RNA. The spheres represent the reporter molecules. Image courtesy of Affymetrix.

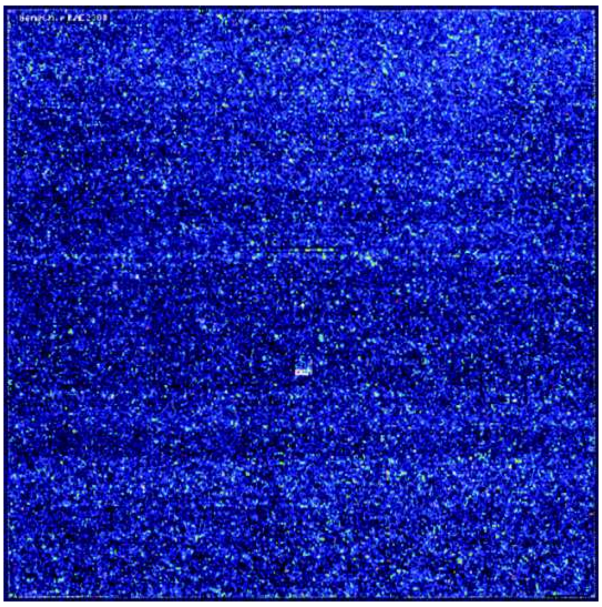

Figure 1.5: A scanned microarray image.

DARWIN using interpreted DARWIN code or compiled C code where interpreted code is too slow. An overview over the implementation and results will conclude the thesis.

1.3 Related Work

The selection algorithm we will present here is closely related to the method presented in [Rah02] by Rahmann *et al.* Unlike earlier methods, the approach presented there is not based so heavily on melting temperature prediction. While it is true that the hybridization and melting process is at the core of every microarray experiment, it is still not fully understood and most existing research focuses on melting temperature prediction in solution and not on microarrays. Another advantage of the Rahmann approach is that it is considerably faster than earlier methods, even though it still gets quite time-consuming with large sequences. Those are the two main reasons why we chose this method as the basis of our implementation. Notable earlier work include [LS01] and [KS02]. The method presented in [LS01] by Li and Stormo is based on the fact that the melting temperature of a near-match is sufficiently low if the near-match contains more than four errors (mismatches, insertions or deletions). Melting temperature estimation is then done only for the remaining candidate oligonucleotides. The approach taken by Kaderali and Schliep in [KS02] uses a heuristic dynamic programming algorithm that tries to find the most stable alignment of a candidate oligonucleotide to every sequence. The specificity of a candidate oligonucleotide is determined by looking at the difference of the melting temperatures between the perfect match to the target gene and the second best match. In principle this approach should provide good results but because of the uncertainties with respect to melting temperature prediction and the heuristic alignment algorithm it remains unclear how close this unspecificity measure comes to reality.

The work mentioned so far mostly deals with oligonucleotide selection for gene identification or detection. For the class of experiments where the goal is to create characteristic signatures or fingerprints of genes, our work will be based on the results presented in [HSS$^+$00] which uses the concept of entropy from information theory.

Chapter 2

Melting Temperature Prediction

When DNA microarray experiments are performed, what is actually measured in the end is the amount of oligonucleotide molecules on the chip that hybridized with a matching sequence. It is therefore of importance to understand the hybridization process in order to choose the best oligonucleotide molecules. In particular it is very useful to predict the temperature at which a given DNA sequence hybridizes.

2.1 DNA Hybridization Properties

When DNA in the double helix state is heated, it will start to denature at a certain temperature and if the temperature is being increased even further, the two strands will separate or *melt* completely. Conversely, if single DNA strands are being cooled down, they will hybridize appropriately i.e. they will form complementary base pairs in the Watson-Crick sense. This is also termed *annealing.* The definition of the melting temperature T_m is as follows:

Definition (Melting Temperature T_m) Given an equal amount of a known DNA sequence and its reverse complement. The melting temperature T_m of this sequence is the temperature at which 50% of the strands are in denatured (or *random coil*) form and 50% are in double helical form.

Note that the above definition implies that only perfectly complementary sequences will hybridize. This is not true but for the moment it is being assumed to be the case. Furthermore, the transition from double helical state to a fully denatured state is not a single cooperative transition but may happen with many intermediate stable states. Since a large number of melting temperature predictions have to be performed for our purpose of probe design, a two-state model where a particular sequence is either in random coil or double helical state is being chosen to keep the computational complexity at an acceptable level.[AS97] and [San98] showed that for oligonucleotides on DNA chips this is a reasonable approximation of the reality and that the intermediate melting states are mainly significant in natural polymers with heterogeneous sequences.

Therefore the following two-state reversible equilibrium annealing reaction of two DNA single strands will be the basis of our subsequent considerations.

$$S_1 + S_2 \overset{K_{eq}}{\rightleftharpoons} D \tag{2.1}$$

S_1 and S_2 denote the two reverse complementary single DNA strands, K_{eq} is the equilibrium constant, i.e. the ratio of the concentration of the double stranded oligonucleotide to the concentrations of the single stranded oligonucleotides.

$$K_{eq} = \frac{[D]}{[S_1][S_2]} \tag{2.2}$$

The interactions between bases in nucleic acids can be divided into two kinds (see also [Cup97]):

Base pairing in the plane of the base pairs due to hydrogen bonding between the opposing bases. Between G and C there are three hydrogen bonds but between A and T there are only two of them.

Base stacking in the plane perpendicular to the plane of the basis due to London dispersion forces and hydrophobic effects

2.2 Simple Models

A very simple model to estimate the melting temperature of DNA duplexes simply counts the base pairs. A very common formula is the following:

$$T_M = 2\ (\text{number of AT pairs})\ + 4\ (\text{number of GC pairs}) \tag{2.3}$$

The reasoning behind this formula is that in AT base pairs there are only two hydrogen bonds whereas in GC pairs there are three hydrogen bonds and therefore more energy is needed to break up the bonds between the GC pairs which yields a higher melting temperature in GC rich DNA duplexes. Obviously the above formula does not model the reality very precisely. A more accurate version which is still based mostly on the composition of the base pairs but also accounts for the concentration of Na$^+$ in the solution has been proposed in [BD98]:

$$\begin{aligned} T_M &= 87.16 + 34.5\frac{\text{number of GC pairs}}{\text{total number of base pairs}} + \\ & \log[Na^+]\left(20.17 - 6.6\frac{\text{number of GC pairs}}{\text{total number of base pairs}}\right) \end{aligned} \tag{2.4}$$

It can be seen that a high concentration of cations such as Na$^+$ (measured in $\frac{\text{mol}}{\text{l}}$) increase the stability of the double helix structure i.e. it increases the melting temperature. Measurements (see also [BD98]) have shown that the stabilizing effect of the cations decreases with increasing GC content of the DNA duplex. There exists an earlier version of the formula where the logarithm is not multiplied with the term which depends on the GC content.

While there is no disagreement over the fact that the melting temperature depends on the composition of the solution in which the experiment takes place, there have been only very few measurements of actual melting temperatures on microarrays. Usually for microarray experiments not the same solutions are used as for the PCR or other experiments. Furthermore, the fact that the oligonucleotides are immobilized on the chip surface by a linker does have an influence on the melting behavior. There have been many measurements conducted for melting temperatures in solution (not on a chip) and it has been shown ([OVG$^+$98] and [San98] provide a good overview) that the nearest neighbor model which will be investigated in the following sections provides a good approximation of real melting temperatures. Even though the more simple models which have been presented above are only good enough for a rough approximation, they are widely used in practice.

2.3 A Short Overview of Thermodynamics

Let us consider a generic chemical reaction:

$$A + B \rightleftharpoons P + Q \tag{2.5}$$

What happens if we mix A, B, P and Q together? It really depends on the initial concentrations. The concentrations will shift toward the equilibrium position, so depending on the initial concentrations the direction of the reaction can be to the left or to the right. The equilibrium constant is defined as follows:

$$K_{eq} = \frac{[P]_{eq}[Q]_{eq}}{[A]_{eq}[B]_{eq}} \tag{2.6}$$

Therefore if the initial ratio of products to reactants,

$$\frac{[P][Q]}{[A][B]}$$

is different from the equilibrium ratio, the reaction will proceed until the actual ratio equals the equilibrium ratio and then stops at equilibrium. Note that thermodynamics does not tell anything about how long a reaction will take, this belongs to the field of kinetics.

If

$$\frac{[P][Q]}{[A][B]} < \frac{[P]_{eq}[Q]_{eq}}{[A]_{eq}[B]_{eq}}$$

then the reaction goes in the direction that increases P and Q and decreases A and B, so the reaction goes to the right. Conversely, if

$$\frac{[P][Q]}{[A][B]} > \frac{[P]_{eq}[Q]_{eq}}{[A]_{eq}[B]_{eq}}$$

the reaction goes to the left.

Another important concept is the free energy ΔG. ΔG is a measure of how far a chemical reaction is from equilibrium and thus represents the amount of energy that can be released by a chemical reaction. If ΔG is negative for a reaction, it is called an *exergonic* reaction and it will happen from left to right. An *endergonic* reaction happens from right to left and ΔG is positive. Generally, exergonic reactions are more favorable because the reaction proceeds from a state of higher energy to one of lower energy, hence the more negative ΔG is, the more favorable the reaction.

The definition of ΔG:

$$\Delta G = -RT \ln \frac{K_{eq}}{\frac{[P][Q]}{[A][B]}} \tag{2.7}$$

or rearranged:

$$\Delta G = -RT \ln K_{eq} + RT \ln \frac{[P][Q]}{[A][B]} \tag{2.8}$$

R is the gas constant ($1.987 \frac{\text{cal}}{\text{K}\cdot\text{mol}}$). Note that the free energy of a favorable reaction can be used to make an unfavorable reaction happen. In order to be able to compare different reactions, ΔG^0 is defined as the free energy change for a reaction under conditions where the product/reactant ratio is 1. Note that ΔG^0 is *not* the free-energy change at equilibrium (that is zero).

$$\Delta G^0 = -RT \ln K_{eq} \tag{2.9}$$

and therefore

$$\Delta G = \Delta G^0 + RT \ln \frac{[P][Q]}{[A][B]} \tag{2.10}$$

The free-energy change of a chemical reaction is a balance between two factors, heat and entropy. In general, reactions that release heat are more favorable than those that do not. Similarly, reactions that increase entropy tend to be more favorable than reactions that make more organized products. Heat energy arises from chemical reactions by making and breaking chemical bonds. Some of that generated heat may be used during the reaction to organize or order the products of the reaction.

$$\Delta G = \Delta H - T\Delta S \tag{2.11}$$

ΔG = **free energy** This is the useful energy that can be obtained from a chemical reaction. Negative for favorable reactions.

ΔH = **enthalpy** This is the net amount of energy available from changes in bondings between reactants and products. If heat is given off, the reaction is favorable and ΔH is negative.

ΔS = **entropy** This is the change in the amount of order during a reaction. Order (i.e. $\Delta S < 0$) is unfavorable, disorder (i.e. $\Delta S > 0$) is favorable.

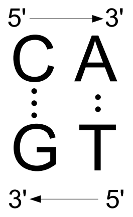

Figure 2.1: Schema of the CA respectively the TG doublet. The dots represent the hydrogen bonds.

AA = TT	
AT	
TA	
CA = TG	
GT = AC	
CT = AG	
GA = TC	
CG	
GC	
GG = CC	

Table 2.1: The 10 unique doublets.

Like with ΔG and ΔG^0 we can define ΔH^0 and ΔS^0 as the enthalpy respectively the entropy change for a reaction where the initial product/reactant ratio is one. Note that the enthalpy or the entropy of a favorable reaction can still be positive if the entropy respectively the enthalpy is negative enough to result in a negative free energy.

2.4 The Nearest Neighbor Model

The models presented so far do not account for the sequence of the strands, i.e. we only look at a particular base pair and we do not consider the neighboring bases in the two strands. The Nearest Neighbor (NN) model always considers a so-called *doublet* which simply consists of two successive base pairs. In duplex DNA there are ten such unique doublets. Some combinations are identical, for example the doublet depicted in figure 2.1 corresponds to both the CA and TG doublet. Table 2.4 lists the doublets.

[San98] compares experimental data from seven different sources and unified them into a single parameter set which is shown in table 2.2.

In order to estimate the melting temperature of a sequence we first compute the enthalpy and entropy using the unified measured values by just adding up the values of the individual doublets. This means that for a DNA duplex with n base pairs, we have n-1 doublets and thus twice n-1 values which we sum together for the entropy and the enthalpy. Additionally we add for each end of the oligonucleotide another correction parameter, depending if the duplex ends with a GC or an AT base pair. It can be shown (see [OVG+98], [BDTU74])

Doublet	ΔH^0 in $\frac{\text{kcal}}{\text{mol}}$	ΔS^0 in $\frac{\text{cal}}{\text{K}\cdot\text{mol}}$
AA = TT	-7.9	-22.2
AT	-7.2	-20.4
TA	-7.2	-21.3
CA = GT	-8.5	-22.7
GT = AC	-8.4	-22.4
CT = AG	-7.8	-21.0
GA = TC	-8.2	-22.2
CG	-10.6	-27.2
GC	-9.8	-24.4
GG = CC	-8.0	-19.9
Terminal GC	0.1	-2.8
Terminal AT	2.3	4.1

Table 2.2: Unified oligonucleotide parameters in 1 M NaCl

that the melting temperature can be estimated by the following formula:

$$T_m = \frac{\Delta H_0}{\Delta S_0 + R \cdot \log \frac{C_T}{4}} - 273.15 \tag{2.12}$$

Here C_T is the concentration of DNA strands (bound and unbound). We subtract 273.15 from the fraction to obtain the temperature in Celsius instead of Kelvin. For the enthalpy and entropy change we simply use the values from table 2.2. Take care of the units (kcal vs. cal). Even though the values provided by this formula agree quite well with measurements, it has to be noted that these values are for experiments for DNA free in solution and that the behavior on a microarray chip is indeed different. There are various publications which investigate this issue (for example [FDP+98] and [VP02]) it is currently still not very well understood and most implementations of oligonucleotide design software use the above formula. Another issue is the dependence on the strand concentration C_T. Of course, this concentration is unknown since, in some way, it is what we want to find out. So in practice, just a fixed value is used, most commonly $10^{-6}\frac{\text{mol}}{\text{L}}$.

We now have a reasonable model which allows us to predict the melting temperature of an oligonucleotide given its composition. We will later use this model to estimate the melting temperatures of desired (i.e. between oligonucleotides and their respective target DNA) and undesired hybridizations. The goal will be to choose the melting temperature and the oligonucleotides in a way that maximizes the desired hybridizations while minimizing undesired hybridizations.

Chapter 3

Suffix Arrays and Related Structures

3.1 Introduction

When we design oligonucleotides for DNA microarrays to find a particular sequence, for example a gene or a whole genome, one of the main tasks is to check if a candidate oligonucleotide is specific to the gene we want to identify or if it occurs somewhere else which would mean that we cannot use the oligonucleotide. Since hybridization can also occur with imperfect matches, we must also look for sequences which do not match exactly but which have a high similarity. Since the data we have to process can be quite large, it is essential to use an efficient search algorithm to perform these searches. We will use suffix arrays for this purpose which are more space-efficient than suffix trees and achieve almost the same performance.

3.2 Basic Definitions and Concepts

Before we can look at the exact details of the suffix array construction algorithm, we need to define some concepts which we will later use. The most basic definition is how we represent DNA as a string.

Strings over the DNA alphabet. Let $\Sigma := A, C, G, T$ be the DNA alphabet. Sometimes additional characters such as a wildcard character (usually denoted as X) are introduced, however we will not consider these, thus all input data must only contain the characters in Σ. We use the character order $A < C < G < T$. In the implementation of the algorithms sometimes we use integers instead of characters and then we code the A as a 1, the C as a 2, the G as a 3 and the T as a 4. We write Σ^+ for the set of all non-empty strings consisting of characters from Σ. For a string $s = (s_0, \ldots, s_{|s|-1})$, we use the notation $s_{(i)} := s_{i \ldots |s|-1}$ to denote the suffix of s starting at position i.

Definition (Longest Common Factor) A common factor of two strings s and t is a string that is both a substring of s and of t. A common factor is a longest common factor if no longer common factor exists. We write $lcf(s, t)$ for the length of the longest common factor of s and t. This is sometimes also called the shared length of two strings.

For a string s, let s^p be the prefix consisting of the first p symbols of s if s contains more than p symbols, and s otherwise.

Definition (Longest Common Prefix) The longest common prefix of two strings s and t is the longest string that is both a prefix of s and t. We write $lcp(s,t) := max_{0 \leq i < min(|s|,|t|)}\{i : s^i = t^i\}$ for its length.

Definition (Suffix Array) Let $s = (s_0, \ldots, s_{n-1}) \in \Sigma^+$ be a sequence of length n. Let $s_{(i)}$ be the suffix of s starting at position i. The suffix array of s is an array $\mathsf{p} = (\mathsf{p}[0], \ldots, \mathsf{p}[n])$ of length $n+1$ that contains the starting positions of the suffixes of s in lexicographical order, i.e. $s_{\mathsf{p}[0]} < s_{\mathsf{p}[1]} < \ldots < s_{\mathsf{p}[n]}$.

Definition (Longest Common Prefix (lcp) Array) Let $s = (s_0, \ldots, s_{n-1}) \in \Sigma$ be a sequence of length n and p be the suffix array of s. Then lcp is an array of length $n+1$, where $\mathsf{lcp}[i]$ is the length of the longest common prefix of $s_{(\mathsf{p}[i-1])}$ and $s_{(\mathsf{p}[i])}$ (for $i > 0$) and $\mathsf{lcp}[0] = 0$.

Definition (h-gram) Let a be a string of length n. We will call the h-gram at a_i the string $(a_i, a_{i+1}, \ldots, a_{i+h-1})$ with subscripts taken *mod* n, except that for $h = 0$ the “0-gram” at a_i is simply a_i. Or more succinctly, the h-gram at a_i is $(a_i, \ldots, a_{i+max(h,1)-1})$.

Definition (h-successor) Let a again be a string of length n. The h-successor of the h-gram at a_i is the h-gram at a_{i+h} again with the subscripts taken *mod* n.

Definition (q-Bucket) Let x be an array which encodes substrings of a string s (for example a suffix array). A q-Bucket is an interval $[l, r]$ such that $\mathsf{lcp}_l < q, \mathsf{lcp}_{r+1} < q$ and $\mathsf{lcp}_i \geq q$ for all $i = l+1, \ldots, r$.

Definition ($<_p, \leq_p, =_p, \neq_p, >_p, \geq_p$) We define the relation $<_p$ to be the lexicographical order of p-symbol prefixes; that is, $u <_p v$ if and only if $u^p < v^p$. The relations $\leq_p, =_p, \neq_p, >_p$ and $\geq_p$ are defined in a similar way.

3.3 Suffix Sort

3.3.1 Suffix Sort Introduction

We first construct a suffix array from a given sequence which we can later use to perform efficient searches and other operations on that sequence. As we have seen, a suffix array is simply a lexicographically sorted array of all suffixes of the sequence. Note that suffix sort and suffix array construction mean the same.

To start off, we will look at an algorithm to construct a suffix array which will form the basis of our further considerations and which we will further refine and enhance subsequently. Our suffix sort algorithm follows closely the ideas presented in [MM93] and [MM97].

The input of our algorithm is a DNA sequence (or, to be precise, an integer array; we use the terms array, string and sequence interchangeably for a data structure which holds DNA sequence information) a and its length n. We will use an integer coding, where the nucleotide A is coded as a 1, C as a 2, G as a 3 and T as a 4. We will modify the elements of this array in the algorithm, so it is necessary to make a copy before calling the algorithm. The array a may only consist of integers $1 \leq x \leq 4$ except the end of the array needs to be marked with a 0, i.e. $a_{n-1} = 0$.

The following data structures will be used during computation of the suffix array:

- a: Array of h-gram codes. Initially holds the array for which we want to find the suffix array. The code a_i encodes the h-gram at the i^{th} position of the original array a.
- al: Linked-list area (or list "body"), no initial value.
- p: Permutation of original array a. Initially the identity permutation.
- pl: List heads, no initial value.
- k: Variable with the number of buckets we have in the current iteration. The array a can contain values in the range $0 \dots k-1$. Initially $k = 5$ since elements in a can be in the range $0 \dots 4$ before we start the algorithm.
- BUCK: Flag at the beginning of "buckets". Used for elements of p.

Note: All the arrays (i.e. a, al, p and pl) have length n.

Precondition of the algorithm: The array a holds n values with $n \geq 1$. Each element is in the range $1 \dots 4$ except the last element a_{n-1} which has value 0 and serves as an end marker. The values of the array represent a DNA sequence.

Postcondition of the algorithm: The array p is the suffix array of a.

We will order 2h-grams for $h = 0, 1, 2, 4, 8, 16, \dots$ by iteratively performing these four steps (the individual steps will be detailed further down):

1. **Lists creation.** Construction of linked lists of like-valued elements of a ordered in reverse order of their h-successors as given by p. Save the resulting list headers in pl and the list "bodies" in al.
2. **Sort.** Make the elements of the array p point to the elements of a in order of increasing value. This is done in reverse, i.e. we start with the biggest values. Mark the first element of each h-bucket in p with the BUCK flag to demarcate the buckets.
3. **Refinement.** Refine the buckets by placing a BUCK mark on each p_i where the h-successors of a_{p_i} and $a_{p_{i-1}}$ differ.
4. **Recode.** Recode a according to the new buckets. Count the new number of buckets and assign that value to k.

Keep in mind that this is just a rough overview of the algorithm, we look at each step more closely in the subsequent sections. Let us look at an example. Suppose we want to find the suffix array of the following array:

$$\mathsf{a}_{initial} = (1, 4, 3, 1, 2, 4, 1, 2, 1, 4, 2, 1, 1, 4, 3, 2, 4, 1, 3, 4, 0)$$

As we already stated above, p is initially the identity permutation:

$$\mathsf{p}_{initial} = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)$$

Hence, in our case n=21 which is also the length of a and p. So in the first iteration h is equal to 0. After the first iteration a is still unchanged and p is a permutation of a which simply sorts the elements:

$$\mathsf{p} = (20, 0, 3, 6, 8, 11, 12, 17, 4, 7, 10, 15, 2, 14, 18, 1, 5, 9, 13, 16, 19)$$

This permutation puts the elements of a in this order:

$$(0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4)$$

After the second iteration (where $h = 1$)

$$\mathsf{a} = (4, 13, 7, 2, 6, 11, 2, 5, 4, 12, 5, 1, 4, 13, 8, 6, 11, 3, 9, 10, 0)$$

and

$$\mathsf{p} = (20, 11, 3, 6, 17, 0, 8, 12, 7, 10, 4, 15, 2, 14, 18, 19, 5, 16, 9, 1, 13)$$

We will now look in detail at the third iteration in which h is equal to 2.

3.3.2 Suffix Sort Step 1: Lists Creation

State before step 1. Each element a_i of the array a encodes an h-gram. We are in the third iteration i.e. $h = 2$ (remember that $h = 0, 1, 2, 4, 8, \ldots$, it is increased at the beginning of each iteration, i.e. before step 1). a has this value:

$$\mathsf{a} = (4, 13, 7, 2, 6, 11, 2, 5, 4, 12, 5, 1, 4, 13, 8, 6, 11, 3, 9, 10, 0)$$

Note that this a is not equal to the initial a. Also this a has values which are greater than four (unlike the initial a). It will later be seen how this happens. In the example, a would encode the following 2-grams:

$$\begin{array}{l} \mathsf{a}_0 = 4 \text{ encodes } (1,4) \\ \mathsf{a}_1 = 13 \text{ encodes } (4,3) \\ \mathsf{a}_2 = 7 \text{ encodes } (3,1) \\ \mathsf{a}_3 = 2 \text{ encodes } (1,2) \\ \mathsf{a}_4 = 6 \text{ encodes } (2,4) \\ \mathsf{a}_5 = 11 \text{ encodes } (4,1) \\ \ldots \end{array}$$

Furthermore, when $h > 0$, p (the permutation of a) lists h-grams in lexicographical order. If $h = 0$, p contains the input order (i.e. the identity permutation: $\mathsf{p} = (0, 1, \ldots, n)$). As we already stated, in our example p has this value:

$$\mathsf{p} = (20, 11, 3, 6, 17, 0, 8, 12, 7, 10, 4, 15, 2, 14, 18, 19, 5, 16, 9, 1, 13)$$

and therefore puts the 2-grams in the following order (remember that we take the subscripts *mod n* and therefore also p_0 which points to the last element of a, namely a_{20}, encodes a 2-gram of length two):

(0,1)	(2,4)
(1,1)	(3,1)
(1,2)	(3,2)
(1,2)	(3,4)
(1,3)	(4,0)
(1,4)	(4,1)
(1,4)	(4,1)
(1,4)	(4,2)
(2,1)	(4,3)
(2,1)	(4,3)
(2,4)	

Figure 3.1 shows again the relation between a and p.

Step 1. We construct linked lists of like-valued codes in a ordered in the reverse order of their h-successors as given by p. For our example, at the end of this step we would have these 14 linked lists:

$$\begin{array}{l}
0 \longrightarrow \mathsf{a}_{20} \\
1 \longrightarrow \mathsf{a}_{11} \\
2 \longrightarrow \mathsf{a}_{3} \longrightarrow \mathsf{a}_{6} \\
3 \longrightarrow \mathsf{a}_{17} \\
4 \longrightarrow \mathsf{a}_{12} \longrightarrow \mathsf{a}_{0} \longrightarrow \mathsf{a}_{8} \\
5 \longrightarrow \mathsf{a}_{7} \longrightarrow \mathsf{a}_{10} \\
6 \longrightarrow \mathsf{a}_{15} \longrightarrow \mathsf{a}_{4} \\
7 \longrightarrow \mathsf{a}_{2} \\
8 \longrightarrow \mathsf{a}_{14} \\
9 \longrightarrow \mathsf{a}_{18} \\
10 \longrightarrow \mathsf{a}_{19} \\
11 \longrightarrow \mathsf{a}_{16} \longrightarrow \mathsf{a}_{5} \\
12 \longrightarrow \mathsf{a}_{9} \\
13 \longrightarrow \mathsf{a}_{13} \longrightarrow \mathsf{a}_{1}
\end{array}$$

This means for example that the elements a_{12}, a_0 and a_8 have value 4, that the element a_{11} is the only one with value 1 and so forth. We store the list heads in pl and the list areas in al, we therefore have:

$$\begin{array}{l}
\mathsf{pl}_0 = 20 \\
\mathsf{pl}_1 = 11 \\
\mathsf{pl}_2 = 3, \mathsf{al}_3 = 6 \\
\mathsf{pl}_3 = 17 \\
\mathsf{pl}_4 = 12, \mathsf{al}_{12} = 0, \mathsf{al}_0 = 8 \\
\mathsf{pl}_5 = 7, \mathsf{al}_7 = 10 \\
\mathsf{pl}_6 = 15, \mathsf{al}_{15} = 4 \\
\mathsf{pl}_7 = 2 \\
\mathsf{pl}_8 = 14 \\
\mathsf{pl}_9 = 18 \\
\mathsf{pl}_{10} = 19 \\
\mathsf{pl}_{11} = 16, \mathsf{al}_{16} = 5 \\
\mathsf{pl}_{12} = 9 \\
\mathsf{pl}_{13} = 13, \mathsf{al}_{13} = 1
\end{array}$$

As already stated, we iterate over the elements in a in a very particular order and push each element to the front of the corresponding list. We iterate over

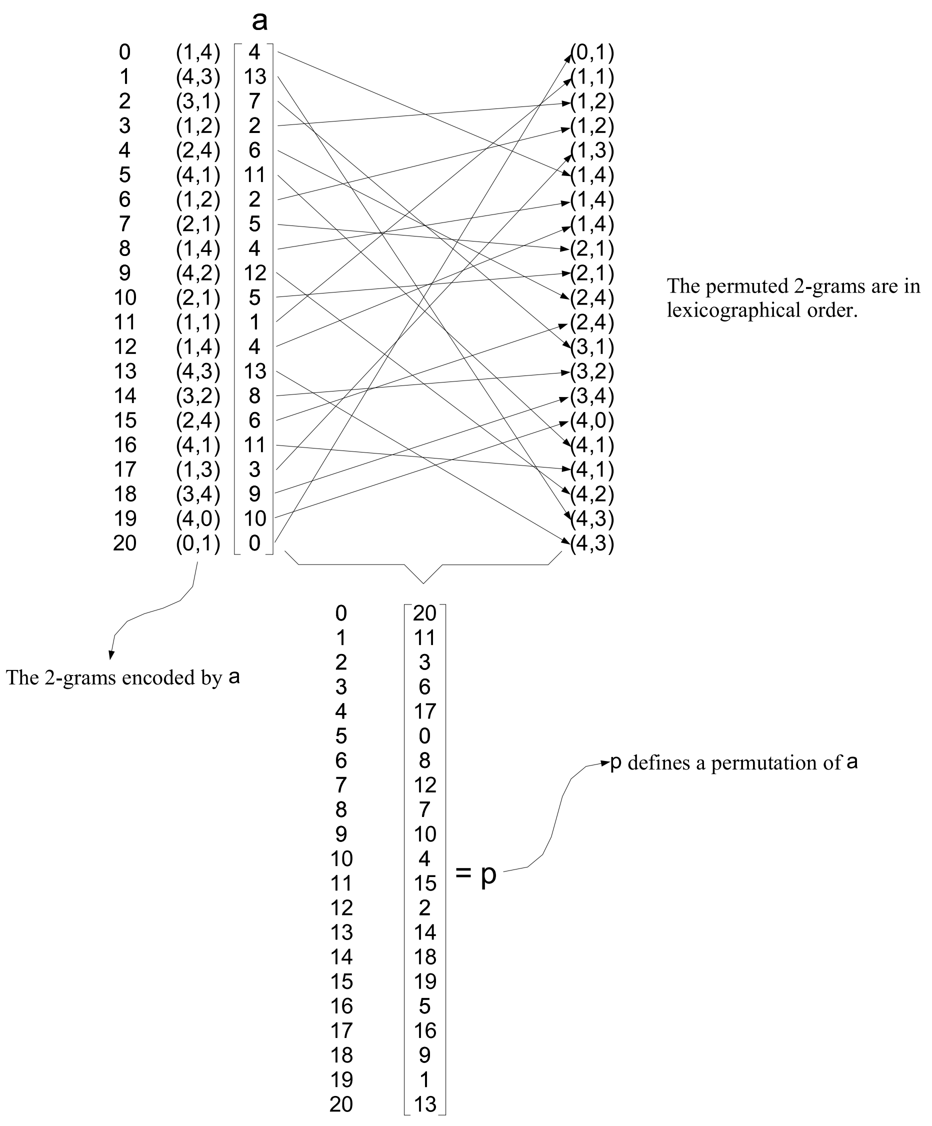

Figure 3.1: Relation between a and p before a new iteration, here $h = 2$ and a as in the example.

p from $p_0 \dots p_{n-1}$. Each element p_i points to an element of a. We retrieve the h-predecessor of this element and put it to the front of the corresponding linked list. Let us look again at our example. Consider we start iterating over p and we look at p_0. It has the value 20 and thus points to a_{20}. Now we want the h-predecessor of a_{20} which is a_{18}, since $h = 2$. Remember that the h-predecessor of a_i is simply a_{i-h} with the subscripts taken mod n. The value of a_{18} is 9. To push a_{18} to the front of the list of elements with value 9 we let the list head pl_9 point to a_{18} by assigning the value 18. We let al_{18} point to the old first element of the list by assigning the old value of pl_3 to al_{18}. This is actually not necessary in this particular case since the list was empty before.

As defined before, the integer variable k contains the current maximum value that elements of a can contain, in the example $k = 14$.

The following C code snippet shows how this can be done:

```
for (i=0; i<n; i++) {
  j = p[i];
  j = predecessor(j, h);
  l = a[j];
  al[j] = pl[l];
  pl[l] = j;
}
```

There is one problem with the given solution. How do we know where the linked lists end? This can be resolved by flagging the elements which are the last elements of a linked list. A good solution is to use the sign bit since we only deal with positive integer values.

It is obvious that the two arrays al and pl contain exactly the same information as a and p. It would therefore be sensible to use the same arrays for both representations of the data and to convert the data "in place". We will show later that this can be done, but for the moment we suppose that we duplicate the data and thus maintain all four arrays.

3.3.3 Suffix Sort Step 2: Sort

State before step 2. We have k linked lists, with the list heads in pl and the list areas in al.

Step 2. In this step we make p point to the elements of a in the order given by the linked lists. We assign the values starting from p_{n-1} and then move down to p_0. We start with the list header pl_k. Since the elements in the linked lists are already in reverse order we can process the linked lists in ascending order while assigning the values to p in descending order. In our example, this results in the following assignments, in this particular order:

$\mathsf{p}_{20} = \mathsf{pl}_{13} = 13$
$\mathsf{p}_{19} = \mathsf{al}_{13} = 1$
$\mathsf{p}_{18} = \mathsf{pl}_{12} = 9$
$\mathsf{p}_{17} = \mathsf{pl}_{11} = 16$
$\mathsf{p}_{16} = \mathsf{al}_{16} = 5$
$\mathsf{p}_{15} = \mathsf{pl}_{10} = 19$
$\mathsf{p}_{14} = \mathsf{pl}_{9} = 18$
$\mathsf{p}_{13} = \mathsf{pl}_{8} = 14$
$\mathsf{p}_{12} = \mathsf{pl}_{7} = 2$
$\mathsf{p}_{11} = \mathsf{pl}_{6} = 15$
$\mathsf{p}_{10} = \mathsf{al}_{15} = 4$
$\mathsf{p}_{9} = \mathsf{pl}_{5} = 7$
$\mathsf{p}_{8} = \mathsf{al}_{7} = 10$
$\mathsf{p}_{7} = \mathsf{pl}_{4} = 12$
$\mathsf{p}_{6} = \mathsf{al}_{12} = 0$
$\mathsf{p}_{5} = \mathsf{al}_{0} = 8$
$\mathsf{p}_{4} = \mathsf{pl}_{3} = 17$
$\mathsf{p}_{3} = \mathsf{pl}_{2} = 3$
$\mathsf{p}_{2} = \mathsf{al}_{3} = 6$
$\mathsf{p}_{1} = \mathsf{pl}_{1} = 11$
$\mathsf{p}_{0} = \mathsf{pl}_{0} = 20$

Furthermore we set the `BUCK` flag for each element that starts a new bucket (i.e. the list heads). In the example above, this would be p_{20}, p_{18}, p_{17}, p_{15}, p_{14}, p_{13}, p_{12}, p_{11}, p_{9}, p_{7}, p_{4}, p_{3}, p_{1} and p_{0}.

3.3.4 Suffix Sort Step 3: Refinement

State before step 3. Permutation `p` lists h-grams in lexicographical order. Furthermore it is bucketed by h-grams. Within each bucket h-grams are ordered by their h-successors. The array `a` which we will use again in step 3 is still the same as in step 1. Following our example, table 3.1 shows the 2-grams encoded by `p` and the corresponding 2-successors and it can be easily seen that the 2-grams are sorted by their 2-successors within each bucket.

Step 3. We refine the buckets by values of h-successors in `a`. This means that we iterate over all elements of `p` and mark each element which differs from its h-successor with a `BUCK` flag to indicate the beginning of a new bucket. The following C code snippet shows how this can be implemented:

```
for (i=0, j=-1; i<n; i++, j=l) {
  l = a[successor(pos[i] & ~BUCK, h)];
  if (l != j)
    pos[i] |= BUCK;
}
```

Since in our example the 2-successors within a bucket always differ from each other, each element of `p` will get the `BUCK` flag.

3.3.5 Suffix Sort Step 4: Recode

State before step 4. The array `p` holds an ordered, bucketed list of 2h-grams. Table 3.2 shows `p` and the refined buckets for our example.

p			a			2-gram	2-successor	`BUCK` flag
p_0	=	20	a_{20}	=	0	(0,1)	(4,3)	x
p_1	=	11	a_{11}	=	1	(1,1)	(4,3)	x
p_2	=	6	a_6	=	2	(1,2)	(1,4)	x
p_3	=	3	a_3	=	2	(1,2)	(4,1)	
p_4	=	17	a_{17}	=	3	(1,3)	(4,0)	x
p_5	=	8	a_8	=	4	(1,4)	(2,1)	x
p_6	=	0	a_0	=	4	(1,4)	(3,1)	
p_7	=	12	a_{12}	=	4	(1,4)	(3,2)	
p_8	=	10	a_{10}	=	5	(2,1)	(1,4)	x
p_9	=	7	a_7	=	5	(2,1)	(4,2)	
p_{10}	=	4	a_4	=	6	(2,4)	(1,2)	x
p_{11}	=	15	a_{15}	=	6	(2,4)	(1,3)	
p_{12}	=	2	a_2	=	7	(3,1)	(2,4)	x
p_{13}	=	14	a_{14}	=	8	(3,2)	(4,1)	x
p_{14}	=	18	a_{18}	=	9	(3,4)	(0,1)	x
p_{15}	=	19	a_{19}	=	10	(4,0)	(1,4)	x
p_{16}	=	5	a_5	=	11	(4,1)	(2,1)	x
p_{17}	=	16	a_{16}	=	11	(4,1)	(3,4)	
p_{18}	=	9	a_9	=	12	(4,2)	(1,1)	x
p_{19}	=	1	a_1	=	13	(4,3)	(1,2)	x
p_{20}	=	13	a_{13}	=	13	(4,3)	(2,4)	

Table 3.1: 2-grams and 2-successors encoded by p in the example. The horizontal lines demarcate the buckets.

p	a	2-gram	2-successor	`BUCK` flag
$p_0 = 20$	$a_{20} = 0$	(0,1)	(4,3)	x
$p_1 = 11$	$a_{11} = 1$	(1,1)	(4,3)	x
$p_2 = 6$	$a_6 = 2$	(1,2)	(1,4)	x
$p_3 = 3$	$a_3 = 2$	(1,2)	(4,1)	x
$p_4 = 17$	$a_{17} = 3$	(1,3)	(4,0)	x
$p_5 = 8$	$a_8 = 4$	(1,4)	(2,1)	x
$p_6 = 0$	$a_0 = 4$	(1,4)	(3,1)	x
$p_7 = 12$	$a_{12} = 4$	(1,4)	(3,2)	x
$p_8 = 10$	$a_{10} = 5$	(2,1)	(1,4)	x
$p_9 = 7$	$a_7 = 5$	(2,1)	(4,2)	x
$p_{10} = 4$	$a_4 = 6$	(2,4)	(1,2)	x
$p_{11} = 15$	$a_{15} = 6$	(2,4)	(1,3)	x
$p_{12} = 2$	$a_2 = 7$	(3,1)	(2,4)	x
$p_{13} = 14$	$a_{14} = 8$	(3,2)	(4,1)	x
$p_{14} = 18$	$a_{18} = 9$	(3,4)	(0,1)	x
$p_{15} = 19$	$a_{19} = 10$	(4,0)	(1,4)	x
$p_{16} = 5$	$a_5 = 11$	(4,1)	(2,1)	x
$p_{17} = 16$	$a_{16} = 11$	(4,1)	(3,4)	x
$p_{18} = 9$	$a_9 = 12$	(4,2)	(1,1)	x
$p_{19} = 1$	$a_1 = 13$	(4,3)	(1,2)	x
$p_{20} = 13$	$a_{13} = 13$	(4,3)	(2,4)	x

Table 3.2: The refined buckets in the example.

Step 4. In this final step of an iteration, a is recoded according to the new buckets. In particular, for each a_i, $\mathsf{a}_i = k$ where k indicates the current bucket number. Initially k is minus one and it is incremented each time a BUCK flag is encountered in p as we go through the elements in p. Therefore the elements in a are in the range $0 \ldots k$. after this step. In addition we increment k once more such that it is equal to the number of buckets. In our example, the recoded a is

$$\mathsf{a} = (6, 19, 12, 3, 10, 16, 2, 9, 5, 18, 8, 1, 7, 20, 13, 11, 17, 4, 14, 15, 0)$$

and the C source for a possible implementation of this step looks like this:

```
for (i=0, k=-1; i<n; i++) {
  if (pos[i] & BUCK)
    k++;
  a[pos[i] & ~BUCK] = k;
}
if (++k >= n)
  break;
```

At the end of each iteration we check if $k = n$. If this equality holds true, this means that each bucket holds only one element and we have found the suffix array. In our example, at the end of this step $k = 21$ which means that the algorithm has finished. We do not need the array a anymore and the suffix array is p:

$$\mathsf{p} = (20, 11, 6, 3, 17, 8, 0, 12, 10, 7, 4, 15, 2, 14, 18, 19, 5, 16, 9, 1, 13)$$

which puts the suffixes in the following order

0
1143241340
121421143241340
124121421143241340
1340
1421143241340
143124121421143241340
143241340
21143241340
21421143241340
24121421143241340
241340
3124121421143241340
3241340
340
40
4121421143241340
41340
421143241340
43124121421143241340
43241340

3.3.6 Space Requirement Reduction

In this section, it will be shown how the space requirement can be reduced from $4n$ to $2n$. This comes not without a price in the form of a slight performance penalty. The algorithm presented in the preceding sections already paved the way for this improvement. As already mention, the arrays p and a basically contain the same information as the arrays pl and al. Let us look at the usage of those arrays:

Step 1. Construction of al and pl from a and p.

Step 2. We use al and pl to fill the array p

Step 3. We add additional BUCK flags in p. We use a to look up h-gram codes.

Step 4. We recode a by using p.

Now consider that al and a share one array and also p and pl share one array. In step one this means that the array p holds two different types of data:

- Elements of p
- Elements of pl

a holds three different types of data:

- Elements of a
- List links of al
- List ends

It was already suggested in the discussion of step one that a flag should be used to mark the end of the linked lists. A flag ORIG will now be used instead on the elements of p to mark the elements which have not yet been processed. So before step one, we mark all elements of p with this ORIG flag. As before we push the elements to the list fronts in the order given by the h-successors in p. As we push an element to the respective list front, we remove the ORIG flag. So at the end of this step, only the elements in p which are at the end of a linked list still have the ORIG flag set. The following code snippet implements the refined step one:

```
for (i=0; i<n; i++) {
  for (j=pos[i]; !(j&ORIG); j=al[j])
  j = pred(j&~ORIG,h);
  l = a[j];
  al[j] = posl[l];
  posl[l] = j;
}
```

Besides the removal of the ORIG flag there is an additional change with respect to the original version. There is an additional for-loop. It is necessary to take care of the lists ends. Consider for example the case where we are in the first iteration, i.e. p is the identity permutation. Furthermore say $a_0 = 1$. Now for $i = 0$ we would assign the value of zero to pl_1. Thus this element will be the last in the linked list of elements with value one. But also p_1 has changed and no longer carries the `ORIG` flag. We must not process this element again and this is exactly what the for-loop does. So after the first step the array a and p have been overwritten by al and pl.

During the second step there can be no collisions between the arrays because we start with the biggest values. New values fill in the n-size array p from the top while the size-k array pl of list heads shrinks towards the bottom. Since no list is empty, the two uses of p will not collide.

In the original version of the algorithm, the next step would be to place `BUCK` marks in p to refine the buckets. But it is necessary to look up the h-successors in a. Because a has been overwritten, it needs to be reconstructed first. The third step will be divided into two sub-steps:

Step 3a: Reconstruction. Reconstruction of a from the buckets in p.

Step 3b: Refinement. The bucket refinement as described in the initial algorithm.

In the fourth step we only use a and p which we still have from the third step. Note that the reconstruction phase in step 3a is exactly the same as in step four. One additional task that needs to be done is to mark all elements of p with the `ORIG` flag for the first step of the next iteration.

3.4 Bucket Arrays

A suffix array is a great help to reduce the time to search for a short sequence in a long sequence. But it is still necessary to look through the suffix array to see if there is a matching suffix, for example by performing a binary search. In order to further reduce the search time, the concept of the bucket array is introduced. The bucket array contains the left endpoints of all q-buckets that occur in the sequence.

Definition (Bucket (bck) array) Define a code $\langle a \rangle$ for each letter $a \in \Sigma$ as follows: Let $\langle A \rangle = 1, \langle C \rangle = 2, \langle G \rangle = 3, \langle T \rangle = 4$. For a q-gram $Q = (Q_0,\dots,Q_{q-1}) \in \Sigma^q$, let $\langle Q \rangle = \sum_{i=0}^{q-1} 4^{q-1-i}(\langle Q_i \rangle - 1)$. For $c = 0, \dots, 4^q - 1$, let bck_c be the index in the suffix array of the left end l of the q-bucket $[l, r]$ of the q-gram Q with $\langle Q \rangle = c$ if Q exists in the suffix array string. If Q does not exist in the suffix array, set $\mathsf{bck}_c = \infty$.

Note that the bucket array element denotes the left end of the corresponding q-bucket. To find the other matches, we can simply look at the following suffixes in the suffix array.

Consider the following example. We want to construct a bucket array of q-grams with length five, therefore $q = 5$ and hence the length of the bucket array is $4^q = 4^5 = 1024$. Now say Q=`ACGTA`, according to the above definition $\langle Q \rangle = 4^4(\langle A \rangle - 1) + 4^3(\langle C \rangle - 1) + 4^2(\langle G \rangle - 1) + 4^1(\langle T \rangle - 1) + 4^0(\langle A \rangle - 1) = 108$. If Q occurs in the suffix array pos at position i, $\mathsf{bck}_{108} = i$. There must be no other occurrence of Q in the sequence which is more to the left, i.e. if there is another occurrence of Q at pos_j, then $\mathsf{pos}_j > \mathsf{pos}_i$ must hold true.

Before the bucket array of a particular sequence can be constructed, q must be fixed. The bucket array can be constructed at the same time as the suffix array and the lcp array of the background sequences. These data structures will subsequently be used to search for prefixes of candidate oligonucleotides in the background sequences. These concepts will be introduced later, but at this point it is important to see the impact of the choice of q. Since all matches between a candidate oligonucleotide and a suffix where the common prefix is smaller than q will be ignored, relevant matches might be lost if q is chosen too small. Furthermore it is desirable to have a large q in order to keep the buckets (i.e. the number of matching suffixes) small. This is beneficial because otherwise for each lookup we have to look through a large number of suffixes in the suffix array which we wanted to avoid with the introduction of the bucket array in the first place. On the other hand, a small q is advantageous for the time and space requirements of the bucket array construction. In [Rah02] Rahmann *et al.* suggest $q = \lfloor \log_4 \frac{n}{128} \rfloor$ as a good default value for q, where n is the length of the suffix array. With this choice of q, the expected size of a bucket is constant for all buckets.

3.5 Enhanced Suffix Arrays

In the preceding sections we have seen three key data structures which will later play a central role in the oligonucleotide selection process. All three data structures are integer arrays which can be built taking a sequence as an input.

Suffix Array A sorted list of all suffixes of the sequence. The length is the same as the length of the input sequence.

LCP Array An array of the longest common prefixes between two consecutive elements in the suffix array. The length is the same as the length of the input sequence.

Bucket array Array of the left endpoints of q-buckets. The length is 4^q and we choose $q = \lfloor \log_4 \frac{n}{128} \rfloor$.

Since all these three arrays are closely related to each other and can be built conveniently at the same time, we call them together as a whole the *enhanced suffix array* of a sequence.

Chapter 4

Probe Selection with Suffix Arrays

4.1 Overview

Most existing oligonucleotide selection algorithms take a set of sequences as an input. Initially there is a large pool of candidate oligonucleotides from all those input sequences. Subsequently the number of candidate oligonucleotides is narrowed down by applying various selection criteria. While some of those criteria are rather simple such as self-complementarity of the oligonucleotides, others can be very costly both in terms of time and memory requirements.

The algorithm which is presented here allows for the identification of genes in the sample under examination (i.e. in the target). Possible applications include the measurement of gene expression, gene detection or classification. The algorithm divides the target sequences in two classes:

Master sequences. These are the sequences we wish to detect. In the case we want to measure the expression of a particular gene, this gene would be the master sequence. The information gained from the experiment will be whether or not the master sequences were detected in the target sequences. There can be one or more master sequences.

Background sequences. The background sequences help to improve the quality of the oligonucleotides. Consider again the case where we want to measure the expression of a gene. Here we would choose the genes from which we know that they are expressed in the target as the background sequences. We want to choose the master sequences as specific as possible with respect to the background sequences. Since it is possible that there are genes in the background sequences which are very similar to the ones in the master sequences, the quality of the resulting oligonucleotides would be very low if we would choose the oligonucleotides just based on the sequences we are interested in.

Consider the following example. Say the first sequence is our master sequence, the other two sequences are background sequences.

```
Sequence 1: 5'-GCTTCGATGCTAAAGCTAAAA-3'
Sequence 2: 5'-AFGGCTTCCGTAGCTAGCTAA-3'
Sequence 3: 5'-AGCTAAAGCCATAGCAT-3'
```

Now if we would choose our oligonucleotides by simply looking at the first sequence, we do not know if it is specific or not. In fact, the beginning as well as the end of the sequence is not specific, as we can see:

```
Sequence 1:    5'-GCTTCGATGCTAAAGCTAAAA-3'
                  |||||
Sequence 2: 5'-AFGGCTTCCGTAGCTAGCTAA-3'
```

and furthermore:

```
Sequence 1: 5'-GCTTCGATGCTAAAGCTAAAA-3'
                         |||||
Sequence 2:   5'-AFGGCTTCCGTAGCTAGCTAA-3'
                         |||||
Sequence 3:              5'-AGCTAAAGCCATAGCAT-3'
```

So clearly, it would be sensible to not include the sequences that are shared by sequence one and any of the other two sequences in our oligonucleotides since we are interested in the presence of the first sequence only. Another way to look at the background sequences would be to look at the master sequence as the “signal” we want to examine and at the background sequences as the “noise”. The more knowledge we have about the “noise”, the better we can increase the “signal-to-noise ratio”.

We are always choosing oligonucleotides for one master sequence at a time. So in the case where we have more than one master sequence, every master sequence except the one which is currently “under examination” is also a background sequence. The goal of the algorithm is to find oligonucleotides which are as specific as possible. Now of course, “specificity” can be defined in a number of ways and we will later introduce our definition of specificity. The steps involved in the oligonucleotide selection algorithm are as follows:

1. Read input sequences and append to each sequence its reverse complement.
2. Compute the enhanced suffix array for all sequences.
3. Do the following for each master sequence (during each iteration, we just have one current master sequence and all other sequences are background sequences because we process only one master sequence at a time):
   (a) Compute melting temperature for each candidate oligonucleotide.
   (b) Determine self-complementarity of candidate oligonucleotides and remove substantially self-complementary oligonucleotides from candidate set.
   (c) Compute LCF statistics against background sequences for each remaining candidate oligonucleotide.
   (d) Compute unspecificity of all remaining candidate oligonucleotides taking into account the LCF statistics and melting temperature.

   (e) Create a ranking of all candidate oligonucleotides sorted by unspecificity.

   (f) Remove overlapping candidate oligonucleotides.

   (g) Choose the best oligonucleotides (i.e. the ones with the lowest unspecificity) for the current master sequence.

4. Determine lowest oligonucleotide melting temperature.

This algorithm is inspired by the one presented in [Rah02] but it contains several modifications such as the inclusion of the self-complementarity criterion. We formally define a master sequence as follows:

Definition (Master sequence) A master sequence is a 4-tuple $M = (m, s, l, T)$ where

- m is a non-negative integer (the index number of the sequence)
- $s = s_1, \ldots, s_{|s|}$ is a sequence of length $|s|$ with $s_i \in \sum \forall i$
- l is the desired oligonucleotide length and is > 0
- $T = (T_1, \ldots, T_{|s|})$ is a vector of non-negative integers. If $T_i = 0$ it means that we have eliminated the oligonucleotide $s_i \ldots s_{i+l}$ from consideration. Otherwise (i.e. if $T_i > 0$) T_i is the melting temperature of the oligonucleotide $s_i \ldots s_{i+l}$.

In the first step we read all the input sequences (normally from a FASTA file). We assign an index number to each sequence. Since the target sequences are double-stranded, we will also have to take the reverse complements into account. So we will append to each sequence read its reverse complement. In the next step we compute the enhanced suffix array for each sequence. The following substeps are performed for each target sequence. We first build a master sequence from the current target sequence and compute the melting temperatures for each candidate oligonucleotide. We remove the candidate oligonucleotides which include the "concatenation point" between sequence and reverse complement from the candidate set.

4.2 Self-complementarity

Candidate oligonucleotides which are substantially self-complementary are removed from the candidate set. We use the method suggested by Li and Stormo in [LS00] which is basically a simplified global alignment with cost-free ends. Consider the following candidate oligonucleotide

```
GGGGGGTATA
```

We now compare the oligonucleotide and its complement by filling in a matrix. For equal bases the matrix value is one and zero otherwise.

$$
\begin{array}{c|cccccccccc}
 & \mathtt{G} & \mathtt{G} & \mathtt{G} & \mathtt{G} & \mathtt{G} & \mathtt{G} & \mathtt{T} & \mathtt{A} & \mathtt{T} & \mathtt{A} \\
\mathtt{T} & 0 & 0 & 0 & 0 & 0 & 0 & \underline{1} & 0 & 1 & 0 \\
\mathtt{A} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \underline{1} & 0 & 1 \\
\mathtt{T} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \underline{1} & 0 \\
\mathtt{A} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \underline{1} \\
\mathtt{C} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathtt{C} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathtt{C} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathtt{C} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathtt{C} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathtt{C} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

Of course we could also align the oligonucleotide with its reverse and then look for complementary bases, but it is simpler to take the reverse complement and then look for equal bases. After the matrix is built, we search for the longest diagonal sequence of ones which is underlined in the example above. This corresponds to the following alignment:

```
5'-GGGGGGTATA-3'
         ||||
      3'-ATATGGGGGG-5'
```

If the self-complementarity exceeds a certain threshold (usually 40% of the oligonucleotide length) the oligonucleotide is removed from the candidate set.

4.3 LCF Profile and LCF Statistics

We compute the LCF statistics for the current master sequence against the background sequences. The LCF statistics will provide information about the specificity of the candidate oligonucleotides with respect to the background sequences. The LCF statistics is based on the longest common factor (LCF) between sequences. Let us first look why the LCF is a good basis for an unspecificity measure. Consider the following example:

```
ATCTCCACCCGGAGCTTGTTCAT  ATCTCCACCCGGAGCTTGTTCAT
||| |||||| |||| ||| |||  |||||||||||||||||||
ATCACCACCCTGAGCGTGTCCAT  ATCTCCACCCGGAGCTTGTCAGG
1) lcf=6, matches=19     2) lcf=19, matches=19

ATCTCCACCCGGAGCTTGTTCAT
|||||||||| ||||||||||||
ATCTCCACCCTGAGCTTGTTCAT
3) lcf=12, matches=22
```

If we would choose the number of matches instead of the LCF as the basis of our unspecificity measure, we would exclude the first sequence from our candidate oligonucleotides because the number of matches is quite high (19 out of 23). But this is overly pessimistic since this duplex is not very stable. The LCF reflects this much better. The number of matches of both the first and the second duplex is 19, but the second duplex has a higher melting temperature because of the many consecutive matches. It is therefore more stable than the first duplex. There are some rare cases where the LCF is overly optimistic. The LCF of the third duplex is quite low. Here the high match count of 22 is better since this duplex is indeed very stable. But the LCF is still preferable because the first case is much more common than the third one.

Definition (LCF profile) Given a master sequence $M = (m, s, l, T)$ with candidate oligonucleotides $o_1, \dots, o_{|s|}$ where the candidate oligonucleotide o_i is equal to $s_i \dots s_{i+l}$ and a set C of N background sequences $\gamma_1, ..., \gamma_N$. The LCF profile $\mathsf{LCF} = (\mathsf{LCF}_{i,j})_{i=1\dots|s|,j=1\dots N}$ is defined by

$$\mathsf{LCF}_{i,j} = \begin{cases} \mathrm{lcf}(o_i, \gamma_j) & \text{if } o_i \text{ is a candidate} \\ 0 & \text{if } o_i \text{ is not a candidate (i.e. if } T_i \text{ is zero)} \end{cases}$$

In other words, $\mathsf{LCF}_{(i,j)}$ is the longest common factor between candidate oligonucleotide o_i and the background sequence j if o_i is a candidate oligonucleotide and zero otherwise.

The LCF profile gives us information about the similarity between the master sequence and each background sequences. But in principle, it does not really interest us to *which* background sequences the master sequence is similar but just to *how many.* This is why we transform the LCF profile to the LCF statistics.

Definition (LCF statistics) Given a master sequence $M = (m, s, l, T)$ with candidate oligonucleotides $o_1, \dots, o_{|s|}$ and a set C of N sequences. Fix an integer $\Delta > 0$ for the *width* of the LCF statistics. $\mathsf{LCFS} = (\mathsf{LCFS}_{i,j})_{i=1\dots|s|,j=1\dots\Delta}$ is then defined by

$$\mathsf{LCFS}_{i,j} = \begin{cases} \#(k : k \neq m \text{ and } \mathsf{LCF}_{i,k} = l - j + 1) & \text{if } o_i \text{ is a candidate} \\ 0 & \text{if } o_i \text{ is not a candidate} \end{cases}$$

In other words, $\mathsf{LCFS}_{i,j}$ is the number of incidents where the longest common factor between candidate oligonucleotide o_i and any background sequence equals to $l - j + 1$ where l is the desired oligonucleotide length as defined before. In particular, the first column $\mathsf{LCFS}_{i=1\dots|s|,j=1}$ contains the number of perfect matches between candidate oligonucleotide o_i and any background sequence. $\#(\cdot)$ denotes the counting function, e.g. $\#(42, 54, 0, 3) = 4$. Note that we exclude the master sequence from the considered background sequences (recall that all master sequences are also background sequences) since obviously there would be a perfect match since this is just exactly the sequence where the candidate oligonucleotide comes from.

The parameter Δ is used to bound the size (or more precisely the width) of the LCFS array. For example, if we want to find oligonucleotides of length 30, we may find it irrelevant how many background sequences have a string of length 9 or less in common with the oligonucleotide. So if we set $\Delta = 21$, we only consider the number of background sequences that have a factor of length 10 to 30 in common with the oligonucleotide.

4.4 Unspecificity Measure

Now we have a set of candidate oligonucleotides for the current master sequence, each with a melting temperature and the LCFS matrix. We combine this information now into a one-dimensional unspecificity measure which we will evaluate for each candidate oligonucleotide.

$$U_i = \begin{cases} \frac{l \cdot \sum_{\delta=1}^{\Delta} \log(\mathsf{LCFS}_{i,\delta}+1) \cdot (\log(N+2))^{-\delta+1}}{T[i]} & \text{if } o_i \text{ is a candidate} \\ +\infty & \text{if } o_i \text{ is not a candidate} \end{cases}$$

This measure is based on a suggestion in [Rah02] which we extended to account for the melting temperature.

4.5 Avoiding Overlaps

We can now sort the oligonucleotides by ascending unspecificity. There is one issue that needs to be taken care of, which is the overlapping of the oligonucleotides. Obviously it is not desirable to have oligonucleotides that overlap, so we pick the oligonucleotide with the lowest unspecificity and then remove all oligonucleotides from the rest of the candidates that overlap with the picked oligonucleotide. Then we pick the second best oligonucleotide and again remove all overlaps. We repeat this procedure until we have the desired number of oligonucleotides for the current master sequence.

Chapter 5

Oligonucleotide Fingerprinting

5.1 Introduction

The method we will present in this chapter uses a quite different foundation than the algorithm discussed in the preceding chapters. Let us first introduce the notion of an *oligonucleotide fingerprint* of the target DNA.

Definition (Oligonucleotide Fingerprint) An oligonucleotide fingerprint represents the interaction between the target DNA and the oligonucleotides in a microarray experiment. The fingerprint is a vector of numerical values where each element describes the hybridization signal between the target DNA and an oligonucleotide, therefore the number of elements in the fingerprint vector corresponds to the number of oligonucleotides we have chosen to put on the microarray. We will only consider vectors with binary values where a zero simply means that the target DNA did not hybridize with the corresponding oligonucleotide and a one indicates hybridization.

Consider the following example. Say we have 10 oligonucleotides which are numbered from 1 to 10. We perform a microarray experiment and the target DNA hybridizes with oligonucleotide 3, 4 and 7. So in this case our fingerprint would be [0011001000]. Now the goal of the algorithm is to choose the oligonucleotides in such a way that the information gained from the resulting fingerprints is maximized. If, for example, all the oligonucleotides would almost always hybridize, the resulting fingerprints would not be very useful since they would provide only little information about the target DNA. Of course, it is true for all oligonucleotide selection algorithms that the resulting oligonucleotides should maximize the information gained from microarray experiments with those oligonucleotides, but the big difference is that we introduce a measure to express the information content as a numeric value.

The resulting fingerprints can be used for example for classification. Ideally, the fingerprint for a particular target DNA is unique. If we do not have any information about the target DNA, then the only way to obtain a unique fingerprint would be to sequence the whole target DNA. This is why we have to restrict ourselves to a class or family of DNA. To choose suitable oligonucleotides we use a number of so-called training sequences which should be similar to the target DNA that will later be examined. The aforementioned information measure uses a concept from information theory which will be introduced in the next section.

5.2 The Concept of Entropy

Entropy is one of the fundamental concepts introduced by Shannon in his revolutionary paper [Sha48] in 1948 which transformed information theory to what it is today. The concept of information is too broad to be captured by a single definition. However, for any probability distribution we define a quantity called *entropy* which has many properties that agree with the intuitive notion of what a measure of information should be. The entropy can also be seen as the amount of uncertainty or disorder. It shares many properties with the concept of entropy in thermodynamics.

Let X be a discrete random variable with alphabet $\mathcal{X}$ and probability mass function $p_X(x) = \Pr\{X = x\}, x \in \mathcal{X}$.

Definition (Entropy) The entropy $H(X)$ of a discrete random variable X is defined by

$$H(X) = -\sum_{x\in\mathcal{X}} p_X(x)\,\log_2 p_X(x)$$

We also write $H(p_X)$ or simply $H(p)$ for the above quantity. Since we take log to the base 2 the entropy is expressed in bits. Note that $H(X) \geq 0$. We use the convention that $0 \log 0 = 0$ which is justified by continuity since $x \log x \to 0$ as $x \to 0$. Thus adding terms of zero probability does not change the entropy.

Let

$$X = \begin{cases} 0 & \text{with probability } \frac{1}{2} \\ 1 & \text{with probability } \frac{1}{2} \end{cases}$$

The entropy $H(X)$ is 1. Now compare this to the following random variable.

$$Y = \begin{cases} 0 & \text{with probability } 1 \\ 1 & \text{with probability } 0 \end{cases}$$

So Y is just always zero and $H(Y) = 0$. It is intuitive that a sequence where the ones and zeros appear with equal probability has more information content than just a sequence of zeros. In fact, with an alphabet of size two, the case where both symbols appear with probability $\frac{1}{2}$ maximizes the entropy. This can be further generalized to alphabets with n symbols. Also in these cases, the random variable where all the symbols of the alphabet are equiprobable maximizes the entropy.

For our purpose, the alphabet of X will consist of all possible fingerprints. Thus if we have k oligonucleotides, the size $|\mathcal{X}|$ of the alphabet is 2^k. For example, say we want to use five oligonucleotides. Hence,

$$\mathcal{X} = \begin{cases} 00000, 00001, 00010, 00011, 00100, 00101, \\ 00110, 00111, 01000, 01001, 01010, 01011, \\ 01100, 01101, 01110, 01111, 10000, 10001, \\ 10010, 10011, 10100, 10101, 10110, 10111, \\ 11000, 11001, 11010, 11011, 11100, 11101, \\ 11110, 11111 \end{cases}$$

which are simply all possible binary sequences of length five. X represents outcomes of microarray experiments. As mentioned before, $H(X)$ is maximized when all elements of the alphabet are equiprobable, hence we have to choose the oligonucleotides in such a way that every fingerprint is equally likely when we perform a microarray experiment.

One could argue that even without the concept of entropy, it would seem sensible to choose the oligonucleotides in that way. This might be true, but it is a convincing mathematical explanation for the choice which otherwise would seem rather unfounded.

5.3 Algorithm

5.3.1 Introduction and Rationale

Our algorithm follows the ideas presented in [HSS+00] where one can also find a more detailed analysis on the quality of the resulting oligonucleotides but it contains only little information on how an implementation could be done.

As before, we interpret the random variable X as outcomes of microarray experiments and the values that X can take on are all possible outcomes, i.e. all possible oligonucleotide fingerprints for the given number of oligonucleotides. Hence the number of oligonucleotides has to be fixed in advance. But how can we know the distribution of the fingerprints obtained from the experiments? Of course, we cannot. What we do is that we restrict ourselves to a class of target DNA with similar characteristics. For example we can restrict ourselves to DNA from a particular organism. Then we have to choose a number of training sequences from that organism. Herwig *et al.* did this in [HSS+00] for both human and rodent sequences. For this purpose, they chose 6000 human and 6000 rodent cDNA sequences at random from the GenBank database and used them as training sequences.

Once appropriate training sequences have been determined, the oligonucleotides are chosen in such a way that the resulting fingerprints of the training sets have maximal information content (i.e. entropy) as defined further above. Now if we would have an infinite amount of time or computational resources we could do the following: As already mentioned, we first have to fix the number of oligonucleotides we want to choose. Suppose we would like to choose 200 oligonucleotides. Furthermore, we suppose that we use a training set of 500 sequences where each sequence has length 1000. All in all this results in

$$500 \cdot 1000 \cdot 2 = 1'000'000 \text{ bases}$$

in the training sequences. We have to multiply by two because we also have to consider the reverse complements. Since we take the candidate oligonucleotides from the training sequences, we would have about 1'000'000 candidate oligonucleotides to choose from. Actually, the real number would be smaller since we cannot take oligonucleotides from the sequence ends and because there are most probably duplicates. For simplicity, let us be conservative and say that we have 200'000 candidate oligonucleotides (that means that we only take 20%!) from which we have to choose the 200 best ones. So this means that we have

$$\frac{200'000!}{(200'000-200)!}$$

possibilities for the set of 200 oligonucleotides. And for each set of oligonucleotides we would have to compute the 500 fingerprints of the training sequences and the corresponding entropy. Obviously, this is not possible within a reasonable amount of time.

In order to reduce the running time we introduce two simplifications:

- We introduce two preselection steps to reduce the number of candidate oligonucleotides. One removes oligonucleotides from the candidate pool based on GC content and complexity. We will define later on how we evaluate the complexity of an oligonucleotide. The second step only retains the most frequent oligonucleotides.
- We only choose one oligonucleotide at a time instead of evaluating all of them in parallel. This means that we first choose one oligonucleotide and then fix it while we look for the second. Once those two are chosen we look for the third oligonucleotide and so forth until we have the desired number of oligonucleotides or if each training sequence has a unique fingerprint. If the latter is the case, we say that these oligonucleotides create a *complete partitioning* of the training set.

In the following subsections we will look at each of these steps in detail.

5.3.2 Preselection I: GC Content and Complexity

We will only consider oligonucleotides which have a minimal number of G or C. The reason is that GC base pairs are more stable than AT base pairs. We could also look at the melting temperature because high GC content results in higher melting temperatures, but since the number of candidate oligonucleotides can be very high, we use the simpler and faster way of just looking at the GC content instead of trying to predict the melting temperature.

Obviously an oligonucleotide which just consists for example of C would be a rather bad choice. In order to remove such oligonucleotides with low complexity, we introduce a complexity measure and then only retain the oligonucleotides which exceed a certain user-supplied complexity threshold.

Definition (Oligonucleotide Complexity) The complexity of an oligonucleotide is computed according to the entropy of the dimer composition of the oligonucleotide. From an oligonucleotide of length l we can extract $l-1$ dimers. The entropy is maximal when each dimer is unique in the oligonucleotide. In that case the entropy is equal to $\log_2(l-1)$. The complexity is normalized so that it is in range between 0 and 1.

Consider the following example: Say we want to compute the complexity of the oligonucleotide `TACGACAC`. This octamer can be split up into the following dimers: TA, AC, CG, GA, AC, CA, AC. Hence the complexity is:

$$\frac{-p_{TA}\log_2 p_{TA} - p_{AC}\log_2 p_{AC} - p_{CG}\log_2 p_{CG} - p_{GA}\log_2 p_{GA} - p_{CA}\log_2 p_{CA}}{\log_2(l-1)}$$

$$= \frac{-\frac{4}{7}\log_2\frac{1}{7} - \frac{3}{7}\log_2\frac{3}{7}}{\log_2 7} = 0.7580$$

Similarly, the complexity for TTGACTAA and TATATATA is 1.0 and 0.3509 respectively. The complexity threshold can be chosen by the user, we use 0.5 as a default value.

5.3.3 Preselection II: Frequency

Since mostly only oligonucleotides which occur frequently in the training sequences result in useful partitions, the second preselection step we introduce only retains the most frequent oligonucleotides. The number of retained oligonucleotides is user-choosable and we use a default value of 2000. We build a table of all the remaining candidate oligonucleotides from the first preselection with their number of occurrences in the training sequences. We then sort the table by the number of occurrences such that the most frequent candidate oligonucleotide is at the top. We then just cut off the list such that we only have the desired number of oligonucleotides left.

5.3.4 Core Selection Algorithm

We first build a suffix array for each training sequence as defined in chapter 3 on page 19 onwards. To find the n^{th} oligonucleotide we perform the following steps:

- For each remaining candidate oligonucleotide (that means all the oligonucleotides from the preselection minus the $n-1$ oligonucleotides that we already chose) we compute the entropy of the fingerprint that would result using this oligonucleotide and the $n-1$ oligonucleotides already chosen. To compute the entropy we first have to find all the training sequences where there is a match with the current candidate oligonucleotide. To do this efficiently, we use the suffix arrays we have built, in particular the bck and lcp data structures.

- We choose the oligonucleotide which resulted in the most partitions. If there are multiple oligonucleotides which create this number of partitions, we choose the one which yields the highest entropy as the n^{th} oligonucleotide and we start over until we have found the desired number of oligonucleotides.

Consider the following example. Say we have the following ten training sequences:

```
1)  CTTTACTCGCGCGTTGGAGA       6) GCGCTCCAACGCATAACTTT
2)  ATACAATAGTGCGGCTCTGT       7) CGCCAGAAGATAGATAGAAT
3)  TCCTTATGAAGTCAACAATT       8) GTGTAAGAAACTGTAATATA
4)  CGCTGGGACTTGCGGCGACT       9) ATAATGAACTTCGGCGAGTC
5)  CATCGTGGTCTCTGTCATTA      10) GTGGAGTTTTTGTTGCATTA
```

Since we also have to consider the reverse complements, we have the following:

```
1)  CTTTACTCGCGCGTTGGAGA TCTCCAACGCGCGAGTAAAG
2)  ATACAATAGTGCGGCTCTGT ACAGAGCCGCACTATTGTAT
3)  TCCTTATGAAGTCAACAATT AATTGTTGACTTCATAAGGA
4)  CGCTGGGACTTGCGGCGACT AGTCGCCGCAAGTCCCAGCG
5)  CATCGTGGTCTCTGTCATTA TAATGACAGAGACCACGATG
6)  GCGCTCCAACGCATAACTTT AAAGTTATGCGTTGGAGCGC
7)  CGCCAGAAGATAGATAGAAT ATTCTATCTATCTTCTGGCG
8)  GTGTAAGAAACTGTAATATA TATATTACAGTTTCTTACAC
9)  ATAATGAACTTCGGCGAGTC GACTCGCCGAAGTTCATTAT
10) GTGGAGTTTTTGTTGCATTA TAATGCAACAAAAACTCCAC
```

Let us suppose we are looking for five oligonucleotides of length four. We will skip the two preselection steps and just suppose that we got the following 40 candidates oligonucleotides out of the preselection:

```
CGCG, GTTG, CAAC, AGTC, GACT, ACTC, TCGC, GCGC,
TGGA, GGAG, CTCC, TCCA, GCGA, GAGT, TGCG, CGGC,
TCTG, CTGT, ACAG, CAGA, GCCG, CGCA, GAAG, CTTC,
GGCG, CGCC, GCGC, CTCG, CGTT, TTGG, GAGA, TCTC,
CCAA, AACG, ACGC, CGAG, GCTC, CTCT, AGAG, GAGC
```

In the first step, we look for the first oligonucleotide which firstly maximizes the number of partitions and secondly maximizes the entropy. Obviously with only one oligonucleotide, the maximum number of partitions is two and the only two cases which would result in a single partition are when a candidate oligonucleotide either occurs in all or none of the training sequences. Also the theoretical maximum entropy is

$$-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}=-\log_2\frac{1}{2}=1$$

which is the case when the candidate oligonucleotide creates two partitions of equal size (in this case of size five since we have a total of ten training sequences). This simply means that an optimal partitioning would be an oligonucleotide that matches with five of the training sequences and does not occur in the other five training sequences.

Going back to our example, we find that all the 40 candidate oligonucleotides create two partitions. So we will just pick the one with the highest entropy. There are two candidate oligonucleotides, `GTTG` and `CAAC` which yield an entropy of 0.970951 which is the maximum. So we can just pick any of those two, we pick `GTTG` because it occurs before the other one. This oligonucleotide occurs in sequence 1, 3, 6 and 10. Therefore we have a probability of $\frac{2}{5}$ that we have a match with this oligonucleotide if we use sequences similar to the training sequences. Or, to put it in a different way, this oligonucleotide creates a partitioning with two partitions, one consists of the sequences 1, 3, 6 and 10 and the other partition consists of the remaining training sequences, which are 2, 4, 5, 7, 8 and 9. The entropy of 0.970951 is thus obtained by computing the following:

$$-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.970951$$

Now that we have fixed the first oligonucleotide, we can start looking for the second one. Again we look for the oligonucleotide which maximizes the number of partitions, but together with the first oligonucleotide we already chose and still use the entropy as a secondary selection criterion. The candidate oligonucleotide set consists of the oligonucleotides we obtained from the preselection process minus the first oligonucleotide we chose, i.e. `GTTG`. Obviously the minimal number of partitions we will get is two, since even if the second oligonucleotide does not introduce any additional partitions, we still have the two partitions introduced by the first oligonucleotide. Examination of the 39 candidate oligonucleotides shows that 16 of them yield four partitions while the other result in three or two partitions. A look at the entropies shows that the maximum is 1.846439, 8 of the 16 oligonucleotides attain this value: `AGTC, GACT, TCGC, GCGA, TGCG, CGCA, GAAG` and `CTTC`. Again we just choose the first one: `AGTC`. The theoretical maximum entropy for two oligonucleotides is:

$$-\log_2\frac{1}{4} = 2$$

As before, the actual maximum entropy attained is below the theoretical maximum. With the two oligonucleotides chosen so far, we can have four possible fingerprints (resulting in the four partitions): `00 01 10` and `11`. With the training sequences, these fingerprints occur 4, 2, 3 and 1 times respectively. Hence the entropy can be computed like this:

$$-\frac{2}{5}\log_2\frac{2}{5} - \frac{1}{5}\log_2\frac{1}{5} - \frac{3}{10}\log_2\frac{3}{10} - \frac{1}{10}\log_2\frac{1}{10} = 1.846439$$

It should be noted that in this particular case, it would have been impossible to attain the maximal theoretical entropy. This is because the maximal entropy can only be attained if the partitions are of equal size. Here we have four partitions and ten training sequences, hence the partition size would have to be 2.5 which is of course not possible. Even though normally a much larger number of training sequences is used, it is the case that the theoretical maximum is only attained very seldomly and with increasing number of oligonucleotides, the gap between theoretical and actual entropy is widening quickly.

We repeat the same steps we have done for the first two oligonucleotides until we have the desired number of oligonucleotides. In our example, in the third step there is a notable difference to the first two steps: The number of partitions or distinctive fingerprints is seven, therefore not only the entropy is below the theoretical maximum but also the number of partitions. The theoretical maximum number of partitions is simply 2^k where k is the number of oligonucleotides. The number of partitions is equal to the number of fingerprints which is a binary sequence of length k.

To finish our example, the remaining three oligonucleotides would be `TGCG, GAGA` and `TCTG`. At the end, the five oligonucleotides we have found result in 10 partitions (theoretical maximum 32) and the entropy is 3.321928 (theoretical maximum 5). Observe that the theoretical maximal entropy is simply the number of oligonucleotides.

Chapter 6

Running Time Considerations

6.1 Introduction

Because we use a number of different algorithms and because of the numerous parameters, it would be relatively difficult to perform a detailed complexity analysis. Since some parts of the implementation are written in interpreted DARWIN code and other parts in much faster precompiled C code, the results of a complexity analysis would give relatively little information about the running time of the oligonucleotide selection process in reality. We will therefore restrict ourselves to a number of time measurements to give a rough idea how long it actually takes in reality to determine the oligonucleotides.

All the measurements in the following sections have been performed on a Linux machine with two Pentium III 800 MHz CPUs, 512 MB of RAM and Linux Kernel 2.4.20. Since DARWIN does not make much use of the second CPU and there were other processes running in the background, all the results cannot be more than a approximate guideline.

6.2 Oligonucleotide Selection with Suffix Arrays

For the oligonucleotide selection with suffix arrays we performed two test series. For both series we used the default values, i.e. $q = 8$, probe length=20, LCFS width=10, self complementarity threshold=0.4 and 5 oligonucleotides per master sequence. Furthermore, we used all the input sequences as master sequences. In the first test series we used 50 randomly generated input sequences with the same length each. Figure 6.1 shows the measured running times. The second series used 100 input sequences, again all of them master sequences. It can be easily seen that the running times increase quickly with growing input data, in the case of 50 sequences of 10'000 base pairs (which results in a total input size of 500'000 base pairs) it already takes almost 10 hours. In the second series with 100 input sequences of length 5000 (total length also 500'000 base pairs) the running time is more than 14 hours.

Unfortunately we were not able to get to run other publicly available implementations because of software incompatibilities. For yeast, Li and Stormo mention a running time of 92 hours to select five oligonucleotides of length 50 for each gene or ORF in [LS01]. They do not mention what kind of machine they used. In [KS02], Kaderali and Schliep report a running time of just about one hour to select oligonucleotides with lengths between 19 and 21 for 58 sequences of average length of 9300 nucleotides. They used a Compaq Tru64 machine with four DEC Alpha EV6.7 CPUs each operating at 667 MHz. This would be considerably faster than our implementation, one reason is certainly that their implementation is completely written in C/C++. In our implementation, very little time is needed to construct the suffix arrays but most of the time (more than 80%) is used to perform the searches afterwards, which is implemented in DARWIN code. A reimplementation of those parts in C would result in a major speed gain.

6.3 Oligonucleotide Fingerprinting

For the fingerprinting algorithm we also ran two test series with default values, one with 500 training sequences of length 50, the other one with the same number of training sequences of length 500. Figure 6.2 shows the measured running times. For the test run with sequences of length 50, the time needed for the preselection was 2005s and the total time to find the 20 oligonucleotides was 3969s. For the training sequences of length 500 the preselection took 7451s and the total time was 11626s. It is quite surprising that the computation time for the oligonucleotides does increase only so slowly because with every additional oligonucleotide the number of fingerprints that need to be considered doubles. It is maybe a sign that there is still too much overhead and only a small fraction of the time is used to do the actual computations.

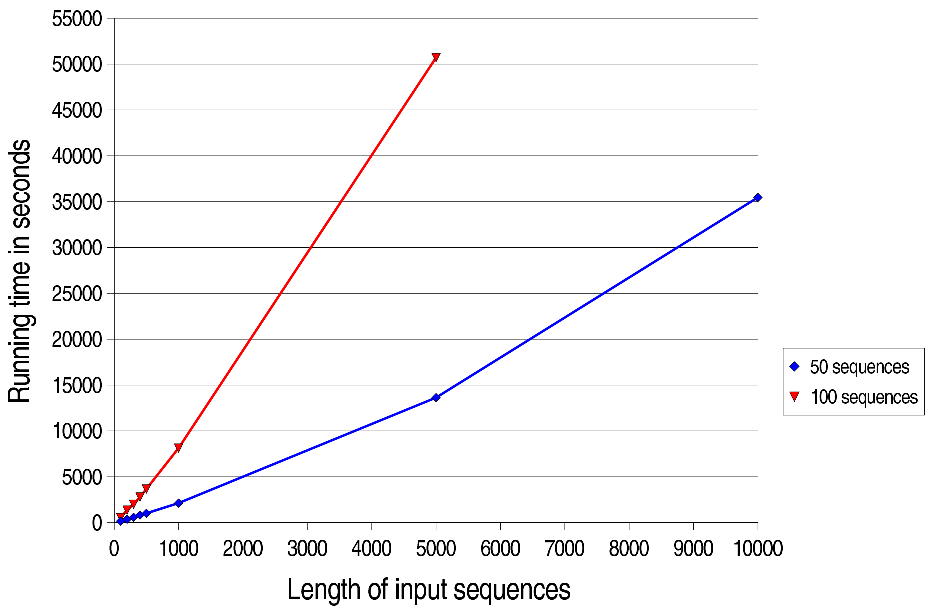

Figure 6.1: Time measurements for oligonucleotide selection with suffix arrays

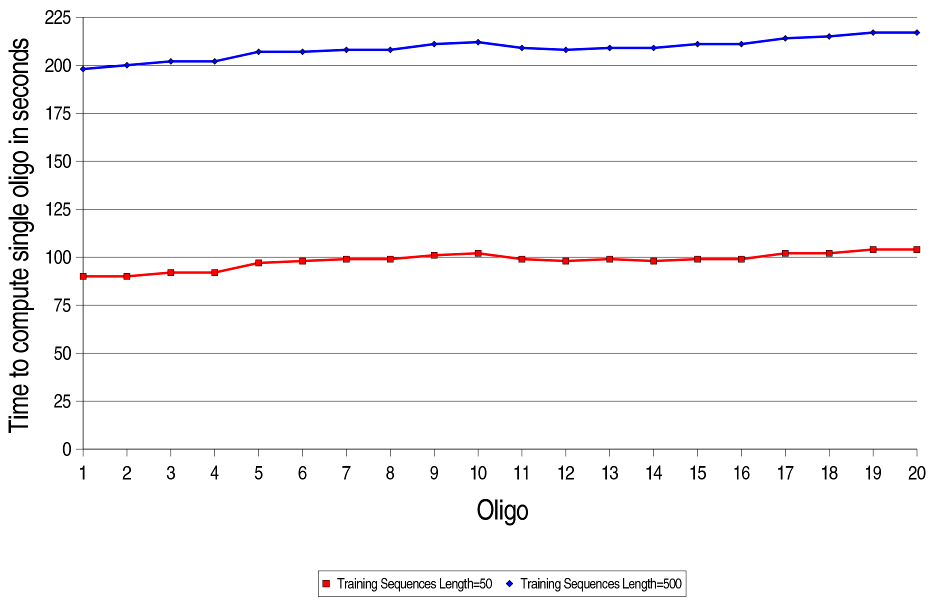

Figure 6.2: Time measurements for oligonucleotide selection for fingerprinting

Chapter 7

Discussion and Conclusion

7.1 Points for Improvements

- With the advent of widespread availability of technologies that make microarray experiments very cheap and easy such as the fabrication of microarrays using ink-jet printing (see [Hug01]), it is not acceptable to wait for several hours or even days to compute the oligonucleotides. Hence the performance still needs to be improved. Firstly this can be done by identifying the performance bottlenecks in the DARWIN code and then by reimplementing those code parts in C. Secondly also the algorithms itself need to become more efficient.

- The oligonucleotide fingerprinting algorithm does not use melting temperature prediction. Instead a simple preselection based on GC content is used. It might be worthwhile to determine the performance penalty of using melting temperature prediction instead.

- Especially in the fingerprinting algorithm it would be useful to have the option to exclude certain hyper-abundant sequences from consideration. Pesole *et al.* have showed the advantages of this in the context of designing PCR primer pairs in [PLG^{+}98].

7.2 Closing Remarks

We have reached our goal of implementing an algorithm in Darwin that selects oligonucleotides which can be directly used in microarray experiments. We chose two different approaches which apply to different kinds of microarray experiments. Unfortunately, it was not possible to test the resulting oligonucleotides in real-world microarray experiments. It is very difficult to assess the quality of the selected oligonucleotides without actually using them in the laboratory because the selection algorithms are based on certain models and assumptions. If we would try to simulate a microarray experiment using a computer, then again it would be necessary to model the hybridization process. Of course, if we would base this model on the same assumptions that we made for the oligonucleotide selection algorithms then obviously the oligonucleotides selected by the algorithm would be "perfect" since they are both based on the same model. So the only reliable way to assess the quality of the oligonucleotides really seems to perform the experiments and then use the results to improve the algorithm and repeat the procedure until the actual and expected results only differ marginally.

Appendix A

Source Code Organization

A.1 Overview

The source code is organized in the following subdirectories:

```
ExtCallsSupport
FindPartition
LCFStatistics
MeltingTemperature
Preselection
PreselectionFrequency
SelfComplementarity
SuffixArray
Toolbox
UnspecificityArray
```

We will discuss the contents of the root directory and each subdirectory file by file in the following sections.

A.2 Root Directory

A.2.1 `500x100.fasta`

Randomly generated FASTA file with 500 sequences, each of length 100 nucleotides.

A.2.2 `50x500.fasta`

Randomly generated FASTA file with 50 sequences, each of length 500 nucleotides.

A.2.3 `fasta-gen.c`

This small command-line utility can be used to generate FASTA files of any size. Invoke without parameters to get information about all available options.

A.2.4 `main.drw`

This is the top-level DARWIN source code file. It contains code to load all other necessary files.

A.2.5 `Makefile`

This is the top-level makefile. To compile all C source code files in the root directory and in the subdirectories, invoke `make all`. To remove all binaries and intermediate files, invoke `make clean`.

A.2.6 `OligoSelectFP.drw`

Contains the main oligonucleotide selection function for fingerprinting.

A.2.7 `OligoSelectSA.drw`

Contains the main oligonucleotide selection function for the selection based on suffix arrays.

A.2.8 `test.drw`

Contains the `test` function which runs all available test routines.

A.3 `ExtCallsSupport` Directory

A.3.1 `extcalls.c` and `extcalls.h`

Those two files have been taken from the DARWIN source code tree. They contain support code to use external functions in DARWIN.

A.3.2 `Makefile`

This makefile compiles all C source code files in this directory.

A.3.3 `WordSize.c`

This file has been taken from the DARWIN source code tree. It is a small command-line utility which generates the `WordSize.h` file which is needed to use external functions in DARWIN.

A.4 `FindPartition` Directory

A.4.1 `FindPartition.drw`

Given a number of training sequences, the `FindPartition` function in this file chooses oligonucleotides such that they create an optimal partitioning of the training sequences.

A.5 `LCFStatistics` Directory

A.5.1 `LCFStatistics.drw`

The `LCFStatistics` function in this file computes the LCF statistics matrix.

A.5.2 `LCFStatisticsTest.drw`

This file contains test routines to verify the correctness of the `LCFStatistics` function.

A.6 `MeltingTemperature` Directory

A.6.1 `PredictTm.drw`

Contains the melting temperature prediction function.

A.6.2 `PredictTmArray.drw`

Computes the melting temperatures of an array of sequences. Uses the `PredictTm` function to perform the actual prediction.

A.6.3 `TmNNParameters.drw`

Contains the parameters for the nearest-neighbor melting temperature prediction model.

A.7 `Preselection` Directory

A.7.1 `Makefile`

This makefile compiles all C source code files in this directory.

A.7.2 `Preselection.c`

Performs a preselection of candidate oligonucleotides based on GC content and complexity. Used by the oligonucleotide fingerprinting algorithm.

A.7.3 `Preselection.drw`

DARWIN placeholder function which calls the C version of the function.

A.8 `PreselectionFrequency` Directory

A.8.1 `PreselectionFrequency.drw`

Performs a preselection of candidate oligonucleotides based on frequency. Used by the oligonucleotide fingerprinting algorithm.

A.8.2 `PreselectionFrequencyTest.drw`

This file contains test routines to verify the correctness of the `PreselectionFrequency` function.

A.9 `SelfComplementarity` Directory

A.9.1 `DropSelfComplementary.drw`

The `DropSelfComplementary` function takes a set of candidate oligonucleotides and uses the `SelfComplementarity` function to remove candidates which are self-complementary.

A.9.2 `Makefile`

This makefile compiles all C source code files in this directory.

A.9.3 `SelfComplementarity.c`

Takes a single candidate oligonucleotide and determines if it is self-complementary or not.

A.9.4 `SelfComplementarity.drw`

DARWIN placeholder function which calls the C version of the function.

A.9.5 `SelfComplementarityTest.drw`

This file contains test routines to verify the correctness of the `SelfComplementarity` function.

A.10 `SuffixArray` Directory

A.10.1 `BucketArray.c` and `BucketArray.h`

Contains the C function to construct bucket arrays.

A.10.2 `EnhancedSuffixArray.c`

Returns an enhanced suffix array of the input sequence.

A.10.3 `EnhancedSuffixArray.drw`

DARWIN placeholder function which calls the C version of the function.

A.10.4 `LCPArray.c` and `LCPArray.h`

Computes the longest common prefix array given an input sequence and a suffix array.

A.10.5 `Makefile`

This makefile compiles all C source code files in this directory.

A.10.6 `SuffixArrayTest.drw`

This file contains test routines to verify the correctness of the `EnhancedSuffixArray` function.

A.10.7 `SuffixSort.c` and `SuffixSort.h`

Constructs the suffix array from a sequence.

A.11 `Toolbox` Directory

A.11.1 `Makefile`

This makefile compiles all C source code files in this directory.

A.11.2 `ReadFasta.c`

Reads a FASTA file.

A.11.3 `ReadFasta.drw`

DARWIN placeholder function which calls the C version of the function.

A.11.4 `Toolbox.drw`

Contains various small helper functions.

A.12 `UnspecificityArray` Directory

A.12.1 `UnspecificityArray.drw`

Computes the unspecificity array for a set of candidate oligonucleotides. Used for the selection algorithm based of suffix arrays.

A.13 Implementation Remarks

Most of the functions that are implemented in C have first been written in DARWIN code. The test routines proved to be a useful tool to ensure that the behavior of the C and the DARWIN version is equivalent. All development work has been done under Linux 2.4.20. We used DARWIN version 3.0 from July 2003. The C source code has been successfully tested with the GNU C Compiler version 2.95.2 and 3.3.

Appendix B

Usage Guide

B.1 Compilation

Type the shell command

```
make all
```

in the top-level source code directory to compile all C source code. To remove all intermediate and binary files type

```
make clean
```

In order to only compile or clean the files in a single subdirectory, use the same commands in that subdirectory. Note that you need to compile the C source code before you can use the functions in DARWIN.

B.2 Execution in DARWIN

You have to start DARWIN from the top-level source code directory, otherwise DARWIN will not find the source code files. To load all necessary files, type the following two DARWIN commands:

```
ReadProgram ('main.drw');
load();
```

This should result in the following output:

```
Darwin: Sequence Searching Facility
Version 3.0, July 2003
  (c) E.T.H. Zurich
> ReadProgram ('main.drw');
> load();
bytes alloc=131048, time=0.020
--- Loading FindPartition/FindPartition.drw
--- FindPartition/FindPartition.drw done
--- Loading LCFStatistics/LCFStatistics.drw
--- LCFStatistics/LCFStatistics.drw done
--- Loading LCFStatistics/LCFStatisticsTest.drw
--- LCFStatistics/LCFStatisticsTest.drw done
--- Loading MeltingTemperature/TmNNParameters.drw
--- MeltingTemperature/TmNNParameters.drw done
--- Loading MeltingTemperature/PredictTm.drw
--- MeltingTemperature/PredictTm.drw done
--- Loading MeltingTemperature/PredictTmArray.drw
--- MeltingTemperature/PredictTmArray.drw done
--- Loading Preselection/Preselection.drw
--- Preselection/Preselection.drw done
--- Loading PreselectionFrequency/PreselectionFrequency.drw
--- PreselectionFrequency/PreselectionFrequency.drw done
--- Loading PreselectionFrequency/PreselectionFrequencyTest.drw
--- PreselectionFrequency/PreselectionFrequencyTest.drw done
--- Loading SelfComplementarity/SelfComplementarity.drw
--- SelfComplementarity/SelfComplementarity.drw done
--- Loading SelfComplementarity/SelfComplementarityTest.drw
--- SelfComplementarity/SelfComplementarityTest.drw done
--- Loading SelfComplementarity/DropSelfComplementary.drw
--- SelfComplementarity/DropSelfComplementary.drw done
--- Loading SuffixArray/EnhancedSuffixArray.drw
--- SuffixArray/EnhancedSuffixArray.drw done
--- Loading SuffixArray/SuffixArrayTest.drw
--- SuffixArray/SuffixArrayTest.drw done
--- Loading Toolbox/ReadFasta.drw
--- Toolbox/ReadFasta.drw done
--- Loading Toolbox/Toolbox.drw
--- Toolbox/Toolbox.drw done
--- Loading UnspecificityArray/UnspecificityArray.drw
--- UnspecificityArray/UnspecificityArray.drw done
--- Loading OligoSelectSA.drw
--- OligoSelectSA.drw done
--- Loading OligoSelectFP.drw
--- OligoSelectFP.drw done
--- Loading test.drw
--- test.drw done
```

To make sure everything works correctly, run the test routines by typing

```
test();
```

You should get the following output:

```
> test();
Performing tests...
--- start unit test: SuffixArray
Building enhanced suffix array (n=15)
1 passed
2 passed
3 passed
4 passed
5 passed
```

```
6 passed
--- end unit test: SuffixArray
--- start unit test: LCFStatistics
Building enhanced suffix array (n=15)
Building enhanced suffix array (n=27)
Building enhanced suffix array (n=32)
Building LCF profile of master sequence (sequence m=1, length=15)
LCF profile done
Computing LCF statistics from LCF profile
LCF statistics done
1 passed
2 passed
3 passed
--- end unit test: LCFStatistics
--- start unit test: SelfComplementary
1 passed
2 passed
3 passed
Checking 30 oligos for self-complementarity
Dropped 8 out of 30 oligos
Checking 30 oligos for self-complementarity
Dropped 8 out of 30 oligos
4 passed
Checking 30 oligos for self-complementarity
Dropped 8 out of 30 oligos
5 passed
--- end unit test: SelfComplementary
--- start unit test: PreselectionFrequency
Building enhanced suffix array (n=20)
Building enhanced suffix array (n=20)
Building enhanced suffix array (n=20)
Beginning preselection of oligos based on frequencies
Processing training sequence 1 out of 3
Processing training sequence 2 out of 3
Processing training sequence 3 out of 3
Retained 8 oligos, max occurences: 3, min: 2
Preselection of oligos based on frequencies done
1 passed
2 passed
--- end unit test: PreselectionFrequency
```

Now that all files are loaded correctly, you can start the oligonucleotide selection process by typing

```
OligoSelectSA();
```

for the suffix array algorithm and

```
OligoSelectFP();
```

for the fingerprinting algorithm. When the selection is done, the results are written to the file `Result-OligoSelectSA.prb` and `Result-OligoSelectFP.prb` respectively. Various status and progress information is displayed during the computations. Note that both of these functions do not take any parameters. If you want to change the parameters, you need to edit the `OligoSelectSA.drw` and `OligoSelectFP.drw` files directly. All the parameters are defined at the beginning of those two files. If you have changed either of those two files, you need to type

```
load();
```

in DARWIN again, otherwise the changes will be ignored.

Appendix C

Formulation of Thesis Objectives

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Scientific Computing
Department of Computer Science

Prof. Dr. Gaston Gonnet

Institute for Scientific Computing
HRS H29
ETH Zurich
CH-8092 Zurich

Tel. direct	+41-1-632 7470
Tel. Secr.	+41-1-632 7471
Tel. Switchboard	+41-1-632 1111
Fax	+41-1-632 1374
E-Mail	gonnet@inf.ethz.ch
Web	http://www.inf.ethz.ch/~gonnet

Zurich, 7th May 2003

Diploma Thesis Project for Thomas Brunner, Designing Oligonucleotides for a Random Chip Used in Taxa Classification

Start: May 12, 2003
End: September 11, 2003

The advent of the oligonucleotidearray technology allows for a massively parallel classification of a wide variety of taxa. This classification of taxa is of great interest in medical diagnosis and phytodiagnostics where the taxa are mainly microorganisms causing diseases. The first step in the procedure of taxa classification via oligonucleotidearrays consists always of a careful selection of the oligonucleotide sequences to be put on the array.

One approach to select oligonucleotides is to start from a given set of taxa with known DNA sequence information. From these DNA sequences oligonucleotides can be selected using multiple sequence alignments. Although this seams to be a very efficient selection procedure, any taxa that was not included in the original given set, cannot be classified.

To circumvent this problem, an alternative approach - the so-called random chip - was developed by one of our collaborators at the Swiss Federal Research Station for Fruit Growing, Viniculture and Horticulture. This approach selects all possible combinations of oligonucleotides of a given length to be put on the array. A tagged single base extension reaction generates absent/present signals in the unknown DNA sample. The pattern of these signals can then be used for a classification of the DNA sample into taxonomic groups.

The main problem with the random chip is that not all oligonucleotides from the set of all possible combinations fulfill the requirements imposed by the hybridization reaction. The two main parameters that have to be adjusted are the melting temperature and the GC-content of the oligonucleotide. So far oligonucleotides for the random chip have been selected by brute force. This means all possible combinations of oligonucleotides have been generated and the ones with unsatisfactory melting temperature or GC-content have subsequently been filtered out. This procedure is only feasible for very short oligonucleotides.

This project contains two tasks. First, a data set on results of melting temperature experiments has to be analyzed. These results are expected to improve melting temperature prediction that have been unsatisfactory with existing methods. Second, an efficient algorithm to generate oligonucleotides of length up to 50 bases that have satisfactory melting temperature and GC-content has to be developed.

The following milestones are planned:

1. Analyze the data from the melting temperature experiment to improve prediction of melting temperatures for oligonucleotides.
2. Develop an algorithm that generates the oligonucleotides to be put on the random chip.
3. Compare sensitivity and robustness of the taxa classification using the random chip approach with other taxa classification methods.

Appendix D

Planning

Designing Oligonucleotides for DNA Microarrays

Planning

Thomas Brunner, 31.8.2003

	12.5-16.5	19.5-23.5	26.5-30.5	2.6-6.6	9.6-13.6	16.6-20.6	23.6-27.6	30.6-4.7	7.7-11.7	14.7-18.7	21.7-25.7	28.7-1.8	4.8-8.8	11.8-15.8	18.8-22.8	25.8-29.8	1.9-5.9	8.9-11.9
Week	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Reading literature	▨																	
Melting temperature	▨	▨																
Updating documentation		▨																
Suffix arrays for gene identification			▨	▨	▨	▨	▨	▨	▨									
Updating documentation						▨	▨	▨	▨	▨								
Sequence Fingerprinting											▨	▨	▨	▨	▨			
Updating documentation														▨	▨	▨	▨	▨
Evaluation, comparisons																▨	▨	▨

Appendix E

Measured Running Times

Time Measurements: Oligo Selection w/Suffix Arrays

Thomas Brunner, 2003-09-10

Sequence Length	Time [s] for SA for 1000 sequences (50 distinctive sequences * 20)
500	52
1000	60
1500	65
2000	70
2500	75
3000	81
3500	87
4000	93
4500	98
5000	103
5500	108
6000	113
6500	119
7000	124
7500	128
8000	139
8500	141
9000	146
9500	151
10000	158
10500	163
11000	168
11500	173
12000	179
12500	186
13000	188
13500	197
14000	203
14500	209
15000	214
15500	223
16000	229
16500	236
17000	242
17500	249
18000	258
18500	262
19000	271
19500	279
20000	286

Time [s]	50 sequences	100 sequences
100	161	543
200	368	1373
300	579	1999
400	822	2796
500	1014	3668
1000	2130	8132
5000	13628	50704
10000	35459	

Time Measurements: Oligo Selection for Fingerprinting

Thomas Brunner, 2003-09-10

Preselection time for 500 training sequences in seconds

Length 50	2005
Length 500	7451

Oligo	Training Sequences Length=50	Training Sequences Length=500
1	90	198
2	90	200
3	92	202
4	92	202
5	97	207
6	98	207
7	99	208
8	99	208
9	101	211
10	102	212
11	99	209
12	98	208
13	99	209
14	98	209
15	99	211
16	99	211
17	102	214
18	102	215
19	104	217
20	104	217
Total time incl. preselection		
	3969	11626

Bibliography

[AS97] H.T. Allawi and J. SantaLucia. Thermodynamics and NMR of internal G-T mismatches in DNA. *Biochemistry*, 36:10581–10594, 1997.

[BD98] R.D. Blake and S.G. Delcourt. Thermal stability of DNA. *Nucleic Acids Res.*, 26, 1998.

[BDTU74] P.N. Borer, B. Dengler, I. Tinoco, and O.C. Uhlenbeck. *J. Mol. Biol.*, 86:843–853, 1974.

[Cup97] J. Cupal. The density states of RNA secondary structures. Master's thesis, University of Vienna, 1997.

[FDP+98] Alexander V. Fotin, Aleksei L. Drobyshev, Dmitri Y. Proudnikov, Alexander N. Perov, and Andrei D. Mirzabekov. Parallel thermodynamic analysis of duplexes on oligodeoxyribonucleotide microchips. *Nucleic Acids Research*, 26(6):1515–1521, 1998.

[HSS+00] Ralf Herwig, Armin O. Schmitt, Matthias Steinfath, John O'Brien, Henrik Seidel, Sebastian Meier-Ewert, Hans Lehrach, and Uwe Radelof. Information theoretical probe selection for hybridisation experiments. *Bioinformatics*, 16(10):890–898, 2000.

[Hug01] Timothy R. Hughes. Expression profiling using microarrays fabricated by an ink-jet oligonucleotide synthesizer. *Nature Biotechnology*, 19:342–347, April 2001.

[KS02] Lars Kaderali and Alexander Schliep. Selecting signature oligonucleotides to identify organisms using DNA arrays. *Bioinformatics*, 18(10):1340–1349, 2002.

[LS00] Fugen Li and Gary D. Stormo. Selecting optimum DNA oligos for microarrays. *IEEE International Symposium on Bio-Informatic & Biomedical Engineering (BIBE 2000)*, 2000.

[LS01] Fugen Li and Gary D. Stormo. Selection of optimal DNA oligos for gene expression arrays. *Bioinformatics*, 17(11):1067–1076, 2001.

[MM93] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, October 1993.

[MM97] Peter M. McIlroy and M. Douglas McIlroy. Suffix sort source code in C. `http://cm.bell-labs.com/cm/cs/who/doug/ssort.c`, 1997.

[OVG$^+$98] Richard Owczarzy, Peter M. Vallone, Frank J. Gallo, Teodoro M. Paner, Michael J. Lane, and Albert S. Benight. Predicting sequence-dependent melting stability of short duplex DNA oligomers. *Biopolymers*, 44:217–239, 1998.

[PLG$^+$98] G. Pesole, S. Liuni, G. Grillo, P. Belichard, T. Trenkle, J. Welsh, and M. McClelland. Geneup: A program to select short PCR primer pairs that occur in multiple members of sequence lists. *BioTechniques*, 25:112–123, 1998.

[Rah02] Sven Rahmann. Rapid large-scale selection of oligonucleotides for microarrays. In *Proceedings of the First IEEE Computer Society Bioinformatics Conference (CSB 2002)*, pages 54–63. IEEE Press, 2002.

[San98] John SantaLucia. A unified view of polymer, dumbbell, and oligonucleotide RNA nearest-neighbour thermodynamics. *Proc. Natl. Acad. Sci. USA*, 95:1460–1465, February 1998.

[Sch03] Mark Schena. *Microarray analysis*. Wiley-Liss, 2003.

[Sha48] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Journal*, 27:379–423, 623–656, 1948.

[VP02] Arnold Vainrub and B. Montgomery Pettitt. Coulomb blockage of hybridization in two-dimensional DNA arrays. *Physical Review*, 66, 2002.