Algorithms for Improving the Design and Production of Oligonucleotide Microarrays

Sérgio Anibal de Carvalho Junior

Februar 2007

Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
(Doctor rerum naturalium)

an der Technischen Fakultät
der Universität Bielefeld

Betreuer:
Dr. rer. nat. Sven Rahmann
Prof. Dr. rer. nat. Jens Stoye

Foreword

Microarrays are a ubiquitous tool in molecular biology with a wide range of applications on a whole-genome scale including high-throughput gene expression analysis, genotyping, and resequencing. Although several different microarray platforms exist, we focus on high-density oligonucleotide arrays, sometimes called DNA chips. One of the advantages of higher density arrays is that they allow the simultaneous measurement of the expression of several thousand genes at once, possibly covering all genes of a species in a single experiment.

Oligonucleotide microarrays consist of short DNA molecules, called *probes*, affixed or synthesized at specific locations of a solid support. Probes are built, nucleotide-by-nucleotide, by a light-directed combinatorial chemistry. Because of the natural properties of light, the quality of a microarray can be compromised if the physical arrangement of the probes on the array and their synthesis schedule are not carefully designed. This thesis is mainly concerned with the problem of designing the layout of a microarray in such a way that the incidence of the *unintended illumination problem* is reduced. We call it the *microarray layout problem* (MLP), using the term *layout* to refer to where and how the probes are synthesized on the array, i.e., their arrangement and their *embeddings*.

In the first chapter of this thesis, we briefly review the role of microarrays in analyzing complex genetic information. We then describe the technology currently employed in the production of high-density microarrays as well as the problems that arise during manufacturing. In Chapter 2, we give a formal definition to the microarray layout problem and describe in detail two quality measures that are used to evaluate a given layout. Finding an optimal layout with respect to any of these two measures seems unlikely, even for very small arrays. As we shall see in Chapter 4, the MLP can be modeled as a quadratic assignment problem (QAP), a classical combinatorial optimization problem that is notoriously hard to solve in practice, giveing further indication that the MLP is, in fact, a hard problem. In practice, the layout problem is usually approached in several "phases" with a range of heuristic algorithms.

The *placement* phase is the subject of Chapter 3. Traditionally, this phase consists of fixing an embedding for all probes and finding an arrangement minimizing a given cost function. We describe several known placement algorithms with an emphasis on methods that can be used to design large arrays. A new algorithm, called Greedy, is also presented. One of the reason why we show the relation between the MLP

and the QAP is that we can now use QAP techniques as placement algorithms. This is interesting because there is a rich literature on methods for solving the QAP. In Chapter 4, we also show the results of using one QAP heuristic to design small artificial chips, and discuss how this approach can be applied to larger microarrays.

Chapter 5 focuses on the *re-embedding* phase that usually follows the placement. In this phase, one attempts to further improve the layout by finding a different embedding of the probes without changing their location on the chip. Again, we review all known re-embedding algorithms, describring the most successful ones in detail. We also introduce a new algorithm, called Priority re-embedding.

In the last decade, commercial microarrays have grown from a few thousands to more than a million probe sequences on a single chip. Many placement algorithms are unable to deal with such large arrays because of their non-linear time and space complexities. For this reason, the layout problem is sometimes broken into smaller sub-problem by a *partitioning* algorithm. This is the focus of Chapter 6, where we present an extensive evaluation of existing algorithms and show how the partitioning phase can improve solution quality and reduce running time.

In Chapter 7, we discuss the disadvantages of the tradional "place and re-embed" approach to the layout problem. We then propose a new algorithm, called Greedy+, that for the first merges the placement and re-embedding phases into a single one. Our results show that Greedy+ indeed outperforms all known placement algorithms.

In Chapter 8, we present a pioneering analysis and evaluation of the layout of several GeneChip® arrays, considered the industry standard in terms of high-density oligonucleotide microarrays. Some design decisions that might affect the quality of these arrays are described in detail. We then use some of the algorithms presented in earlier chapters to propose alternative layouts for two of the latest generation of GeneChip arrays, showing how the risk of the unintended illumination problem can be reduced.

Another problem related to the production of microarrays is to find the shortest synthesis schedule for a given set of probes, which we refer to as the *shortest deposition sequence problem* (SDSP). The SDSP is an instance of the shortest common supersequence problem (SCSP), a classical problem in computer science that is known to be NP-complete even under various restrictions. Several existing heuristics are able to find good approximate solutions for the SCSP, but, in Chapter 9, we investigate the feasibility of finding *the shortest* deposition sequence for currently available oligonucleotide microarrays. Chapter 10 concludes this thesis with a short discussion about the presented results.

Publications. Parts of this thesis have been published in advance. The conflict index model for evaluating a microarray layout (Chapter 2) and the Pivot Partitioning

algorithm (Section 6.4) were first presented at the Workshop on Algorithms in Bioinformatics (WABI), in Zürich (de Carvalho Jr. and Rahmann, 2006a). The conflict index model was also presented, together with the QAP formulation of the microarray layout problem (Chapter 4), at the German Conference on Bioinformatics (GCB) in Tübingen (de Carvalho Jr. and Rahmann, 2006b). The work on the shortest common supersequence (Chapter 9) was first published as a technical report at the Faculty of Technology of Bielefeld University (de Carvalho Jr. and Rahmann, 2005). Finally, a book chapter containing a more accessible description of the microarray layout problem and of several algorithms presented here, including the previously unpublished Greedy+, 1-Dimensional and 2-Dimensional Partitioning, is expected to appear in late 2007 (de Carvalho Jr. and Rahmann, 2007).

This thesis also contains previously unpublished material, namely:

- the Greedy placement algorithm (Section 3.6) that outperforms previous algorithms in terms of conflict index minimization;
- the Priority re-embedding algorithm (Section 5.5) that achieves marginal improvements compared to the best known algorithms;
- an analysis of the layout of several commercially available GeneChip arrays with respect to the defined evaluation criteria (Chapter 8).

Acknowledgments. This work was carried out while I was a member of the Junior research group (recently-renamed) Computational Methods for Emerging Technologies (COMET), which is part of the AG Genominformatik led by Prof. Jens Stoye. I thank all present and former colleagues as well as students of the International NRW Graduate School in Bioinformatics and Genome Research and the Graduiertenkolleg Bioinformatik, of which I am also a member, for the nice research atmosphere I found in Bielefeld, and for an enjoyable time I had in the last three years.

Special thanks go to Dr. Sven Rahmann for suggesting the topic and for the opportunity to work under his supervision. This work owes much to his expertise. Whenever I write "we" in this thesis, I mean "Sven and I". On several occasions, the support of the Bioinformatics Resource facility (BRF) at the CeBiTec (Center for BioTechnology) was crucial to the success of this work, and I cannot thank them enough for their help. Epameinondas Fritzilas, Ferdinando Cicalese, José Augusto Amgarten Quitzau, and Klaus-Bernd Schürmann read early drafts of several chapters of this thesis and helped to improve them in many ways. I would also like to thank Dr. Peter Hahn and Chris MacPhee for working on several QAP instances of Chapter 4 and for helpful discussions.

Sérgio A. de Carvalho Jr. Bielefeld, Februar 2007

Contents

Foreword i

1 Introduction 1
- 1.1 High-density oligonucleotide microarrays . . . 2
- 1.2 Manufacturing and design problems . . . 5

2 The Microarray Layout Problem 9
- 2.1 Problem statement . . . 11
- 2.2 Border length . . . 11
- 2.3 Conflict index . . . 13
- 2.4 Chip quality measures . . . 16
- 2.5 How hard is the microarray layout problem? . . . 16

3 Placement Algorithms 19
- 3.1 Optimal masks for uniform arrays . . . 19
- 3.2 TSP and threading algorithms . . . 20
- 3.3 Epitaxial placement . . . 22
- 3.4 Sliding-Window Matching . . . 23
- 3.5 Row-Epitaxial . . . 24
- 3.6 Greedy . . . 25
- 3.7 Summary . . . 33

4 MLP and the Quadratic Assignment Problem 35
- 4.1 Quadratic assignment problem . . . 35
- 4.2 QAP formulation of the MLP . . . 36
- 4.3 QAP heuristics . . . 38
- 4.4 Results . . . 40
- 4.5 Discussion . . . 41

5 Re-embedding Algorithms 45
- 5.1 Optimum Single Probe Embedding . . . 45
- 5.2 Chessboard . . . 47
- 5.3 Greedy and Batched Greedy . . . 48
- 5.4 Sequential re-embedding . . . 49
- 5.5 Priority re-embedding . . . 51

5.6 Summary 54

6 Partitioning Algorithms **57**
6.1 1-Dimensional Partitioning 58
6.2 2-Dimensional Partitioning 59
6.3 Centroid-based Quadrisection 66
6.4 Pivot Partitioning 67
6.5 Summary 74

7 Merging Placement and Re-embedding **77**
7.1 Greedy+ 77
7.2 Results 78
7.3 Summary 83

8 Analysis of Affymetrix Microarrays **87**
8.1 Introduction 87
8.2 Layout Analysis 89
8.3 Alternative Layouts 92
8.4 Summary 93

9 The Shortest Deposition Sequence Problem **95**
9.1 Introduction 95
9.2 Alternatives 97
9.3 Upper bound algorithms 100
9.4 Lower Bound Algorithms 101
9.5 Implementation 104
9.6 Results 110
9.7 Disscussion 111

10 Discussion **115**
10.1 Outlook 118

Chapter 1

Introduction

In the last few years, the genomes of an increasing number of organisms have been sequenced, generating a vast amount of information. Sequencing the genomes, however, is just the first step in understanding these organisms at the molecular level, and the focus has turned to understanding the function of genes and other parts of the genome, as well as understanding their regulation at a genome-wide scale, a field known as *functional genomics*.

The central dogma of molecular biology states that the genetic information in the DNA is *transcribed* into portable messenger RNA (mRNA) molecules that are subsequently *translated* into proteins. While the DNA is viewed as a storage device for genetic instructions, proteins actually execute these instructions in several forms such as enzymes, transcription factors, structural elements, immunoglobulins, hormones and signaling molecules.

A deoxyribonucleic acid (DNA) molecule is a repeating chain composed of four different nucleotides: adenine (`A`), guanine (`G`), cytosine (`C`) and thymine (`T`). DNA molecules are structurally organized in duplexes consisting of two helical DNA molecules coiled around a common axis, forming a structure known as the double helix. The messenger ribonucleic acid (mRNA) is a copy of a segment of one DNA strand with uracil (`U`) replacing thymine (`T`). The basic building blocks for the proteins are the amino acids. There are 22 amino acids naturally occurring in plants, animals and bacteria. The sequence that forms a protein is coded directly in the mRNA in terms of successive groups of three nucleotides called *codons*. The *genes* are the RNA-encoding segments of the DNA, and they are said to be *expressed* in a cell when they are transcribed. The the set of all mRNA molecules, or transcripts, produced in one or a population of cells is called *transcriptome*.

To meet the challenge posed by functional genomics, new and highly ingenious experimental techniques have been developed. Among them, microarrays have emerged as the method of choice for large-scale gene expression studies because they provide an efficient and rapid method to investigate the entire transcriptome of a cell.

The complementary nature of the DNA double helix is the basis for the large-scale measurement of mRNA levels with microarrays. Under the right conditions, two complementary nucleic acid molecules (or *strands*) combine to form double stranded helices, a reaction know as *hybridization*. This principle allows the use of selected DNA strands with a known sequence of nucleotides (the *probes*) to query complex populations of unidentified, complementary strands (the *targets*).

1.1 High-density oligonucleotide microarrays

Several microarray technologies are available today, based on a variety of fabrication techniques including printing with fine-pointed pins onto glass slides, ink-jet printing, electrochemistry on microelectrode arrays and photolithography. This thesis is mainly concerned with the production of *high-density oligonucleotide microarray*, sometimes called DNA chips, that are fabricated by photolithography.

This type of microarray consists of relatively short DNA probes synthesized at specific locations, called *features* or *spots*, of a solid surface. Each probe is a single-stranded DNA molecule of 10 to 70 nucleotides that perfectly matches with a specific part of a target molecule. The probes are used to verify whether (or in which quantity) the targets are present in a given biological sample.

The first step of a microarray experiment consists of collecting mRNAs or genomic DNA from the cells or tissue under investigation. The mixture to be analyzed is prepared with fluorescent tags and loaded on the array, allowing the targets to hybridize with the probes. Any unbound molecule is washed away, leaving on the array only those molecules that have found a complementary probe. Finally, the array is exposed to a light source that induces fluorescence, and an optical scanner reads the intensity of light emitted at each spot.

Under ideal conditions, each probe will hybridize only to its target. Thus, it is possible to infer whether a given molecule is present in the sample by checking whether there is light coming from the corresponding spot of the array. The expression level of a gene in a cell can also be inferred because each spot contains several million identical probes, and the strength of the fluorescent signal on a spot is expected to be proportional to the concentration of the target in the sample. In practice, each target is queried by several probes (called *probe set*), and complex statistical calculations are performed to infer the concentration from the observed signals.

Microarrays have been extensively used for cellular gene expression monitoring and profiling (Schena et al., 1995; Lockhart et al., 1996) with diverse applications such as discovery of gene functions (Cho et al., 1998; Hughes et al., 2000), drug target identification and validation (Marton et al., 1998; Liotta and Petricoin, 2000), analysis of drug response (Debouck and Goodfellow, 1999), classification of clinical samples

(Perou et al., 1999) and detection of splicing variants (Hu et al., 2001). Microarrays are also used for genotypic analysis, in two main areas: SNPs analysis, and mutation and variant detection. Single nucleotide polymorphisms (SNP) are the most common source of genetic variation and, in fact, large number of SNPs have been discovered using microarrays (Lindblad-Toh et al., 2000). Special mutation detection arrays have also been used, for instance, to identify HIV variants (Kozal et al., 1996).

The advantage of high-density oligonucleotide microarrays is that thay can have more than a million spots, and are thus able to query tens of thousands of genes, possibly covering the entire genome of an organism. This type of microarray was originally designed in the late 1980s as a tool for DNA sequencing, a technology that is known as Sequencing by Hybridization (SBH). Today, the pioneering Affymetrix GeneChip® arrays, for instance, have up to 1.3 million spots on a coated quartz substrate measuring a little over 1 cm ^{2}. The spots are as narrow as 5 μm (5 microns, or 0.005 mm), and are arranged in a regularly-spaced rectangular grid (McGall and Christians, 2002).

1.1.1 Photolithography

GeneChip arrays are produced by combinatorial chemistry and techniques derived from micro-electronics and integrated circuits fabrication. Probes are typically 25 bases long and are synthesized on the chip, in parallel, in a series of repetitive steps. Each step appends the same kind of nucleotide to probes of selected regions of the chip. The sequence of nucleotides added in each step is called *deposition sequence* or *synthesis schedule*. The selection of which probes receive the nucleotide is achieved by photolithography (Fodor et al., 1991, 1993; Lipshutz et al., 1999).

Figure 1.1 illustrates this process: The quartz wafer of a GeneChip array is initially coated with a chemical compound topped with a light-sensitive protecting group that is removed when exposed to ultraviolet light, activating the compound for chemical coupling. A lithographic mask is used to direct light and remove the protecting groups of only those positions that should receive the nucleotide of a particular synthesis step. A solution containing adenine (`A`), thymine (`T`), cytosine (`C`) or guanine (`G`) is then flushed over the chip surface, but the chemical coupling occurs only in those positions that have been previously deprotected. Each coupled nucleotide also bears another protecting group so that the process can be repeated until all probes have been fully synthesized.

Photolithographic masks are notoriously expensive and cannot be changed once they have been manufactured. Thus, any change in the chip layout requires the production of a new set of masks. A similar method of *in situ* synthesis known as Maskless Array Synthesizer (MAS) was later developed to eliminate the need of such masks (Singh-Gasson et al., 1999). Probes are still built by repeating cycles of deprotection

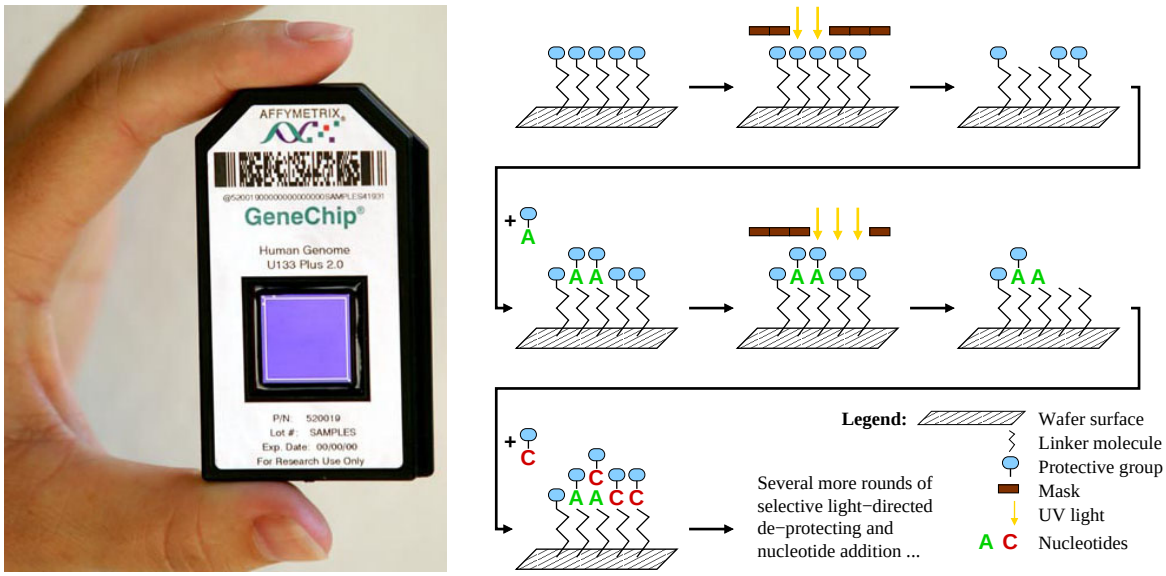


Figure 1.1: Left: Affymetrix GeneChip array (source: Affymetrix, Inc.). Right: probe synthesis via photolithographic masks. The chip is coated with a chemical compound and a light-sensitive protecting group; masks are used to direct light and activate selected probes for chemical coupling; nucleotides are appended to deprotected probes; the process is repeated until all probes have been fully synthesized.

and chemical coupling of nucleotides. The illumination, however, relies on an array of miniature mirrors that can be independently controlled to direct or deflect the incidence of light on the chip.

NimbleGen Systems, Inc. currently uses its Maskless Array Synthesizer (MAS) technology based on its own Digital Micromirror Device (DMD) similar to Texas Instruments' Digital Light Processor (DLP) that can control 786 000 to 4.2 million individual pixels of light to produce microarrays with spots as small as 16 μm $\times$ 16 μm (Nuwaysir et al., 2002). The geniom® system of febit biotech GmbH, a highly-automated self-contained platform for customized microarray production, also uses a micromirror array to direct the synthesis process (Baum et al., 2003). Recently, the same technology has also been used to synthesize arrays of peptides using 20 natural amino acids as well as synthetic amino acid analogs (Pellois et al., 2002; Gao et al., 2003; Li et al., 2004; Bhushan, 2006).

1.1.2 The unintended illumination problem

Regardless of which method is used to direct light (masks or micromirror arrays), it is possible that some probes are accidentally activated for chemical coupling because

of light diffraction, scattering or internal reflection on the chip surface. This unwanted illumination of regions introduces unexpected nucleotides that change probe sequences, significantly reducing their chances of successful hybridization with their targets. Moreover, these faulty probes may also introduce cross-hybridizations, which can interfere in the experiments performed with the chip.

This problem is more likely to occur near the borders between a masked and an unmasked spot (in the case of maskless synthesis, between a spot that is receiving light and a spot that is not). This observation has given rise to the term *border conflict*.

It turns out that by carefully designing the *arrangement* of the probes on the chip and their *embeddings* (the sequences of masked and unmasked steps used to synthesize each probe), it is possible to reduce the risk of unintended illumination. This issue becomes even more important as there is a need to accommodate more probes on a single chip, which requires the production of spots at higher densities and, consequently, with reduced distances between probes.

1.2 Manufacturing and design problems

The main focus of this thesis is to design the layout of a microarray in such a way that we minimize the incidence of the unintended illumination problem, what we call the *microarray layout problem* (MLP). The MLP is the focus of Chapters 2 to 8. A related problem is the *shortest deposition sequence problem*, which attempts to find the shortest deposition sequence to synthesize a given set of probes in order to reduce manufacturing time, cost and probability of errors. This problem is studied in more details in Chapter 9.

We conclude this chapter by briefly describing other interesting mathematical and computational problems that arise in the design and production of oligonucleotide microarrays. Recently, Kahng et al. (2003c, 2006) and Atlas et al. (2004) proposed methodologies to integrate the various steps in the design of a microarray chip, including probe selection, deposition sequence design and, ultimely, layout design.

Probe selection. Although a probe should only hybridize to its target, it is known that, in practice, cross-hybridizations are likely to occur. The goal of the probe selection problem is to find the smallest number of probes with the specified length covering all gene of interest satisfying the three criteria: homogeneity, sensitivity and specificity as proposed by (Lockhart et al., 1996). Homogeneity ensures that can hybridize to their targets at about the same experimental temperature. Sensitivity

detects self-complementarity and prevents probes with secondary structures. Specificity ensures that probes are unique to each gene and eliminates probes that could cross-hybridize.

This problem has been extensively studied in the past few years (Li and Stormo, 2001; Kaderali and Schliep, 2002), and many algorithms have been proposed to speed up the specificity check, regarded as the most computational intensive step (Rahmann, 2002; Sung and Lee, 2003; Chou et al., 2004). Among the presented approaches, Rahmann (2002) proposed a fast algorithm based on suffix arrays (Manber and Myers, 1990) that eliminates candidates that have a long common factor with other genes.

Mask decomposition problem. Once the probes have been selected and the layout of the chip has been designed, the photolithographic masks must be produced. The masks used by Affymetrix are fabricated by a series of "flashes", with each flash producing a rectangular part of the mask. The cost of a mask is directly proportional to the number of flashes (Hubbell and Stryer, 1998; Hubbell et al., 1999) and, in fact, there may be a limit in the number of flashes before a more expensive fabrication technology must be used. Ideally, each mask must be decomposed in the minimum number of rectangles in order to reduce costs and incidence of errors.

Hannenhalli et al. (2002) studied this problem, called the mask decomposition problem, as an instance of the rectilinear polygon interior cover problem, which, according to Garey and Johnson (1979) was first shown to be NP-hard by Masek (Unpublished manuscript). Although approximation algorithms with small performance ratios are known (Franzblau and Kleitman, 1986), Hannenhalli et al. (2002) explored the particular characteristics of photolithographic masks to devise an efficient algorithm which found provably optimal decompositions for a set of relatively small GeneChip arrays.

Probe quality control. During the production of a microarray chip, it is possible that one synthesis step may be entirely compromised, resulting in damages to all probes that receive the nucleotide of that particular step, and, consequently, invalidating any experimental result obtained with the chip. In order to detect such failures, Affymetrix have introduced the idea of producing a set of *quality control probes* (QC) on their chips (Affymetrix, Inc., 2002). Target molecules for each QC probe are deliberately added to the biological mixture during the experiment with the chip. If no synthesis step fails, the QC probes should exhibit similar signal intensities. Thus, by measuring the fluorescent signal emitted by each QC probe, it is possible to infer if they have been correctly synthesized or not.

In fact, several copies of each quality control probe are produced on different spots of the chip using different synthesis schedules (embeddings) in such a way that it is possible to check if a synthesis step was compromised (Hubbell and Pevzner, 1999) (and maybe even identify systematic problems in the chip production). However,

the validation proposed by Hubbell and Pevzner (1999) does not take into account possible defects on isolated spots containing QC probes caused by other manufacturing problems. For this reason, robust schemes based on a combinatorial design approach that guarantee coverage of all synthesis steps and that are able to tolerate a great number of unreliable QC probes have been proposed (Alon et al., 2001; Sengupta and Tompa, 2002; Colbourn et al., 2002; Khan et al., 2003).

Chapter 2

The Microarray Layout Problem

In this chapter we give a more precise definition of the microarray layout problem (MLP) and define criteria for evaluating a given layout. The description that follows assumes that probes are synthesized with photolithographic masks, but the concepts also apply to the maskless production (with micromirror arrays). Two evaluation criteria are presented: *border length* and *conflict index*. As shown later, the conflict index model can be seen as a generalization of the border length model.

Formally, we have a set of probes $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, where each $p_k \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}^*$ with $1 \leq k \leq n$ is produced by a series of T synthesis steps. Frequently, but not necessarily, all probes have the same length ℓ. Each synthesis step t uses a mask M_t to induce the addition of a particular nucleotide $N_t \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ to a subset of $\mathcal{P}$ (Figure 2.1). The *nucleotide deposition sequence* $N = N_1 N_2 \ldots N_T$ corresponding to the sequence of nucleotides added at each synthesis step is a supersequence of all $p \in \mathcal{P}$.

A microarray chip consists of a set of spots, or sites, $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$, where each spot s is specified by its coordinates on the chip surface and accommodates a unique probe $p_k \in \mathcal{P}$. Note that we usually refer to s as containing a single probe p_k although, in practice, it contains several million copies of it. Each probe is synthesized at a unique spot, hence there is a one-to-one assignment between probes and spots (if we assume that there are as many spots as probes, i.e., $m = n$). Real microarrays may have complex physical structures but we assume that the spots are arranged in a rectangular grid with n_r rows and n_c columns. We also assume that probes can be assigned to any spot.

In general, a probe can be *embedded* within N in several ways. An embedding of p_k is a T-tuple $\varepsilon_k = (\varepsilon_{k,1}, \varepsilon_{k,2}, \ldots, \varepsilon_{k,T})$ in which $\varepsilon_{k,t} = 1$ if probe p_k receives nucleotide N_t (at step t), and 0 otherwise. In particular, a *left-most embedding* is an embedding in which the bases are added as early as possible (as in ε_1 in Figure 2.1). Similarly, a *right-most embedding* is an embedding in which the bases are added as late as possible (as in ε_8 in Figure 2.1).

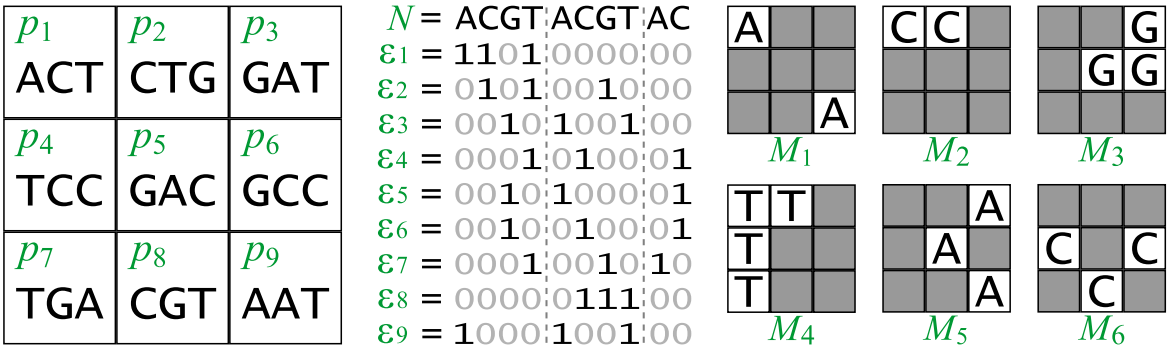


Figure 2.1: Synthesis of a hypothetical 3×3 chip with photolithographic masks. Left: chip layout and the 3-mer probe sequences. Center: deposition sequence with 2.5 cycles (cycles are delimited with dashed lines) and probe embeddings (asynchronous). Right: first six masks (masks 7 to 10 not shown).

We say that an embedding ε_k is *productive* (unmasked) at step t if $\varepsilon_{k,t} = 1$, or *unproductive* (masked) otherwise. The terms productive and unproductive can also be used to denote unmasked and masked spots, respectively.

The deposition sequence is often a repeated permutation of the alphabet, mainly because of its regular structure and because such sequences maximize the number of distinct subsequences (Chase, 1976). The deposition sequence shown in Figure 2.1 is a 2.5-time repetition of `ACGT`, and we thus say that it has two and a half *cycles*.

For cyclic deposition sequences, it is possible to distinguish between two types of embeddings: *synchronous* and *asynchronous*. In the former, each probe has exactly one nucleotide added in every cycle of the deposition sequence; hence, 25 cycles or 100 steps are needed to synthesize probes of length 25. In the latter, probes can have any number of nucleotides added in any given cycle, allowing shorter deposition sequences. For this reason, asynchronous embeddings are usually the choice for commercial microarrays. For instance, all GeneChip arrays that we know of can be asynchronously synthesized in 74 steps with $N = (\texttt{TGCA})^{18}\texttt{TG}.$, i.e., 18.5 cycles of `TGCA` — we refer to this sequence as the *standard Affymetrix deposition sequence* (see Chapter 8).

Ideally, the deposition sequence should be as short as possible in order to reduce manufacturing time, cost and probability of errors (Rahmann, 2003). Finding the shortest deposition sequence to synthesize a set of probes is an instance of a classical computer science problem known as the shortest common supersequence problem, which will be the focus of Chapter 9. For the MLP, however, we assume that N is a fixed sequence given as input.

2.1 Problem statement

Given a set of probes $\mathcal{P}$, a geometry of spots $\mathcal{S}$, and a deposition sequence N as specified above, the MLP asks to specify a chip layout (λ, ε) that consists of

1. a bijective assignment $\lambda : \mathcal{S} \to \{1, \ldots, n\}$ that specifies a probe index $k(s)$ for each spot s (meaning that $p_{k(s)}$ will be synthesized at s),
2. an assignment $\varepsilon : \{1, \ldots, n\} \to \{0, 1\}^T$ specifying an embedding $\varepsilon_k = (\varepsilon_{k,1}, \ldots, \varepsilon_{k,T})$ for each probe index k, such that $N[\varepsilon_k] :\equiv (N_t)_{t:\varepsilon_{k,t}=1} = p_k$,

such that a given penalty function is minimized. We introduce two such penalty functions: total border length and total conflict index.

2.2 Border length

The first formal definition of the unintended illumination problem was given by Hannenhalli et al. (2002), who defined the *border length* $\mathcal{B}_t$ of a mask M_t as the number of borders separating masked and unmasked spots at synthesis step t, that is, the number of border conflicts in M_t. Formally,

$$\mathcal{B}_t := \frac{1}{2} \cdot \sum_{s,s' \in \mathcal{S}} \mathbb{1}_{\{s \text{ and } s' \text{ are adjacent}\}} \cdot \mathbb{1}_{\{\varepsilon_{k(s),t} \neq \varepsilon_{k(s'),t}\}}. \tag{2.1}$$

where $\mathbb{1}_{\{cond\}}$ is the indicator function that equals 1 if condition *cond* is true, and 0 otherwise. The *total border length* of a given layout (λ, ε) is the sum of border lengths over all masks, that is

$$\mathcal{B}(\lambda, \varepsilon) := \sum_{t=1}^{T} \mathcal{B}_t. \tag{2.2}$$

The *border length minimization problem* was then defined as the problem of finding a layout minimizing the total border length (Hannenhalli et al., 2002). As an example, the six masks shown in Figure 2.1 have $\mathcal{B}_1 = 4$, $\mathcal{B}_2 = 3$, $\mathcal{B}_3 = 5$, $\mathcal{B}_4 = 4$, $\mathcal{B}_5 = 8$ and $\mathcal{B}_6 = 9$. The total border length of that layout is 52 (masks M_7 to M_{10} are not shown).

Hamming distance. In the next chapters, we refer to the *Hamming distance* $H(k, k')$ between the embeddings ε_k and $\varepsilon_{k'}$ as the number of synthesis steps in which they differ. Formally,

$$H(k, k') := \sum_{t=1}^{T} \mathbb{1}_{\{\varepsilon_{k,t} \neq \varepsilon_{k',t}\}}. \tag{2.3}$$

Note that $H(k, k')$ gives the number of border conflicts generated when probes with embeddings ε_k and $\varepsilon_{k'}$ are placed in adjacent spots.

2.2.1 Lower bounds

Lower bounds for the BLMP with synchronous and asynchronous embeddings were given by Kahng et al. (2002), based on a simple graph formulation. Unfortunately, both lower bounds are not tight, and their computation is time-consuming, especially for large chips.

Synchronous embeddings. Let L be a complete directed graph over the set of probes $\mathcal{P}$ with arcs weighted with the Hamming distance between the (unique) embeddings of the corresponding probes.

Since a probe can have at most four neighbors on the chip, we delete all but the four arcs with the least weights of every node. Furthermore, assuming that the chip is a rectangular grid with n_r rows and n_c columns, we delete the heaviest $2 \cdot (n_r + n_c)$ remaining arcs, because the spots on the borders of the chip have less than four neighbors. It is not difficult to see that the cost of any placement must be greather than the total arc weight of L, and we obtain the following theorem.

Theorem 2.1. *The total arc weight of L is a lower bound on the total border length of the optimum layout with synchronous embeddings.*

Asynchronous embeddings. With asynchronous embeddings, we can construct a similar complete directed graph L'. For the arc weights, however, it is necessary to estimate the minimum number of border conflicts between the two probes (among all of their possible embeddings).

Kahng et al. (2002) observed that the number of bases of a probe p_k that can be "aligned" with bases of $p_{k'}$ cannot exceed the length of $LCS(p_k, p_{k'})$, where $LCS(p_k, p_{k'})$ is the *longest common subsequence* of p_k and $p_{k'}$. Therefore, an arc of L' between probes p_k and $p_{k'}$ can be weighted with $\ell - |LCS(p_k, p_{k'})|$, where ℓ is the length of both probe sequences (assuming probes have the same length).

We can then delete all but the four arcs with the least weights of each probe and, subsequently, the heaviest $2 \cdot (n_r + n_c)$ remaining arcs of L', to obtain the following theorem.

Theorem 2.2. *The total arc weight of L' is a lower bound on the total border length of the optimum layout with asynchronous embeddings.*

2.3 Conflict index

The border length measures the quality of an individual mask or set of masks. With this model, however, it is not possible to know how the border conflicts are distributed among the probes. Ideally, all probes should have roughly the same risk of being damaged by unintended illumination, so that all signals are affected in approximately the same way.

The *conflict index* is a quality measure defined with the aim of estimating the risk of damaging probes at a particular spot (de Carvalho Jr. and Rahmann, 2006b); it is thus a per-spot or per-probe measure instead of a per-mask measure. Additionally, it takes into account two practical considerations observed by Kahng et al. (2003a):

a) stray light might activate not only adjacent neighbors but also spots that lie as far as three cells away from the targeted spot;

b) imperfections produced in the middle of a probe are more harmful than in its extremities.

For a proposed layout (k, ε), the conflict index $\mathcal{C}(s)$ of a spot s whose probe $p_{k(s)}$ is synthesized in T masking steps according to its embedding vector $\varepsilon_{k(s)}$ is

$$\mathcal{C}(s) := \sum_{t=1}^{T} \Bigg(\mathbb{1}_{\{\varepsilon_{k(s),t}=0\}} \cdot \omega(\varepsilon_{k(s)}, t) \cdot \sum_{\substack{s' \text{: neighbor} \\ \text{of } s}} \mathbb{1}_{\{\varepsilon_{k(s'),t}=1\}} \cdot \gamma(s, s') \Bigg). \tag{2.4}$$

The indicator functions ensure the following conflict condition: During step t, there is a conflict at spot s if and only if s is masked ($\varepsilon_{k(s),t} = 0$) and a close neighbor s' is unmasked ($\varepsilon_{k(s'),t} = 1$) — since light directed at s' may somehow reach s. When s is unmasked, it does not matter if it accidentally receives light targeted at a neighbor, and when s' is masked, there is no risk that it damages probes of s since it is not receiving light.

Function $\gamma(s, s')$ is a "closeness" measure between s and s' (to account for observation a). We define it as

$$\gamma(s, s') := (d(s, s'))^{-2}, \tag{2.5}$$

where $d(s, s')$ is the Euclidean distance between the spots s and s'. Note that in (2.4), s' ranges over all neighboring spots that are at most three cells away from s (see Figure 2.2, left), which is in accordance with observation a. In general, we use the terms *close neighbor* or simply *neighbor* of a spot s to refer to a spot s' that is at most three cells away (vertically and horizontally) from s. In other words, s' is inside a 7×7 region centered around s. This is in contrast to the terms *direct* or *immediate neighbor* of s, used to denote a spot s' that is adjacent to s (in other words, when s'

0.06	0.08	0.10	0.11	0.10	0.08	0.06
0.08	0.13	0.20	0.25	0.20	0.13	0.08
0.10	0.20	0.50	1.00	0.50	0.20	0.10
0.11	0.25	1.00	s	1.00	0.25	0.11
0.10	0.20	0.50	1.00	0.50	0.20	0.10
0.08	0.13	0.20	0.25	0.20	0.13	0.08
0.06	0.08	0.10	0.11	0.10	0.08	0.06

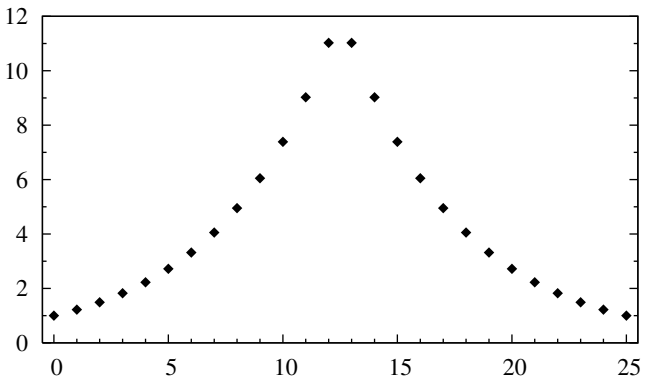


Figure 2.2: Ranges of values for both γ and ω on a typical Affymetrix chip where probes of length $\ell = 25$ are synthesized in $T = 74$ masking steps. Left: approximate values of the distance-dependent weighting function $\gamma(s, s')$ for a spot s in the center and close neighbors s'. Right: position-dependent weights $\omega(\varepsilon, t)$ on the y-axis for each value of $b_{\varepsilon,t} \in \{0, \ldots, 25\}$ on the x-axis, using $\theta = 5/\ell$ and $c = 1/\exp(\theta)$.

shares a common border with s on the chip). Obviously, an immediate neighbor s' is also a close neighbor of s.

The position-dependent weighting function $\omega(\varepsilon, t)$ accounts for the significance of the location inside the probe where the undesired nucleotide is introduced in case of accidental illumination (observation b). We defined it as:

$$\omega(\varepsilon, t) := c \cdot \exp(\theta \cdot \lambda(\varepsilon, t)) \tag{2.6}$$

where $c > 0$ and $\theta > 0$ are constants, and for $1 \leq t \leq T$,

$$\lambda(\varepsilon, t) := 1 + \min(b_{\varepsilon,t}, \ell_\varepsilon - b_{\varepsilon,t}), \tag{2.7}$$

$$b_{\varepsilon,t} := \sum_{t'=1}^{t} \varepsilon_{t'}, \qquad \ell_\varepsilon := \sum_{t=1}^{T} \varepsilon_t = b_{\varepsilon,T}. \tag{2.8}$$

In other words, ℓ_ε is the length of the final probe specified by ε (equal to the number of ones in the embedding), and $b_{\varepsilon,t}$ denotes the number of nucleotides added up to and including step t. The parameter θ controls how steeply the exponential weighting function rises towards the middle of the probe (Figure 2.2, right). In our experiments, unless stated otherwise, we use probes of length $\ell = 25$, and parameters $\theta = 5/\ell$ and $c = 1/\exp(\theta)$. We can now speak of the *total conflict index* of a given layout (λ, ε) as the sum of conflict indices over all spots, that is

$$\mathcal{C}(\lambda, \varepsilon) := \sum_{s} \mathcal{C}(s). \tag{2.9}$$

Conflict index distance. Many of the algorithms discussed in later chapters were initially developed for border length minimization, and they usually rely on the Hamming distance defined earlier (2.3). We have adapted some of these algorithms to work with conflict index minimization by using the *conflict index distance*, which extends the Hamming distance by taking into account the position inside the probe where the conflict occurs (observation b). The conflict index distance $C(k, k')$ between the embeddings ε_k and $\varepsilon_{k'}$ is defined as:

$$C(k,k') := \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{k,t}=0 \text{ and } \varepsilon_{k',t}=1\}} \cdot \omega(\varepsilon_k, t) + \mathbb{1}_{\{\varepsilon_{k',t}=0 \text{ and } \varepsilon_{k,t}=1\}} \cdot \omega(\varepsilon_{k'}, t) \Big). \quad (2.10)$$

The conflict index distance $C(k, k')$ can be interpreted as the sum of the conflict indices resulting from placing probes with embeddings ε_k and $\varepsilon_{k'}$ at hypothetical neighboring spots, ignoring the distance between these spots (note that there is no dependency on γ) and the conflicts generated by other neighbors.

2.3.1 The choices of γ and ω

The conflict index $\mathcal{C}(s)$ attempts to estimate the risk of damaging the probes of a spot s due to unintended illumination. The definitions of γ and ω given here are an arbitrary choice in an attempt to capture the characteristics of the problem.

However, the most appropriate choice of γ depend on several attributes of the specific technology utilized to produce the chips such as the size of the spots, the density of the probes on the chip, the physical properties of the light being used (intensity, frequency, etc.), the distance between the light source and the mask, and the distance between the mask (or the micromirrors) and the chip surface.

The most appropriate choice of ω depend on the chemical properties of the hybridization between probes and targets. Although it is generally agreed that the chances of a successful hybridization are higher if a mismatched base occurs at the extremities of the formed duplex instead of at its center (Hubbell et al., 1999; Southern et al., 1999; Guo et al., 1997), the precise effects of this position is not yet fully understood and has been an active topic of research (Binder et al., 2004; Binder and Preibisch, 2005).

We propose the use of an exponential function, so that ω grows exponentially from the extremities of the probe to its center (see Figure 2.2, right). The motivation behind this definition is that the probability of a successful stable hybridization of a probe with its target should increase exponentially with the absolute value of its Gibbs free energy, which increases linearly with the length of the longest perfect match between probe and target.

Finding the best choice of γ and ω for a particular technology is beyond the scope of this thesis. We note, however, that all algorithms discussed in the next chapters were developed to work independently of the values given by these functions. In other words, should γ and ω be defined differently, no changes to the algorithms are necessary.

2.4 Chip quality measures

Most of the algorithms discussed in the next chapters can work with border length as well as conflict index minimization. In our experiments, we will usually present results with both measures, making a distinction between border length minimization (BLM) and conflict index minimization (CIM).

The relation between these two measures becomes clear if $\gamma(s, s')$ and $\omega(\varepsilon, t)$ are redefined as follows: Set $\gamma(s, s') := 1$ if s' is a direct neighbor of s, and $:= 0$ otherwise. Also, set $c = 1/2$ and $\theta = 0$ so that $\omega(\varepsilon, t) := 1/2$ independently of the position in the probe where the conflict occurs. Now $\sum_s \mathcal{C}(s) = \sum_{t=1}^{T} \mathcal{B}_t$; that is, total border length is equivalent to the total conflict index for a particular choice of γ and ω. For the choices (2.5) and (2.6), they are not equivalent but still correlated, since a good layout has low border lengths as well as low conflict indices.

To better compare border lengths for chips of different sizes, we usually divide the total border length by the number n_b of internal borders of the chip, which equals $n_r(n_c - 1) + n_c(n_r - 1)$ if the the chip is a rectangular grid with n_r rows and n_c columns. We thus call $\mathcal{B}(\lambda, \varepsilon)/n_b$ the *normalized border length*, NBL for short, of a given layout (λ, ε). This can be further divided by the number of synthesis steps to give the *normalized border length per mask* $\mathcal{B}(\lambda, \varepsilon)/(n_b \cdot T)$. We may also refer to the normalized border length of a particular mask M_t as B_t/n_b. Since $B_t \leq n_b$, $B_t/n_b \leq 1$ and thus $\mathcal{B}(\lambda, \varepsilon)/n_b \leq T$.

Similarly, it is useful to divide the total conflict index by the number of probes on the chip, and we define the *average conflict index*, ACI for short, of a layout as $\mathcal{C}(\lambda, \varepsilon)/|\mathcal{P}|$.

2.5 How hard is the microarray layout problem?

The MLP appears to be hard because of the super-exponential number of possible arrangements, although no NP-hardness proof is yet known. A formulation of the MLP as a quadratic assignment problem (QAP) is given in Chapter 4. The QAP is a classical combinatorial optimization problem that is, in general, NP-hard, and particularly hard to solve in practice (Çela, 1997). Optimal solutions are thus unlikely

to be found even for small chips and even if we assume that all probes have a single predefined embedding.

If we consider all possible embeddings (up to several million for a typical Affymetrix probe), the MLP is even harder. For this reason, the problem has been traditionally tackled in two phases. First, an initial embedding of the probes is fixed and an arrangement of these embeddings on the chip with minimum conflicts is sought. This is usually referred to as the *placement* phase. Second, a post-placement optimization phase *re-embeds* the probes considering their location on the chip, in such a way that the conflicts with neighboring spots are further reduced. Often, the chip is *partitioned* into smaller sub-regions before the placement phase in order to reduce running times, especially on larger chips.

The most important placement algorithms are surveyed in Chapter 3, whereas re-embedding algorithms are discussed in Chapter 5. Partitioning algorithms are the focus of Chapter 6. Finally, we present recent developments that simultaneously place and re-embed probes in Chapter 7.

Chapter 3

Placement Algorithms

The input for a placement algorithm consists of a geometry of spots $\mathcal{S}$, the deposition sequence N, and a set of probes $\mathcal{P}$, where each probe is assumed to have at least one embedding in N. The output is a one-to-one assignment λ of probes to spots. If there are more spots than probes to place, one can add enough "empty" probes that do not introduce any conflicts with the other probes (since light is never directed to their spots).

All algorithms discussed in this section assume that an initial embedding of the probes is given, which can be a left-most, right-most, synchronous or otherwise pre-computed embedding — a placement algorithm typically does not change the given embeddings.

3.1 Optimal masks for uniform arrays

Feldman and Pevzner (1994) were the first to formally address the unintended illumination problem. They showed how a placement for a *uniform array* with minimum number of border conflicts can be constructed using a two-dimensional Gray code. Uniform arrays are arrays containing all 4^ℓ probes of a given length ℓ, which require a deposition sequence of length $4 \cdot \ell$. These arrays were initially developed for the technique known as Sequencing by Hybridization (Southern et al., 1992).

In general, the term Gray code refers to an ordering of a set of elements in which successive elements differ in some pre-specified, usually small, way (Savage, 1997). The construction of Feldman and Pevzner is based on a two-dimensional Gray code composed of strings of length ℓ over a four-letter alphabet. It generates a $2^\ell \times 2^\ell$ array filled with ℓ-mer probes in which each pair of adjacent probes (horizontally or vertically) differs by exactly one letter. This construction is illustrated in Figure 3.1. An $(\ell + 1)$-mer array is constructed by first copying the ℓ-mer array into the upper left quadrant of the $(\ell + 1)$-mer array and reflecting it horizontally and vertically into the other three quadrants. The letter in front of the probes in the upper left quadrant

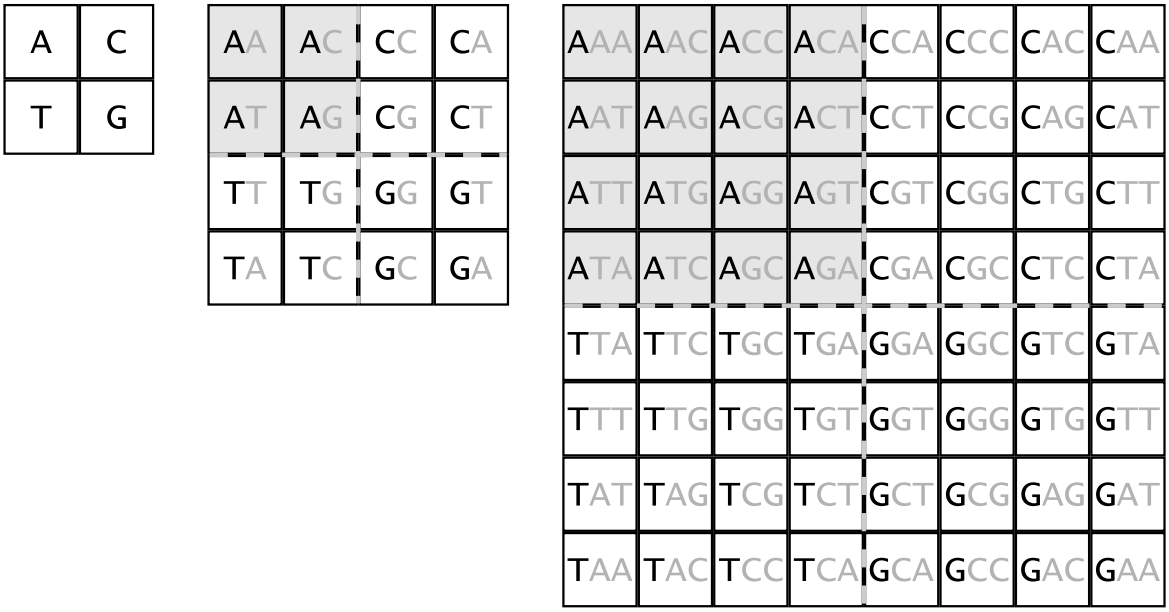


Figure 3.1: Construction of a placement for uniform arrays (containing the complete set of ℓ-mer probes) based on a two-dimensional Gray code, resulting in layouts with minimum number of border conflicts.

of the ℓ-mer array is added to all probes in the upper left quadrant of the $(\ell + 1)$-mer array. The probes of the other three quadrants are extended in the same way.

It can be shown that such placement generates masks with a minimum number of border conflicts if probes are synchronously embedded (see Figure 3.2). However, because this construction is restricted to uniform arrays and synchronous embeddings, it is of limited practical importance for current microarrays.

3.2 TSP and threading algorithms

The border length problem on arrays of arbitrary probes was first discussed by Hannenhalli et al. (2002). The article reports that the first Affymetrix chips were designed using a heuristic for the traveling salesman problem (TSP). The idea is to build a weighted graph with nodes representing probes, and edges containing the Hamming distances between their embeddings (see Equation 2.3). A TSP tour on this graph is heuristically constructed and *threaded* on the array in a row-by-row fashion (Figure 3.3a).

For uniform arrays, every solution of the TSP corresponds to a (one-dimensional) Gray code since consecutive elements in the tour differ in only one position, thus minimizing border conflicts between neighboring probes. For general arrays, a TSP solution also

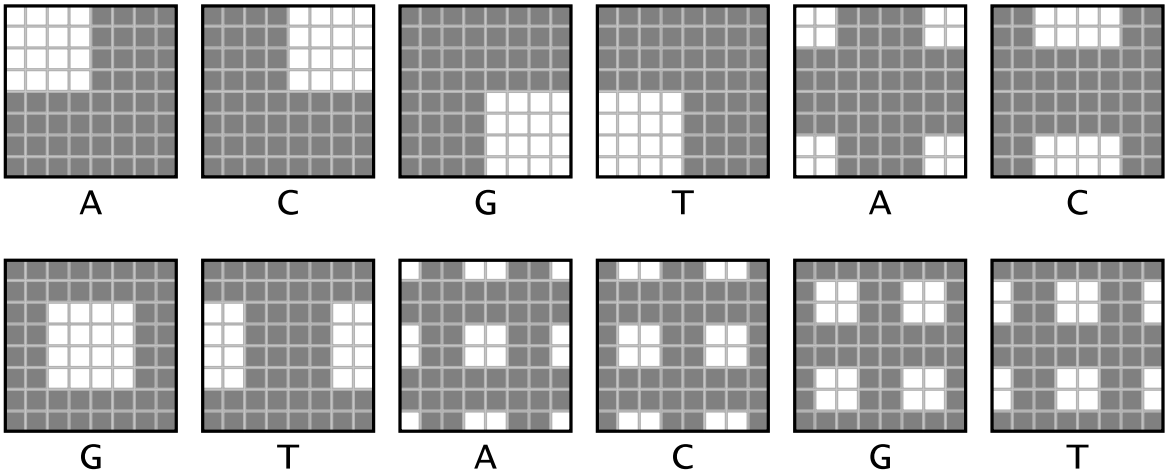


Figure 3.2: Masks for the 8×8 uniform array of Figure 3.1 when probes are synchronously embedded into $(\texttt{ACGT})^3$. Masked spots are represented by shaded squares, unmasked spots by white squares. Note that masks of the same cycle have the same number of border conflicts.

reduces border conflicts as consecutive probes in the tour are likely to be similar. Threading the (one-dimensional) tour on a two-dimensional chip, row-by-row, on a leads to an arrangement where consecutive probes in the same row have few border conflicts, but probes in the same column may have very different embeddings.

Another problem of this approach is that the TSP is known to be NP-hard (Gross and Yellen, 2004), so computing an optimal TSP tour even for a small 300×300 array is not feasible, and only fast approximation algorithms are suitable. In practice, Hannenhalli et. al. managed to achieve marginal improvements in tour cost using the 2-opt algorithm for TSP of Lin and Kernighan (1973) and an algorithm for weighted matching due to Gabow (1976). Unfortunately, their efforts resulted in only 1.05% reduction in tour cost for a chip with 66 000 probes when compared to the greedy TSP algorithm initially used at Affymetrix.

Since improvements in the cost of the TSP tour seemed unlikely, Hannenhalli et. al. turned their attention to the problem of threading the tour on the chip. They studied several threading alternatives, which they collectively called *k-threading* (Figure 3.3). A k-threading is a variation of the standard row-by-row threading, in which the right-to-left and left-to-right paths are interspaced with alternating upward and downward movements over k sites (the row-by-row threading can be seen as a k-threading with $k = 0$); k is called the *amplitude* of the threading. Hannenhalli et. al. experimentally observed that 1-threading may reduce total border length of layouts constructed with TSP tours in up to 20% for large chips when compared to row-by-row threading.

From now on, we will use the term TSP $+k$-threading to refer to the method of computing a TSP tour and threading it on the array using k-threading.

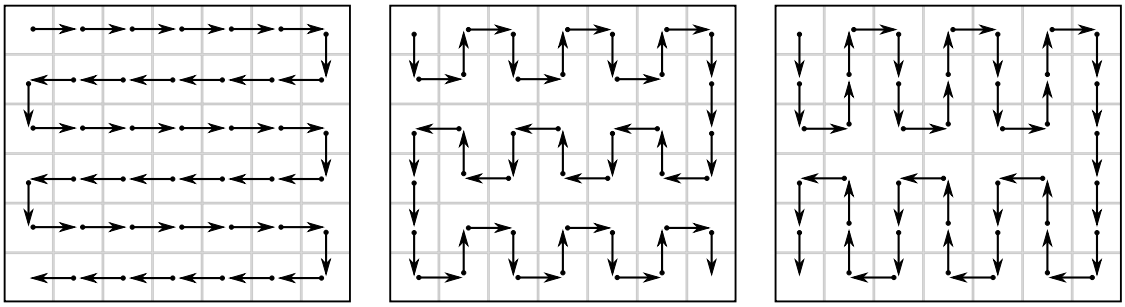


Figure 3.3: Different ways of *threading* probes on a chip. a) Standard row-by-row (0-threading); b) 1-threading; c) 2-threading.

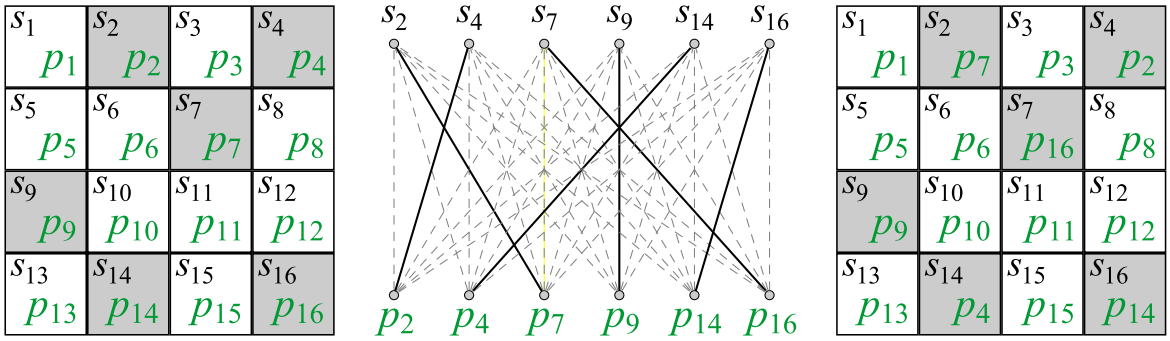
3.3 Epitaxial placement

A different strategy inspired by techniques used in the design of VLSI circuits, called Epitaxial placement, or *seeded crystal growth*, was proposed by Kahng et al. (2002). It essentially grows a placement around a single starting "seed" using a greedy heuristic. Although it was originally designed for chips with synchronous embeddings, it can be trivially implemented for asynchronous embeddings as well.

The algorithm starts by placing a random probe in the center of the array and continues to insert probes in spots adjacent to already-filled spots. Priority is given to spots whose all four neighbors are filled, in which case a probe with the minimum number of border conflicts with the neighbors is placed. Otherwise, all spots with $1 \leq i < 4$ filled neighbors are examined. For each spot s, the algorithm finds a non-assigned probe p whose number of border conflicts with the filled neighbors of s, $c(s, p)$, is minimal and assigns a normalized cost $\bar{c}(s, p) := \sigma_i \cdot c(s, p)/i$ for this assignment, where $0 < \sigma_i \leq 1$ are scaling coefficients (the authors propose $\sigma_1 = 1$, $\sigma_2 = 0.8$, and $\sigma_3 = 0.6$). The assignment with minimum $\bar{c}(s, p)$ is made and the procedure is repeated until all probes have been placed.

In order to avoid repeated cost computations, the authors propose keeping a list of probe candidates, for each spot, sorted by their normalized costs. This list must be updated whenever one of its neighbors is filled; thus, it is updated at most four times (but only two times on average).

With this algorithm, Kahng et. al. claim a further 10% reduction in border conflicts over the TSP + 1-threading approach of Hannenhalli et al. (2002). However, the Epitaxial algorithm has at least quadratic time complexity as it examines every non-placed probe to fill each spot, and large memory requirements if a list of probe candidates is kept for each spot. Hence, like the TSP approach, it does not scale well to large chips. In their experiments, the Epitaxial algorithm needed 274 seconds to design a 100×100 chip, but 4 441 seconds to design a 200×200 chip. That is a 16.2-fold



a) b) c)

Figure 3.4: Sliding-Window Matching algorithm. a) Initial arrangement of probes p_1 to p_{16} inside a 4×4 window (with spots s_1 to s_{16} and a selected maximal independent set of spots (shaded). b) Bipartite graph with selected probes and spots, and a minimum weight perfect matching (dark edges) resulting in a minimum cost re-assignment of probes to spots. c) New arrangement inside the window according to the perfect matching.

increase in running time for a 4-fold increase in number of spots. Chips of larger dimensions could not be computed because of prohibitively large running time and memory requirements.

3.4 Sliding-Window Matching

The Sliding-Window Matching algorithm (Kahng et al., 2003a), SWM for short, is not exactly a placement algorithm as it iteratively improves an existing placement that can be constructed, for instance, by TSP + 1-threading (Section 3.2).

The authors noted that the TSP tour can be conveniently substituted by lexicographically sorting the probe sequences or, alternatively, their binary embedding vectors with a linear-time radix sort. The sorting is faster, but it is also likely to produce a worse initial placement than the TSP tour, with consecutive embeddings being similar only in their first synthesis steps. The authors argue that this is of little importance in practice given that this placement is only used as a starting point for the SWM algorithm, and the lexicographical sorting should be the choice for large microarrays because computing a TSP tour takes prohibitively long for chips larger than 500×500 spots. (From now on, we will use the term sorting $+k$-threading, or simply k-threading, to refer to the method of sorting probes lexicographically and threading them on the array using k-threading.)

As its name implies, SWM works inside a window that starts at the top left of the chip and slides from left to right, top to bottom, while maintaining a certain amount of overlap between each iteration. When the window reaches the right-end of the chip,

it is re-started at the left-end of the next set of rows, also retaining an overlap with the preceding set of rows.

At each iteration, the algorithm attempts to reduce the total border length inside the window by relocating some of its probes (Figure 3.4a). First, a random maximal independent set of spots is selected, and the probes assigned to these spots are removed. The term independent refers to the fact that selected spots can be re-assigned to probes without affecting the border length of other selected spots. The algorithm then creates a bipartite graph with nodes representing the removed probes and the now vacant spots (Figure 3.4b). The edges of this graph are weighted with the number of border conflicts that are generated by the corresponding assignment. Finally, a minimum weight perfect matching on this graph is computed, and the indicated assignments are made (Figure 3.4c).

A minimum weight perfect matching requires polynomial time (Gross and Yellen, 2004), but the small graphs generated by SWM can be computed rather quickly. The authors experimentally observed that the best results are obtained with small window sizes (e.g. 6×6) and an overlap of half the window size. Moreover, employing less effort in each window and executing more cycles of optimization gives better results than more effort in each window and less cycles.

Selecting an independent set of spots ensures that the cost of each new assignment can be computed independently of the other assignments. The SWM was designed for border length minimization (BLM) and it takes advantage of the fact that, in this model, an independent set of spots can be constructed by selecting spots that do not share a common border. SWM can be adapted for conflict index minimization (CIM) by using larger windows containing relatively sparse independent sets (to our knowledge, this has not been implemented yet). Therefore several random independent sets should be constructed before moving the window.

3.5 Row-Epitaxial

Row-Epitaxial (Kahng et al., 2003a) is a variant of the Epitaxial algorithm with two main differences introduced to improve scalability: i) spots are filled in a pre-defined order, namely, from top to bottom, left to right, and ii) only a limited number Q of probe candidates are considered for filling each spot.

Like SWM, Row-Epitaxial improves an initial placement that can be constructed by, for example, sorting + 1-threading. For each spot s with a probe p, it looks at the next Q probes that lie in close proximity (to the right or below s), and swaps p with the probe that generates the minimum number of border conflicts between s and its left and top neighbors.

In the experiments conducted by Kahng et al. (2003a, 2004), Row-Epitaxial was the best large-scale placement algorithm (for BLM), achieving up to 9% reduction in border conflicts over TSP + 1-threading, whereas SWM achieved slightly worse results but required significantly less time.

Row-Epitaxial can also be adapted to CIM by swapping a probe of a spot s with the probe candidate that minimizes the sum of conflict indices in a region around s restricted to those neighboring probes that are to the left or above s (those which have already found their final positions).

Table 3.1 shows the results of using Row-Epitaxial for both border length and conflict index minimization on chips with random probe sequences (uniformly generated). Probes were lexicographically sorted and left-most embedded into the standard 74-step Affymetrix deposition sequence and threaded on the array with k-threading. The resulting layouts were then used as a starting point for Row-Epitaxial.

Although Hannenhalli et al. (2002) suggested 1-threading for laying out a TSP tour on the chip, our results show that increasing the threading's amplitude from $k = 0$ to $k = 4$ usually improves the initial layout produced by sorting + k-threading, both in terms of border length and conflict index minimization. For example, increasing the amplitude from $k = 0$ to $k = 4$ reduced the normalized border length of the initial layout in up to 6.56% (from 23.6828 to 22.1279) and the average conflict index in up to 4.51% (from 689.6109 to 658.5097) on 800×800 chips.

However, the best initial layouts rarely led to the best final layout produced by Row-Epitaxial. With BLM the best results were usually achieved with $k = 0$, whereas with CIM there was no clear best value for k. In any case, the difference due to varying k for the threading were rather small for Row-Epitaxial — at most 0.78% in normalized border length (from 16.9760 with $k = 0$ to 17.1085 with $k = 4$) and 0.26% in average conflict index (from 448.0140 with $k = 0$ to 449.1653 with $k = 4$), both on a 800×800 chip with $Q = 5$K (we use "K" to denote a multiple of a thousand).

Our results also give further indication that the running time of Row-Epitaxial is approximately $O(Qn)$, i.e., linear in the chip size, where Q is a user-defined parameter that controls the number of probe candidades examined for each spot. In this way, solution quality can be traded for running time: More candidates yield better layouts but also demand more time.

3.6 Greedy

As discussed in the previous section, the best results obtained with Row-Epitaxial rarely came from the best initial layouts (produced by k-threading). This is probably because Row-Epitaxial ignores the probe order used by k-threading when it looks

Table 3.1: Normalized border length and average conflict index of layouts produced by Row-Epitaxial (Row-Eptx) on random chips of various dimensions, with initial layouts produced by sorting + k-threading. Running times are reported in minutes and include the time for k-threading and Row-Epitaxial. All results are averages over a set of five chips.

			Border length minimization			Conflict index minimization		
Dim.	Q	k	k-threading	Row-Eptx	Time	k-threading	Row-Eptx	Time
300 × 300	5K	0	24.9649	**18.2935**	1.1	701.8698	462.5194	4.9
		1	24.1235	18.2999	1.3	690.8091	**462.4656**	5.1
		2	23.8695	18.3072	1.2	685.5916	462.6394	4.6
		3	23.7993	18.3226	1.2	683.5980	462.5885	5.1
		4	**23.7588**	18.3279	1.3	**682.3542**	462.7775	5.1
	10K	0	24.9649	**18.1477**	2.8	701.8698	444.0354	9.7
		1	24.1235	18.1529	2.8	690.8091	444.0904	9.3
		2	23.8695	18.1519	2.9	685.5916	444.1960	10.0
		3	23.7993	18.1591	2.8	683.5980	**443.9850**	10.6
		4	**23.7588**	18.1603	2.9	**682.3542**	444.1745	9.8
	20K	0	24.9649	18.0274	7.2	701.8698	426.7824	18.9
		1	24.1235	18.0325	6.9	690.8091	426.8863	18.5
		2	23.8695	18.0277	6.6	685.5916	426.8832	19.3
		3	23.7993	**18.0272**	6.6	683.5980	426.8694	19.6
		4	**23.7588**	18.0321	7.5	**682.3542**	**426.6600**	20.2
500 × 500	5K	0	24.2693	**17.6000**	4.3	693.5428	456.2042	15.2
		1	23.3454	17.6095	4.1	682.2097	**456.1341**	15.2
		2	23.0797	17.6246	4.3	676.4884	456.5261	14.1
		3	22.9632	17.6474	3.8	672.8160	456.5337	14.1
		4	**22.9162**	17.6670	3.7	**671.2636**	456.8203	15.3
	10K	0	24.2693	**17.4503**	13.1	693.5428	438.7075	33.9
		1	23.3454	17.4523	12.8	682.2097	438.7379	33.6
		2	23.0797	17.4582	12.7	676.4884	**438.6477**	30.4
		3	22.9632	17.4685	12.5	672.8160	438.8183	30.8
		4	**22.9162**	17.4755	12.5	**671.2636**	438.9280	32.8
	20K	0	24.2693	17.3303	28.2	693.5428	421.1358	66.7
		1	23.3454	**17.3297**	29.0	682.2097	421.1580	63.6
		2	23.0797	17.3308	27.4	676.4884	421.1087	67.7
		3	22.9632	17.3344	27.4	672.8160	**420.9758**	65.1
		4	**22.9162**	17.3376	27.7	**671.2636**	421.0436	64.2
800 × 800	5K	0	23.6818	**16.9760**	12.2	689.6109	**448.0140**	36.9
		1	22.6092	16.9927	12.2	672.2254	448.1474	40.3
		2	22.3205	17.0187	11.7	664.9753	448.6130	38.6
		3	22.1958	17.0589	11.7	660.5923	448.9159	40.2
		4	**22.1279**	17.1085	12.0	**658.5097**	449.1653	40.0
	10K	0	23.6818	**16.8032**	37.0	689.6109	**432.2283**	88.2
		1	22.6092	16.8111	39.1	672.2254	432.5153	91.4
		2	22.3205	16.8235	37.7	664.9753	432.5031	85.8
		3	22.1958	16.8353	37.7	660.5923	432.6652	90.1
		4	**22.1279**	16.8622	39.0	**658.5097**	432.6980	91.9
	20K	0	23.6818	**16.6771**	83.1	689.6109	**415.6470**	174.2
		1	22.6092	16.6803	83.2	672.2254	415.7402	181.1
		2	22.3205	16.6851	83.4	664.9753	415.6622	179.7
		3	22.1958	16.6915	86.5	660.5923	415.7609	172.3
		4	**22.1279**	16.7007	80.3	**658.5097**	415.7951	190.9

for probe candidates to fill a certain spot (Row-Epitaxial always looks for candidates in the next Q spots, row-by-row, regardless of how probes were threaded on the array). Another possible disadvantage of the k-threading + Row-Epitaxial approach is that each swap made by Row-Epitaxial shuffles the probes in the not-changed spots, destroying the lexicographical order used during the threading.

In this section, we present a new placement algorithm, Greedy, that combines the Row-Epitaxial greedy heuristic and a k-threading filling strategy in a single phase, using a linked list of probes to maintain the probe order during the whole placement. Like Row-Epitaxial, Greedy fills the spots in a greedy fashion, i.e., for each spot s, it examines Q probe candidates and chooses the one that can be placed at s with minimum cost (Greedy can also be easily implemented for border length as well as for conflict index minimization).

There are two main differences to Row-Epitaxial. First, instead of (re-)filling spots row-by-row, spots are filled with k-threading (there is no need for an initial layout). Perhaps more importantly, Greedy sorts the probes lexicographically and keeps them in a doubly-linked list. This list is used to maintain the lexicographical order during placement. Moreover, it is also used to improve the chances of finding a candidate having fewer conflicts with the last placed probe (which will be its neighbor on the chip): Once a probe p is selected to fill a certain spot, it is removed from the list and the next search of candidates examines the probes around p's former position in the list, e.g., $Q/2$ probes to the left and to the right of p.

Table 3.2 shows the results of using Greedy for both border length and conflict index minimization on the same set of (random) chips that have been previously used for the experiments with Row-Epitaxial (Table 3.1). The best layouts were always achieved with $k = 0$. Interestingly, increasing the amplitude of the threading from $k = 1$ to $k = 4$ always improved the results in terms of border length. In terms of conflict index, increasing k from 1 to 3 worsened the results; in most cases, increasing it from 3 to 4 improved the results.

In terms of BLM, Greedy and Row-Epitaxial produced similar results, with the best layout of Greedy being sometimes marginally better and sometimes marginally worse than the best layout of Row-Epitaxial. In terms of CIM, however, Greedy was constantly and significantly better than Row-Epitaxial, achieving up to 5.65% reduction in average conflic index (from 415.6470 to 392.1786) on a 800×800 chip with $Q = 20$K.

In our results, Greedy was between 13.9% and 59.9% slower than Row-epitaxial in the BLM case (19.7% on average), and between 3.7% and 18.1% in the CIM case (only 5.6% on average). The difference between Row-epitaxial and Greedy drops in the CIM case because the extra time spent in computing the cost of each candidate is higher than in the BLM case, which reduces the impact of the time required to keep the doubly-linked list.

Table 3.2: Normalized border length (NBL) and average conflict index (ACI) of layouts produced by Greedy on random chips of various dimensions. The results of Row-Epitaxial on the same set of chips (Table 3.1) is shown for comparison. Running times in minutes.

			Border length minimization				Conflict index minimization			
			Row-Epitaxial		Greedy		Row-Epitaxial		Greedy	
Dim.	Q	k	NBL	Time	NBL	Time	ACI	Time	ACI	Time
300^2	5K	0	**18.2935**	1.1	**18.3182**	1.6	462.5194	4.9	**440.5166**	5.4
		1	18.2999	1.3	18.5037	1.6	**462.4656**	5.1	444.7837	5.3
		2	18.3072	1.2	18.4222	1.6	462.6394	4.6	446.8662	5.3
		3	18.3226	1.2	18.3863	1.6	462.5885	5.1	447.7464	5.0
		4	18.3279	1.3	18.3728	1.5	462.7775	5.1	447.6559	5.3
	10K	0	**18.1477**	2.8	**18.1830**	4.3	444.0354	9.7	**426.3480**	10.9
		1	18.1529	2.8	18.3912	4.7	444.0904	9.3	429.5617	11.3
		2	18.1519	2.9	18.3058	4.5	444.1960	10.0	431.7555	11.1
		3	18.1591	2.8	18.2732	4.6	**443.9850**	10.6	432.6821	11.3
		4	18.1603	2.9	18.2415	4.6	444.1745	9.8	432.3800	11.0
	20K	0	18.0274	7.2	**18.0576**	9.6	426.7824	18.9	**415.5003**	30.3
		1	18.0325	6.9	18.2813	9.2	426.8863	18.5	418.2357	21.3
		2	18.0277	6.6	18.1985	9.2	426.8832	19.3	419.4866	21.1
		3	**18.0272**	6.6	18.1617	9.5	426.8694	19.6	420.7345	20.1
		4	18.0321	7.5	18.1328	8.8	**426.6600**	20.2	420.7332	21.0
500^2	5K	0	**17.6000**	4.3	**17.5830**	5.8	456.2042	15.2	**432.3023**	15.9
		1	17.6095	4.1	17.7842	5.3	**456.1341**	15.2	437.2417	16.2
		2	17.6246	4.3	17.7087	5.3	456.5261	14.1	439.7432	15.6
		3	17.6474	3.8	17.6759	5.4	456.5337	14.1	441.3441	16.2
		4	17.6670	3.7	17.6561	5.4	456.8203	15.3	441.0668	16.1
	10K	0	**17.4503**	13.1	**17.4673**	15.8	438.7075	33.9	**415.6951**	35.4
		1	17.4523	12.8	17.6765	16.0	438.7379	33.6	419.7788	33.4
		2	17.4582	12.7	17.5936	16.8	**438.6477**	30.4	422.1943	36.3
		3	17.4685	12.5	17.5550	16.2	438.8183	30.8	424.0554	34.6
		4	17.4755	12.5	17.5324	15.7	438.9280	32.8	423.7936	35.2
	20K	0	17.3303	28.2	**17.3554**	33.3	421.1358	66.7	**401.4609**	67.1
		1	**17.3297**	29.0	17.5829	34.0	421.1580	63.6	404.9949	69.8
		2	17.3308	27.4	17.4939	34.1	421.1087	67.7	406.9576	67.8
		3	17.3344	27.4	17.4519	34.7	**420.9758**	65.1	408.5048	69.4
		4	17.3376	27.7	17.4273	33.7	421.0436	64.2	408.4556	68.4
800^2	5K	0	**16.9760**	12.2	**16.9124**	15.3	**448.0140**	36.9	**426.0757**	41.9
		1	16.9927	12.2	17.1259	15.5	448.1474	40.3	430.9759	42.7
		2	17.0187	11.7	17.0551	15.1	448.6130	38.6	433.8504	42.9
		3	17.0589	11.7	17.0214	15.4	448.9159	40.2	435.4797	43.3
		4	17.1085	12.0	17.0009	15.4	449.1653	40.0	435.2589	43.2
	10K	0	**16.8032**	37.0	**16.7951**	45.5	**432.2283**	88.2	**408.3982**	91.4
		1	16.8111	39.1	17.0122	43.2	432.5153	91.4	412.9971	95.1
		2	16.8235	37.7	16.9333	45.1	432.5031	85.8	415.7934	91.7
		3	16.8353	37.7	16.8935	45.9	432.6652	90.1	417.5229	95.2
		4	16.8622	39.0	16.8748	45.6	432.6980	91.9	417.5098	91.3
	20K	0	**16.6771**	83.1	**16.6980**	95.8	**415.6470**	174.2	**392.1786**	186.4
		1	16.6803	83.2	16.9263	93.2	415.7402	181.1	396.3923	185.9
		2	16.6851	83.4	16.8376	93.8	415.6622	179.7	399.0043	186.9
		3	16.6915	86.5	16.7947	94.4	415.7609	172.3	400.7189	183.3
		4	16.7007	80.3	16.7727	97.2	415.7951	190.9	400.7257	189.3

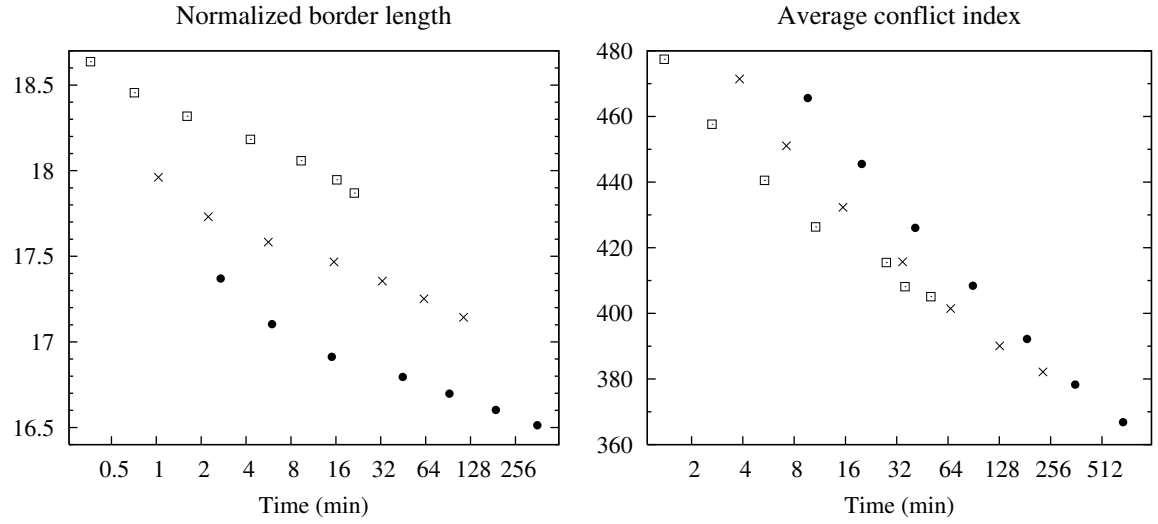


Figure 3.5: Trade-off between solution quality and running time (in logarithmic scale) with the Greedy algorithm on random chips of dimensions 300×300 (⊡), 500×500 (×) and 800×800 (•). The number Q of candidates per spot are 1.25K, 2.5K, 5K, 10K, 20K, 40K, and 80K (from left to right). Layouts are measured by normalized border length (left) and average conflict index (right).

It should be noted that, like Row-Epitaxial, Greedy has the drawback of treating the last spots of a chip "unfairly": While Q probe candidades are examined for each of the first $n - Q + 1$ filled spots, the last $Q - 1$ spots have fewer than Q candidates (in particular, when the last spot is being filled, there is only one probe candidate). As a result, we usually observe comparatively higher levels of conflicts in the last filled spots.

We also observed that, in terms of border length, increasing Q above 5K has little positive effect (see Figure 3.5). For instance, on 800×800 chips, increasing Q from 5K to 20K reduced the normalized border length by only 1.27% (from 16.9124 to 16.6980 with $k = 0$), while requiring approximately six times more time. In terms of conflict index, however, increasing Q even above 40K still results in significant improvements for large chips. For instance, on 800×800 chips, increasing Q from 40K to 80K reduced the average conflict index by 3.18% (from 378.3110 to 366.8446 with $k = 0$, data not shown). The fact that increasing Q has more effect in terms of conflict index is probably because, in this measure, there is more room for optimization as the conflicts can be moved to the extremities of the probes (while retaining the same number of border conflicts) and a larger number of neighbors are involved.

Figure 3.6 shows the normalized border length per masking step of layouts produced by Greedy for a 500×500 chip with border length and conflict index minimization. With BLM, the generated layout has most border conflicts concentrated between steps 7 and 58. The last masks of this layout have low levels of border conflicts because

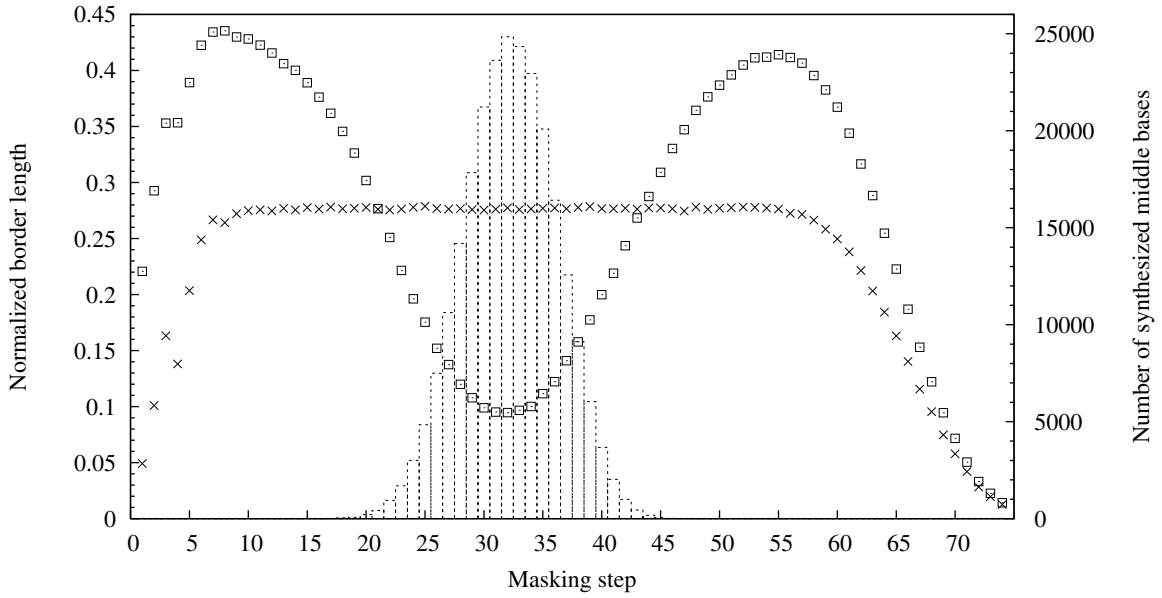


Figure 3.6: Normalized border length (on the left y-axis) per masking step of layouts produced by Greedy for a 500×500 chip with border length (×) and conflict index (⊡) minimization using 0-threading and $Q = 20$K. Chip contained random probe sequences left-most embedded in the standard 74-step Affymetrix deposition sequence. The number of middle bases synthesized at each step is shown in boxes (right y-axis).

the probes are left-most embedded, which leaves most embeddings in an unproductive state during the final synthesis steps. As a result of the lexicographical sorting of probes, the first masks also have relatively few conflicts. A representation of selected photolithographic masks for this layout are shown in Figure 3.7. Layers of masked and unmasked regions that result from sorting probes lexicographically can be seen in masks M_1 to M_8. Masks M_9 to M_{62} are very "noisy" as there seems to be little regularity in their arrangement. After M_{62}, masks start to get "darker" as most probes have been already fully synthesized.

With CIM, Greedy shifts border conflicts away from the steps that add the middle bases (between steps 20 and 45; see Figure 3.6), which effectively reduces the average conflict index. Not surprisingly, this reduction comes at the expense of an increase in total border length — the normalized border length of this particular chip rose from 17.3513 with BLM to 19.8461 with CIM. Figure 3.8 shows a representation of selected masks for this layout. Note that the first masks (M_1 to M_4) still exhibit a "layered" structured, althought the layers are much narrower and the masks noisier than the first masks of Figure 3.7. In the central masks M_{20} to M_{45}, especially between M_{25} and M_{40}, it is possible to see clusters of masked and unmasked spots that cause the reduction in average conflict index.

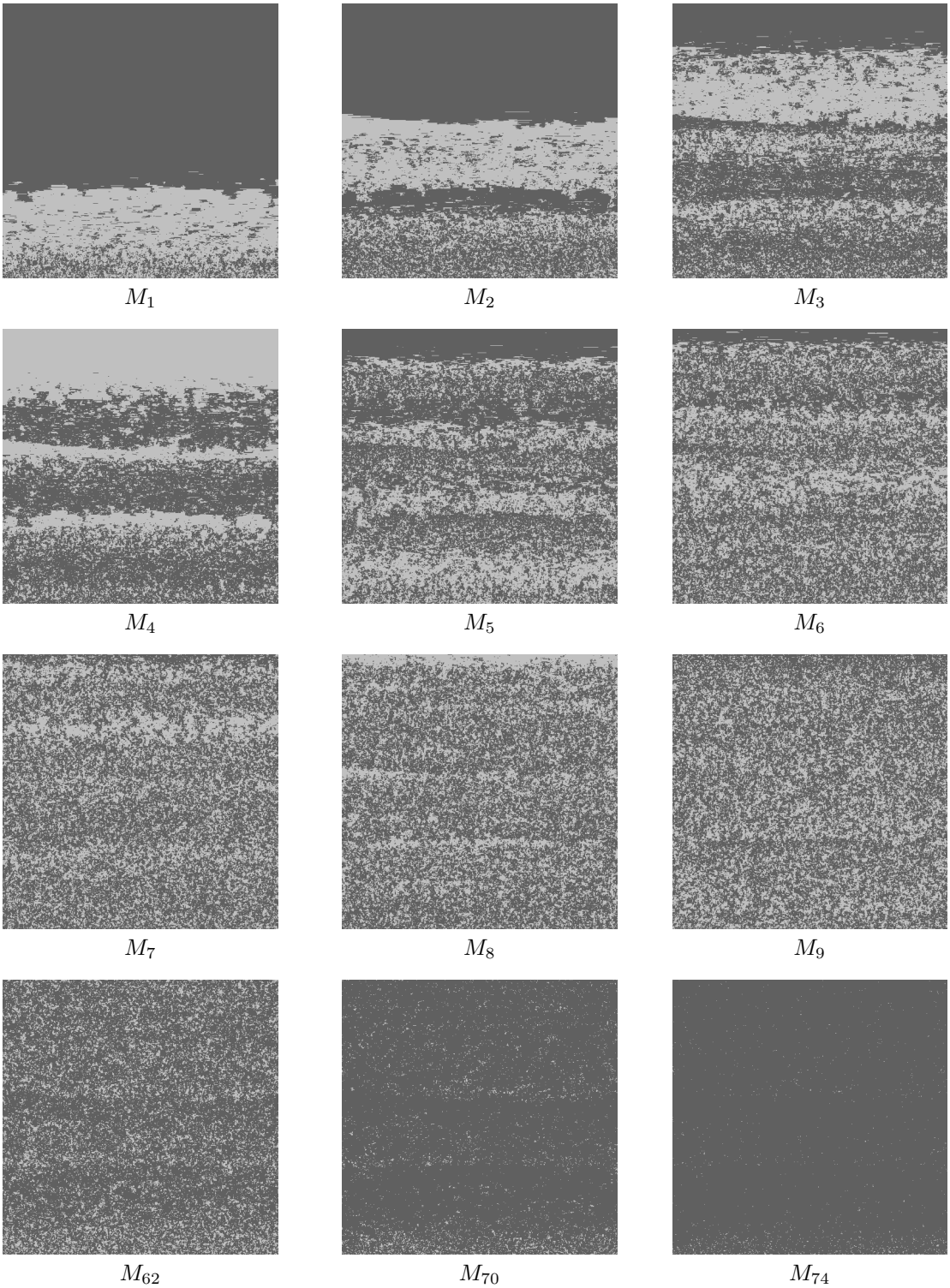


Figure 3.7: Selected masks generated by Greedy with border length minimization for a random 500×500 chip with 25-mer probes left-most embedded in the standard Affymetrix deposition sequence. Unmasked (masked) spots are represented by light (dark) dots.

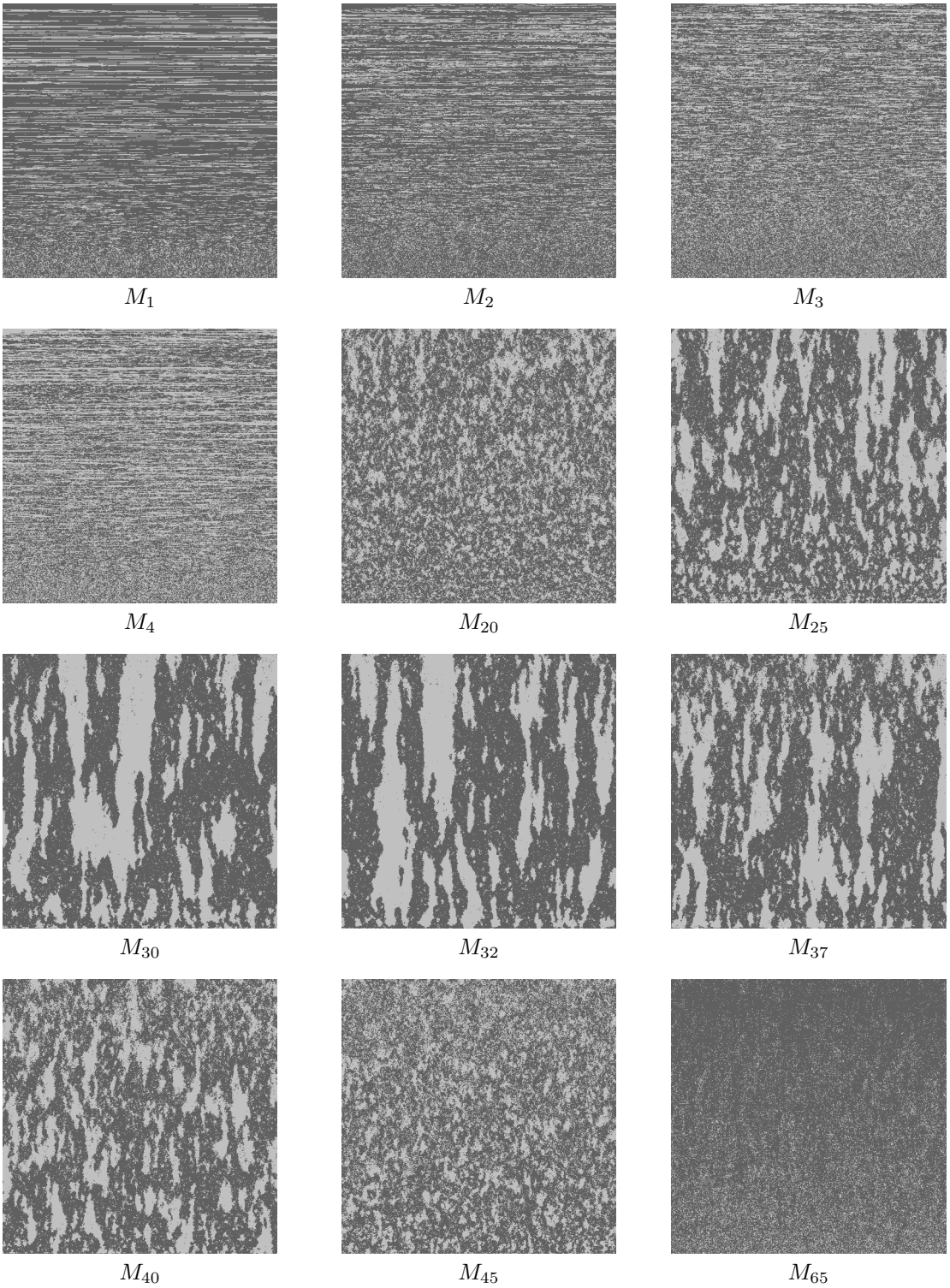


Figure 3.8: Selected masks generated by Greedy with conflict index minimization for a random 500×500 chip with 25-mer probes left-most embedded in the standard Affymetrix deposition sequence. Unmasked (masked) spots are represented by light (dark) dots.

3.7 Summary

In this chapter, we have surveyed placement algorithms for the microarray layout problem, including an optimal placement strategy for uniform arrays based on a two-dimensional Gray code, an approach based on the traveling salesman problem and different threading techniques. For general arrays, we have presented more experimental results with Row-Epitaxial, the best known placement algorithm to date, and studied the impact of the choice of threading for its initial layout.

We have also introduced a new placement algorithm, called Greedy. Greedy achieved similar results in terms of BLM and better results in terms of CIM compared to Row-Epitaxial. For BLM, Row-Epitaxial is faster than Greedy and should still be the method of choice. For CIM, however, the improvements achieved by Greedy over Row-Epitaxial justify the small increase in running time.

Chapter 4

MLP and the Quadratic Assignment Problem

In this chapter, we show that the microarray layout problem (MLP) with general distance-dependent and position-dependent weights is an instance of the *quadratic assignment problem* (QAP), a classical combinatorial optimization problem introduced by Koopmans and Beckmann (1957), which opens up the way for using QAP techniques to design microarray chips.

We then use an existing QAP heuristic algorithm called GRASP to design the layout of small artificial chips, comparing our results with the best known placement algorithm. The chapter ends with a discussion about how this approach can be combined with other existing algorithms to design and improve larger microarrays.

4.1 Quadratic assignment problem

The quadratic assignment problem (QAP) can be stated as follows. Given $n \times n$ real-valued matrices $F = (f_{ij}) \geq 0$ and $D = (d_{kl}) \geq 0$, find a permutation π of $\{1, 2, \ldots n\}$ such that

$$\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i)\pi(j)} \rightarrow \min . \tag{4.1}$$

The attribute *quadratic* stems from the fact that the target function can be written with n^2 binary indicator variables $x_{ik} \in \{0, 1\}$, where $x_{ik} := 1$ if and only if $k = \pi(i)$. The objective (4.1) then becomes

$$\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot \sum_{k=1}^{n} \sum_{l=1}^{n} d_{kl} \cdot x_{ik} \cdot x_{jl} \rightarrow \min,$$

such that $\sum_k x_{ik} = 1$ for all i, $\sum_i x_{ik} = 1$ for all k, and $x_{ik} \in \{0, 1\}$ for all (i, k). The objective function is a quadratic form in x.

The QAP has been used to model a variety of real-life problems. One common example is the facility location problem where n facilities must be assigned to n locations. The facilities could be, for instance, the clinics, doctors or services (X-ray, emergency room, etc.) provided by a hospital and the locations could be the available rooms of the hospital building.

In this scenario, F is called the *flow matrix* as f_{ij} represents the flow of materials or persons from facility i to facility j. Matrix D is called the *distance matrix*, as d_{kl} gives the distance between locations k and l. One unit of flow is assumed to have an associated cost proportional to the distance between the facilities, and the optimal permutation π defines a one-to-one assignment of facilities to locations with minimum cost.

4.2 QAP formulation of the MLP

The MLP can be seen as an instance of the QAP, where we want to find a one-to-one correspondence between spots and probes minimizing a given penalty funtion such as total border length or total conflict index (defined in Chapter 2). To formulate it, we use the facility location example by viewing the probes as locations and the spots as facilities, i.e., the spots are assigned to the probes. The flow matrix F then contains the "closeness" values between spots, while the distance matrix D contains the conflicts between probe embeddings.

We first give the general formulation for conflict index minimization case; the border length minimization case is obtained by using the particular weight functions given in Section 2.4.

In a realistic setting, we may have more spots available than probes to place. Below, we show that this does not cause problems as we can add enough "empty" probes and define their weights appropriately.

Perhaps more severely, we assume that all probes have a single pre-defined embedding in order to force a one-to-one relationship. A more elaborate formulation would consider all possible embeddings of a probe, but then it becomes necessary to ensure that only one embedding of each probe is used. This still leads to a quadratic integer programming problem, albeit with slightly different side conditions.

Our goal is to design a microarray minimizing the sum of conflict indices over all spots $s \in \mathcal{S}$, i.e.,

$$\sum_{s \in \mathcal{S}} \mathcal{C}(s) \to \min .$$

The "flow" f_{ij} between spots i and j depends on their distance on the chip; in accordance with the conflict index model, we set

$$f_{ij} := \mathbb{1}_{\{i,j \text{ neighbors}\}} \cdot \gamma(i,j) \tag{4.2}$$

where "neighbors" means that spots i and j are at most three cells away (horizontally and vertically) from each other. Note that most of the flow values on large arrays are zero. For border length minimization, the case is even simpler: We set $f_{ij} := 1$ if spots i and j are adjacent, and $f_{ij} := 0$ otherwise.

The "distance" d_{kl} between probes k and l depends on the conflicts between their embeddings ε_k and ε_l. To account for possible "empty" probes to fill up surplus spots, we set $d_{kl} := 0$ if k or l or both refer to an empty probe — i.e., empty probes never contribute to the target function since we do not mind if nucleotides are erroneously added to spots assigned to empty probes. For real probes, we set

$$d_{kl} := \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{k,t}=0\}} \cdot \omega(\varepsilon_k, t) \cdot \mathbb{1}_{\{\varepsilon_{l,t}=1\}} \Big). \tag{4.3}$$

Note that d_{kl} is related to the conflict index distance $C(k,l)$ defined in Section 2.3 (Equation 2.10):

$$\begin{aligned}
& d_{kl} + d_{lk} \\
= \; & \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{k,t}=0\}} \cdot \omega(\varepsilon_k, t) \cdot \mathbb{1}_{\{\varepsilon_{l,t}=1\}} \Big) + \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{l,t}=0\}} \cdot \omega(\varepsilon_l, t) \cdot \mathbb{1}_{\{\varepsilon_{k,t}=1\}} \Big) \\
= \; & \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{k,t}=0 \text{ and } \varepsilon_{l,t}=1\}} \cdot \omega(\varepsilon_k, t) \Big) + \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{l,t}=0 \text{ and } \varepsilon_{k,t}=1\}} \cdot \omega(\varepsilon_l, t) \Big) \\
= \; & \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{k,t}=0 \text{ and } \varepsilon_{l,t}=1\}} \cdot \omega(\varepsilon_k, t) + \mathbb{1}_{\{\varepsilon_{l,t}=0 \text{ and } \varepsilon_{k,t}=1\}} \cdot \omega(\varepsilon_l, t) \Big) \\
= \; & C(k,l)
\end{aligned}$$

In the case of border length minimization, where $\theta = 0$ and $c = 1/2$ (see Section 2.4), we obtain that $d_{kl} + d_{lk} = H(k,l) = H(l,k)$, where H_{kl} denotes the Hamming distance between the embeddings ε_k and ε_l (Equation 2.3).

It now follows that for a given assignment π, we have,

$$f_{ij} \cdot d_{\pi(i)\pi(j)} = \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{\pi(i),t}=0\}} \cdot \omega(\varepsilon_{\pi(i)}, t) \cdot \mathbb{1}_{\{\varepsilon_{\pi(j),t}=1\}} \cdot \mathbb{1}_{\{i,j \text{ neighbors}\}} \cdot \gamma(i,j) \Big).$$

The objective function (4.1) then becomes

$$\begin{aligned}
& \sum_i \sum_j f_{ij} \cdot d_{\pi(i)\pi(j)} \\
= & \sum_i \sum_j \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{\pi(i),t}=0\}} \cdot \omega(\varepsilon_{\pi(i)}, t) \cdot \mathbb{1}_{\{\varepsilon_{\pi(j),t}=1\}} \cdot \mathbb{1}_{\{i,j \text{ neighbors}\}} \cdot \gamma(i,j) \Big) \\
= & \sum_i \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{\pi(i),t}=0\}} \cdot \omega(\varepsilon_{\pi(i)}, t) \cdot \sum_j \mathbb{1}_{\{i,j \text{ neighbors}\}} \cdot \mathbb{1}_{\{\varepsilon_{\pi(j),t}=1\}} \cdot \gamma(i,j) \Big) \\
= & \sum_i \sum_{t=1}^{T} \Big(\mathbb{1}_{\{\varepsilon_{\pi(i),t}=0\}} \cdot \omega(\varepsilon_{\pi(i)}, t) \cdot \sum_{\substack{j\text{: neighbor} \\ \text{of } i}} \mathbb{1}_{\{\varepsilon_{\pi(j),t}=1\}} \cdot \gamma(i,j) \Big) \\
= & \sum_i \mathcal{C}(i),
\end{aligned}$$

and indeed equals the total conflict index with our definitions of $F = (f_{ij})$ and $D = (d_{kl})$.

Remark. Note that it is technically possible to switch the definitions of F and D, i.e., to assign probes to spots instead of spots to probes as we do now, without modifying the mathematical problem formulation. However, this would lead to high distance values for neighboring spots and many zero distance values for independent spots, a somewhat counterintuitive model. Also, some QAP heuristics initially find pairs of objects with large flow values and place them close to each other. Therefore, the way of modeling F and D may be significant.

4.3 QAP heuristics

We have shown how the microarray placement problem can be modeled as a quadratic assignment problem. However, the QAP is known to be NP-hard and particularly hard to solve in practice. Instances of size larger than $n = 20$ are generally considered to be impossible to solve to optimality. Fortunately, several heuristics exist, including approaches based on tabu search, simulated annealing and genetic algorithms (for a survey, see Çela, 1997; Loiola et al., 2007). Our formulation is thus of interest because we can now use existing QAP heuristics to design the layout of microarrays minimizing either the sum of border lengths or conflict indices.

As an example, we briefly describe a general QAP heuristic known as GRASP (Li et al., 1994), which was first used for solving the QAP by Feo and Resende (1995),

and an improved version called GRASP with path-relinking (Oliveira et al., 2004), that we used to design small microarray chips with our formulation.

4.3.1 GRASP with Path-relinking

GRASP (Greedy Randomized Adaptive Search Procedure) is comprised of two phases: a construction phase where a random feasible solution is built, and a local search phase where a local optimum in the neighborhood of that solution is sought. In the following description we use the terms of the facility location problem: f_{ij} is the flow between facilities i and j, d_{kl} is the distance between locations k and l.

The construction phase starts by sorting the (n^2-n) elements of the distance matrix in increasing order and keeping the smallest $E := \lfloor \beta(n^2-n) \rfloor$ elements, where $0 < \beta < 1$ is a restriction parameter given as input.

$$d_{k_1 l_1} \leq d_{k_2 l_2} \leq \cdots \leq d_{k_E l_E}.$$

Similarly, the $(n^2 - n)$ elements of the flow matrix are sorted, this time in decreasing order, and the largest E elements are kept:

$$f_{i_1 j_1} \geq f_{i_2 j_2} \geq \cdots \geq f_{i_E j_E}.$$

Then, the costs of assigning pairs of facilities to pairs of locations are computed. The cost of initially assigning facility i_q to location k_q and facility j_q to location l_q for some $q \in \{1, \ldots, E\}$ is $d_{k_q l_q} f_{i_q j_q}$. GRASP sorts the vector

$$(d_{k_1 l_1} f_{i_1 j_1},\ d_{k_2 l_2} f_{i_2 j_2},\ \ldots,\ d_{k_E l_E} f_{i_E j_E}),$$

keeping the $\lfloor \alpha E \rfloor$ smallest elements, where $0 < \alpha < 1$ is another restriction parameter. A simultaneous assignment of a pair of facilities to a pair of locations is selected at random among those with the $\lfloor \alpha E \rfloor$ smallest costs, and a feasible solution is then built by making a series of greedy assignments.

In the local search phase, GRASP searches for a local optimum in the neighborhood of the constructed solution. Several search strategies and definitions of neighborhood can be used. One possible approach is to check every possible swap of assignments and make only those which improve the current solution until no further improvements can be made.

The construction and local search phases are repeated for a given number of times, and the best solution found is returned.

Path-relinking. GRASP takes no advantage of the knowledge gained in previous iterations to build or improve an obtained solution, i.e., each new solution is built from scratch.

GRASP with path-relinking is an extension of the basic GRASP algorithm that uses an "elite set" to store the best solutions found. It incorporates a third phase that chooses, at random, one elite solution that is used to improve the solution produced at the end of the local search phase.

Solutions p and q are combined as follows. For every location $k = 1, \ldots, n$, the path-relinking algorithm attempts to exchange facility p_k assigned to location k in solution p with facility q_k assigned to location k in the elite solution. In order to keep the solution p feasible, it exchanges p_k with p_l, where $p_l = q_k$. This exchange is performed only if it results in a better solution. The result of the path-relinking phase is a solution r that is at least as good as the better of p and q.

4.4 Results

We present experimental results of using GRASP with path-relinking (GRASP-PR) for designing the layout of small artificial chips, and compare them with the layouts produced by the Greedy placement algorithm (described in Section 3.6), with the number Q of candidates per spot set to a sufficiently large value so that all available probes are considered for each spot.

We used a C implementation of GRASP-PR provided by Oliveira et al. (2003) with default parameters (32 iterations, $\alpha = 0.1$, $\beta = 0.5$, and elite set of size 10). The main routine takes three arguments: the dimension n of the problem (in our case, the number of spots or probes) and matrices F and D. The matrices were generated using the formulations presented in Section 4.2.

The data set consists of chips with probes of length 25 uniformly generated and asynchronously embedded in a deposition sequence of length 74. The running times and the border lengths of the resulting layouts are shown in Table 4.1 (all results are averages over a set of ten chips).

Our results show that GRASP-PR produces layouts with lower border lengths than Greedy on the smaller chips. On 6×6 chips, GRASP-PR outperforms Greedy by 2.14 percentage points on average ($15.94\% - 13.80\%$), when compared to the initial random layout. On 9×9 chips, however, this difference drops to 0.16 percentage points, while Greedy generates better layouts on 11×11 or larger chips. In terms of running time, Greedy is faster and shows little variation as the number of probes grows. In contrast, the time required to compute a layout with GRASP-PR increases at a fast rate.

Table 4.1: Total border length of random chips compared with the layouts produced by Greedy and GRASP with path-relinking. Reductions in border length are reported in percentages compared to the random layout.

	Random	Greedy placement			GRASP with path-relinking		
Chip dimension	Border length	Border length	Reduction (%)	Time (sec.)	Border length	Reduction (%)	Time (sec.)
6×6	1 989.20	1 714.60	13.80	0.01	1 672.20	15.94	2.73
7×7	2 783.20	2 354.60	15.40	0.02	2 332.60	16.19	6.43
8×8	3 721.20	3 123.80	16.05	0.03	3 099.13	16.72	12.49
9×9	4 762.00	3 974.80	16.53	0.05	3 967.20	16.69	25.96
10×10	5 985.20	4 895.60	18.20	0.06	4 911.40	17.94	47.57
11×11	7 288.40	5 954.40	18.30	0.10	5 990.73	17.80	87.48
12×12	8 714.00	7 086.20	18.68	0.11	7 159.80	17.84	152.42

Table 4.2: Average conflict indices of random chips compared with the layouts produced by Greedy and GRASP with path-relinking.

	Random	Greedy placement			GRASP with path-relinking		
Chip dimension	Avg. C. Index	Avg. C. Index	Reduction (%)	Time (sec.)	Avg. C. Index	Reduction (%)	Time (sec.)
6×6	524.28	495.15	5.56	0.05	467.08	10.91	3.68
7×7	558.25	521.90	6.51	0.07	489.32	12.35	8.84
8×8	590.51	551.84	6.55	0.09	515.69	12.67	19.48
9×9	613.25	568.62	7.28	0.11	533.79	12.96	38.83
10×10	628.50	576.49	8.28	0.11	539.69	14.13	73.09
11×11	642.72	588.91	8.37	0.12	551.41	14.21	145.67
12×12	656.86	598.21	8.93	0.12	561.21	14.56	249.19

Table 4.2 shows better results in terms of conflict indices. GRASP-PR consistently produces better layouts on all chip dimensions, achieving up to 6.38% less conflicts on 10×10 chips, for example, when compared to Greedy. In terms of running times, GRASP-PR is even slower than in the border length case. The reason is not clear, but it could be because the distance matrix contains fewer zero entries with the conflict index formulation.

The gains in terms of conflict index of both Greedy and GRASP-PR are clearly less than the gains in terms of border length (when compared to the initial random layout). This may be because the probe embeddings are fixed and the reduction of conflicts is restricted to the relocation of the probes, which only accounts for one part of the conflict index model.

4.5 Discussion

The QAP is notoriously hard to solve, and currently known exact methods start to take prohibitively long already for slightly more than 20 objects, i.e., we could barely solve

the problem exactly for 5×5 arrays. Fortunately, the literature on QAP heuristics is rich, as many problems in operations research can be modeled as QAPs. Here we used one such heuristic to identify the potential of the MLP-QAP-relation.

As our results show, however, even heuristic algorithms are too slow to deal with chips of dimensions larger than 12×12, and although we could design a 20×20 chip with a QAP heuristic within a day, we have to keep in mind that this would still be a very small part of bigger problem as real microarray dimensions range from 200×200 up to 1164×1164.

For this reason, we restricted our experiments to such small chips and QAP heuristics that could handle the problems within a few minutes. Up to now, finding exact solutions even to these small microarrays seems to be an incredible hard task. We mention here experiments conducted by Dr. Peter Hahn, who used two branch-and-bound algorithms to solve some problem instances from Table 4.1. With RTL-2 (Adams et al., To appear), it was possible to find two solutions with total border length of 1 652 for a selected 6×6 chip, being only 1.43% better than the solution found with GRASP-PR (1 676), although it took RTL-2 about 6.5 hours, in contrast with the less than 3 seconds needed by GRASP-PR. A lower bound calculation for the same problem resulted in 1 624, so the RTL-2 solution is only 1.69% higher, while the gap to the GRASP-PR solution is about 3.10%.

For another selected problem of dimension 7×7, Dr. Hahn found one solution with border length 2 290 using RTL-1 (Hahn et al., 1998), being about 1.72% better than the solution found by GRASP-PR (2 330), although it took RTL-1 some 29 hours, in contrast with the less than 7 seconds needed for the GRASP-PR run. The results obtained with exact QAP solvers give an idea of how hard the quadratic assignment problem actually is, and show that the results with GRASP-PR are a good compromise when time is limited.

Improved results for several selected problem instances from Tables 4.1 and 4.2 were also reported by Chris MacPhee using GATS, a hybrid genetic / tabu search algorithm, although these results were obtained on a number of large memory SMP machines, each having 144 processors and 576 GB of global memory. The latest results for these selected problems are available online.[1]

4.5.1 Alternatives

It is clear that, because of the large number of probes on industrial microarrays, it is not feasible to use GRASP-PR (or any other currently available QAP method) to design an entire microarray chip. However, we showed that it is certainly possible to use it on small sub-regions of a chip, which opens up the way for two alternatives.

[1] `http://gi.cebitec.uni-bielefeld.de/comet/chiplayout/qap`

First, the QAP approach could be used combined with a partitioning algorithm such as those discussed in Chapter 6 to the design the smaller regions that result from the partitioning. This, however, does not seem promising because, as we will see later, a partitioning is a compromise in solution quality, and level of partitioning required to achieve the dimensions supported by the QAP approach is too high.

It is interesting to extrapolate the times shown on Table 4.1 to predict the total time that would be required to design the layout of commercial microarrays, if we were to combine GRASP-PR with a partitioning algorithm. If the partitioning produced 6×6 regions, 37 636 sub-regions would be created from the 1164×1164 Affymetrix Human Genome U133 Plus 2.0 GeneChip array, one of the largest Affymetrix chips. Since each sub-region takes around 3 seconds to compute with GRASP-PR, the total time required for designing such a chip would be a little over 31 hours (ignoring the time for the partitioning itself).

If the partitioning produced 12×12 regions, 9 409 sub-regions would be created and, at 2.4 minutes each, the total time would be more than 16 days. This is probably prohibitive, although it is certainly possible to reduce the time of each GRASP-PR execution by running it on faster machines or run them in parallel.

A better alternative is to use the QAP approach to improve an existing layout, iteratively, by relocating probes inside a defined region of the chip, in a sliding-window fashion. Each iteration of this method would produce an instance of a QAP whose size equals the number of spots inside the window.The QAP heuristics could then be used to check whether a different arrangement of the probes inside the window can reduce the conflicts. For this approach to work, however, we also need to take into account the conflicts due to the spots around the window. Otherwise, a new layout with less internal conflicts could be achieved at the expense of increasing conflicts on the borders of the window.

A simple way of preventing this problem is to solve a larger QAP instance consisting of the spots inside the window as well as those in a layer (of three spots) around it. The spots outside the window obviously must remain unchanged, and that can be done by fixing the corresponding elements of the permutation π. Note that there is no need to compute f_{ij} if spots i and j are both outside the window, nor d_{kl} if probes k and l are assigned to spots outside the window.

Chapter 5

Re-embedding Algorithms

After the placement phase, it is no longer possible to reduce conflicts if probes are synchronously embedded. With asynchronous embeddings, however, layouts can usually be further improved by *re-embedding* the probes without changing their locations on the chip, in what is sometimes called a *post-placement optimization* phase.

All re-embedding algorithms discussed in this chapter are based on the Optimum Single Probe Embedding (OSPE) introduced by Kahng et al. (2002). OSPE is a dynamic programming algorithm for computing an optimum embedding of a single probe with respect to its neighbors, whose embeddings are considered as fixed. The algorithm was originally developed for border length minimization (BLM) but here we present a more general form designed for conflict index minimization (CIM) that first appeared in (de Carvalho Jr. and Rahmann, 2006a).

5.1 Optimum Single Probe Embedding

The Optimum Single Probe Embedding algorithm, OSPE for short, can be seen as a special case of a global alignment between a probe sequence p of length ℓ and the deposition sequence N of length T, disallowing mismatches and gaps in N. We assume that p is placed at spot s, and that we know the embeddings of all probes placed at spots near s (spots that are at most three cells away from s, horizontally and vertically, in accordance with the conflict index model).

The optimal embedding of p into N is built by determining the minimum cost of embedding a prefix of p into a prefix of N: We use an $(\ell + 1) \times (T + 1)$ matrix D, where $D[i, t]$ is defined as the minimum cost of an embedding of $p[1..i]$ into $N[1..t]$ for $0 \leq i \leq \ell$, $0 \leq t \leq T$. The cost is the sum of conflicts induced by the embedding of $p[1..i]$ on its neighbors (when s is unmasked and a neighbor is masked), plus the conflicts suffered by $p[1..i]$ because of the embeddings of its neighbors (when s is masked and a neighbor is unmasked).

We can compute the value for $D[i,t]$ by looking at two previous entries in the matrix: $D[i,t-1]$ and $D[i-1,t-1]$. The reason is that $D[i,t]$ is the minimum cost of embedding $p[1..i]$ up to the t-th synthesis step of N, which can only be obtained from the previous synthesis step $(t-1)$ by either masking or unmasking spot s at step t.

If s is productive (unmasked) at step t, base N_t is appended to $p[1..i-1]$; this is only possible if $p[i] = N[t]$. In this case a cost U_t is added for the risk of damaging probes at neighboring spots s'. We know that $p[1..i-1]$ can be embedded in $N[1..t-1]$ with optimal cost $D[i-1,t-1]$. Hence, the minimum cost at step t, if s is productive, is $D[i-1,t-1]+U_t$. According to the conflict index model,

$$U_t := \sum_{\substack{s':\ \text{neighbor}\\ \text{of } s}} \mathbb{1}_{\{\varepsilon_{k(s'),t}=0\}} \cdot \omega(\varepsilon_{k(s')}, t) \cdot \gamma(s', s). \tag{5.1}$$

If s is unproductive (masked) at step t, no base is appended to $p[1..i-1]$, but a cost $M_{i,t}$ must be added for the risk of damaging p (by light directed at neighboring spots s'). Since $D[i,t-1]$ is the minimum cost of embedding $p[1..i]$ in $N[1..t-1]$, the minimum cost up to step t, if s is unmasked, is $D[i,t-1]+M_{i,t}$.

Note that $M_{i,t}$ depends on the number of bases probe p already contains (that is, on i): Each unmasked neighbor s' generates a conflict on p with cost

$$\gamma(s,s') \cdot c \cdot \exp(\theta \cdot (1 + \min\{i, \ell - i\})),$$

in accordance with (2.6)–(2.8). Thus,

$$M_{i,t} := c \cdot \exp(\theta \cdot (1 + \min\{i, \ell - i\})) \cdot \sum_{\substack{s':\ \text{neighbor}\\ \text{of } s}} \mathbb{1}_{\{\varepsilon_{k(s'),t}=1\}} \cdot \gamma(s, s'). \tag{5.2}$$

Finally, $D[i,t]$ is computed as the minimum cost of the possible actions,

$$D[i,t] := \begin{cases} \min\{\, D[i,t-1] + M_{i,t},\ D[i-1,t-1] + U_t \,\} & \text{if } p[i] = N[t], \\ D[i,t-1] + M_{i,t} & \text{if } p[i] \neq N[t]. \end{cases}$$

The first column of D is initialized as follows: $D[0,0] = 0$ and $D[i,0] = \infty$ for $0 < i \leq \ell$, since no probe of length $\ell > 0$ can be embedded into an empty deposition sequence. The first row is initialized by setting $D[0,t] = D[0,t-1] + M_{0,t}$ for $0 < t \leq T$.

If we assume that costs U_t and $M_{i,t}$ can be computed in constant time, the time complexity of the OSPE algorithm is $O(\ell T)$ since there are $O(\ell T)$ entries in D to compute. The algorithm can be rather time-consuming in the general form presented here, since we have to look at the embeddings of up to 48 neighbors around s. Naturally, it runs

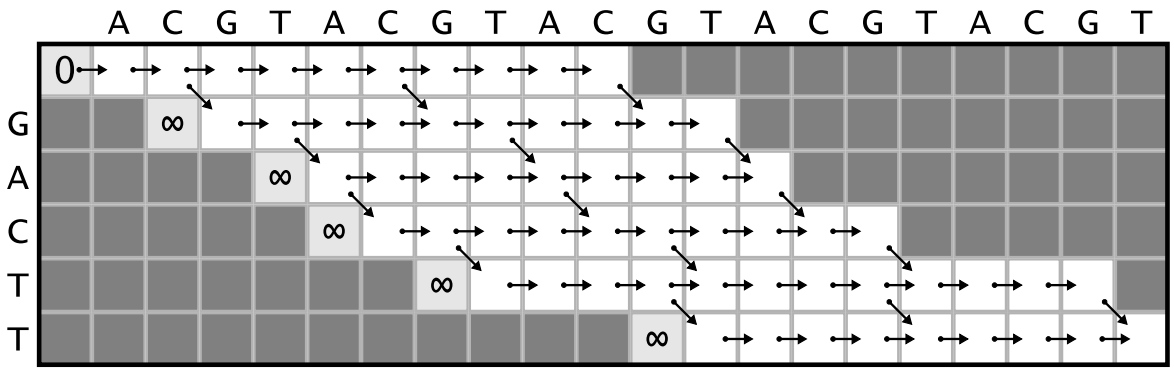


Figure 5.1: OSPE's dynamic programming matrix for computing an optimal embedding of a probe p = GACTT in a deposition sequence $N = (\texttt{ACGT})^5$. Dark shaded cells are not computed. Arrows show all paths in the matrix leading to a valid embedding of p in N.

much faster for border length minimization, since there are only four neighbors to look at, and there are neither position-dependent (ω) nor distance-dependent (γ) weights to compute. In practice, a simple optimization can significantly reduce running time: in each row, only the columns between the left-most and right-most embeddings of p in N need to be computed (see Figure 5.1).

Once D is computed, the minimum cost is $D[\ell, T]$, and an optimal embedding of p in N can be constructed by tracing a path from $D[\ell, T]$ back to $D[0, 0]$, similarly to the procedure used to build an optimal global alignment. This takes $O(T)$ time.

The OSPE algorithm is the basic operation of several post-placement optimization algorithms: Chessboard, Greedy and Batched Greedy, and Sequential, as well as our new Priority re-embedding algorithm. The main difference between these algorithms lies in the order in which the probes are re-embedded.

Since OSPE never increases the amount of conflicts in the region around the re-embedded probe, optimization algorithms can execute several re-embedding operations without risk of worsening the current layout. Moreover, each probe may be re-embedded several times since new improvements may be possible once its neighbors have been changed. In fact, all algorithms presented here work in repeating cycles of optimization until no more improvements are possible (when a local optimal solution is found), until improvements drop below a given threshold W, or until a given number of cycles (or *passes*) have been executed.

5.2 Chessboard

The Chessboard re-embedding algorithm (Kahng et al., 2002) was initially designed for border length minimization and it takes advantage of the fact that, in this model, a

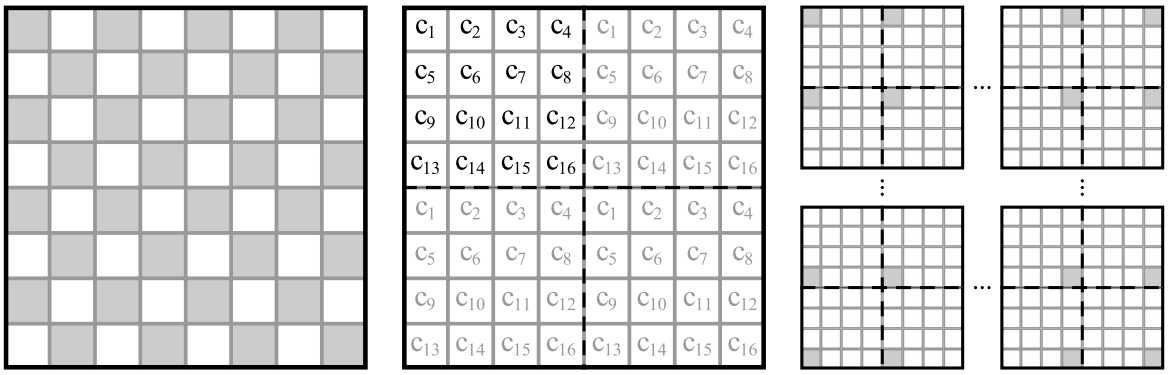


Figure 5.2: a) The chessboard-like bi-coloring of a chip used by the Chessboard re-embedding algorithm for border length minimization; b) a possible coloring of the chip for conflict index minimization using 16 colors (c_1 to c_{16}), resulting in sets of independent spots; c) four of the 16 sets of independent spots (shaded) that can be re-embedded in the same iteration.

chip can be bi-colored like a chessboard, in such a way that the embeddings of probes located on white spots are independent of those placed on black spots (Figure 5.2a).

Chessboard uses this coloring to alternate the optimal re-embedding of probes located on black and white spots with respect to their neighbors: Each pass of Chessboard consists of re-embedding all probes of black spots and then all probe of white spots.

The chessboard coloring guarantees that probes re-embedded in the same step are independent with respect to the border length model, i.e. they can be re-embedded without affecting the border conflicts of other spots with the same color. For conflict index minimization, the same principle can be applied by using $4 \times 4 = 16$ colors instead of 2 as illustrated in Figure 5.2 (to the best of our knowledge this has not yet been implemented).

5.3 Greedy and Batched Greedy

As its name implies, the Greedy re-embedding algorithm (Kahng et al., 2002) utilizes a greedy strategy for choosing the order in which probes are re-embedded. At each iteration, Greedy examines every spot of the chip and computes the maximum reduction of border conflicts achievable by optimally re-embedding its probe. It then selects a spot with the highest gain (reduction of conflicts) and re-embeds its probe optimally, updating the gains of adjacent spots.

A faster version of this algorithm, called Batched Greedy (Kahng et al., 2002), pre-selects several independent spots for re-embedding and thus sacrifices its greedy nature in favour of running time by postponing the update of gains.

Like Chessboard, Greedy and Batched Greedy were initially developed for border length minmization, and they can also be extended for conflict index minimization. The main difference is that, once a probe is re-embedded, more neighbors need to be updated. For Batched Greedy, the selection of independent spots need to take into account the minimum distance of four cells (horizonally and vertically) between spots, in accordance with the conflict index model (Section 2.3). Hence fewer spots may be re-embedded in the same iteration.

5.4 Sequential re-embedding

The Sequential algorithm (Kahng et al., 2003b) employs a much simpler and, surprisingly, more efficient strategy. The algorithm just proceeds spot by spot, from top to bottom, left to right, re-embedding each probe optimally in regard to its neighbors. Once the end of the array is reached, Sequential restarts at the top left corner of the array for the next iteration.

The algorithm is not only simple but also fast since there is no need to compute achievable gains for each spot. Nonetheless, Sequential achieved the greatest reduction of border conflicts in the experiments of Kahng et al. (2003b). The authors argue that the main shortcoming of Chessboard and Greedy is that they always re-embed an independent set of spots at a time, and dropping this requirement should allow faster propagation of the effects of new embeddings and hence convergence to a better local optimum.

Tables 5.1 and 5.2 show the results of using Sequential to re-embed the probes of chips produced by the Greedy placement algorithm (Section 3.6). The chips initially contained random probes of length 25, uniformly generated, and left-most embedded in the standard Affymetrix deposition sequence. The threshold W was set to 0.2% (Sequential stopped as soon as the total reduction of conflicts in one pass droped below 0.2%). In all cases, the threshold was reached after two passes.

The reduction of conflicts achieved by Sequential were small (at most 0.579% with border length and 0.829% for CIM), which shows that there is little room for improvements once the placement is fixed. In fact, the more time is spent during placement (greater Q), the less reduction of conflicts is achieved by re-embedding. For instance, on a 300×300 chip, the reduction in average conflict index dropped by 0.12 percentage point (from 0.829% to 0.709%) when the number of candidades per spot considered by Greedy during placement was increased from 5K to 20K.

Table 5.1: Normalized border length (NBL) before and after an optimization phase with the Sequential re-embedding algorithm. Placement was produced by the Greedy placement algorithm (Section 3.6) with border length minimization, 0-threading, and number Q of candidates per spot set to 5K and 20K. The average number of passes executed by Sequential before the threshold $W = 0.2\%$ was reached is shown. The reduction of conflicts is also shown in percentage. Running times are reported in seconds and all results are averages over a set of five chips. The time spent by Sequential is also shown as a percentage of the total time (placement plus re-embedding).

	Greedy placement			Sequential re-embedding				
Dim.	Q	NBL	Time	NBL	Reduct.	Passes	Time	%Total time
300×300	5K	18.3182	98.5	18.2121	0.579%	2.0	4.8	4.617%
	20K	18.0576	577.9	17.9726	0.471%	2.0	4.8	0.830%
500×500	5K	17.5830	345.7	17.4851	0.557%	2.0	12.7	3.538%
	20K	17.3554	1 999.8	17.2779	0.446%	2.0	12.6	0.625%
800×800	5K	16.9124	916.8	16.8201	0.546%	2.0	32.6	3.437%
	20K	16.6980	5 749.7	16.6258	0.432%	2.0	32.4	0.560%

Table 5.2: Average conflict index (ACI) before and after an optimization phase with the Sequential re-embedding algorithm with $W = 0.2\%$. Placement was produced by the Greedy placement algorithm with conflict index minimization, 0-threading, and Q set to 5K and 20K.

	Greedy placement			Sequential re-embedding				
Dim.	Q	ACI	Time	ACI	Reduct.	Passes	Time	%Total time
300×300	5K	440.5166	322.4	436.8630	0.829%	2.0	188.9	36.944%
	20K	415.5003	1 818.6	412.5536	0.709%	2.0	189.9	9.457%
500×500	5K	432.3023	952.5	428.7410	0.824%	2.0	527.3	35.632%
	20K	401.4609	4 027.2	398.6096	0.710%	2.0	528.3	11.597%
800×800	5K	426.0757	2 512.1	422.6277	0.809%	2.0	1 357.9	35.087%
	20K	392.1786	11 182.8	389.3929	0.710%	2.0	1 352.5	10.790%

Although the reductions of conflicts were relatively small, Sequential required approximately half a minute to re-embedded (two times) all probes of a 800×800 chip in the BLM case, which represented about 3.44% of the aggregate time (placement and re-embedding) when $Q = 5$K and only 0.56% when $Q = 20$K.

In some cases, Sequential even provided comparable reduction of border conflicts, in less time, than increasing Q for Greedy. For instance, on a 800×800 chip, Greedy placement with $Q = 20$K and two passes of Sequential re-embedding produced, in approximately half of the time, a layout with only 0.14% more border conflicts than Greedy with $Q = 40$K and no re-embedding (16.6258 NBL in 96.4 minutes versus 16.6026 in 189.0 minutes, respectively; data not shown). In other words, running Sequential is sometimes more efficient than spending more time during placement.

Figure 5.3 shows the normalized border length per masking step of a selected 500×500 chip before and after a re-embedding phase with Sequential for BLM. It is clear that

the reduction of conflicts is achieved mainly between steps 45 and 65, at the expense of a small increase in conflicts in the final synthesis steps. This is a result of fixing the placement with left-most embedded probes, which leaves no room for improvements in the first masks.

In terms of CIM, the reductions were slightly higher but Sequential was over 40 times slower than in the BLM case, taking up to 36.9% of the aggregate time. This, coupled with the fact that Greedy gives significant reductions of conflicts with increasing Q even beyond 40K, makes it difficult to justify the time spent with re-embedding, unless when Q is approaching its limit (number of probes on the chip) and one is looking for the best layout possible.

Figure 5.4 shows the normalized border length per masking step of the same 500×500 chip of Figure 5.3 before and after a re-embedding phase with Sequential for CIM (placement was produced by Greedy also for CIM). Again, reduction of conflicts is restricted to the second half of synthesis steps because of the left-most embeddings, although with relatively better improvements when compared with the border length case.

Our results gives further indication that Sequential has approximately linear time complexity (if we consider that each OSPE operation can be done in constant time). Sequential performed around 19 400 re-embeddings per second in the BLM case and around 475 re-embeddings per second in the CIM case, on average.

5.5 Priority re-embedding

In this section we describe a new re-embedding algorithm, called Priority re-embedding (PR), which uses a priority queue to control the order in which probes are re-embeded.

The algorithm starts by scanning the chip for probes which have a unique embedding in the deposition sequence. These are called *pivots* and they are used as starting locations from where the re-embeddings propagate to other spots of the chip: Once a pivot is found, all of its four adjacent spots on the chip are added to the priority queue. We assume that the chip has at least one pivot, otherwise the deposition sequence could be shortened. If this is not the case, however, we can also use probes with the minimum number of embeddings among all probes as pivots.

If the probes are initially left-most embedded, every embedding with at least one productive step in the last synthesis cycle corresponds to a probe with a unique embedding. If probes are not left-most embedded, we can compute the number of embeddings $E(p, N)$ of a probe p in the deposition sequence N in $O(\ell \cdot T)$ time with dynamic programming, where ℓ is the length of the probe and T is the length of N. In practice, it is possible to compute $E(p, N)$ for a million probes in a few seconds.

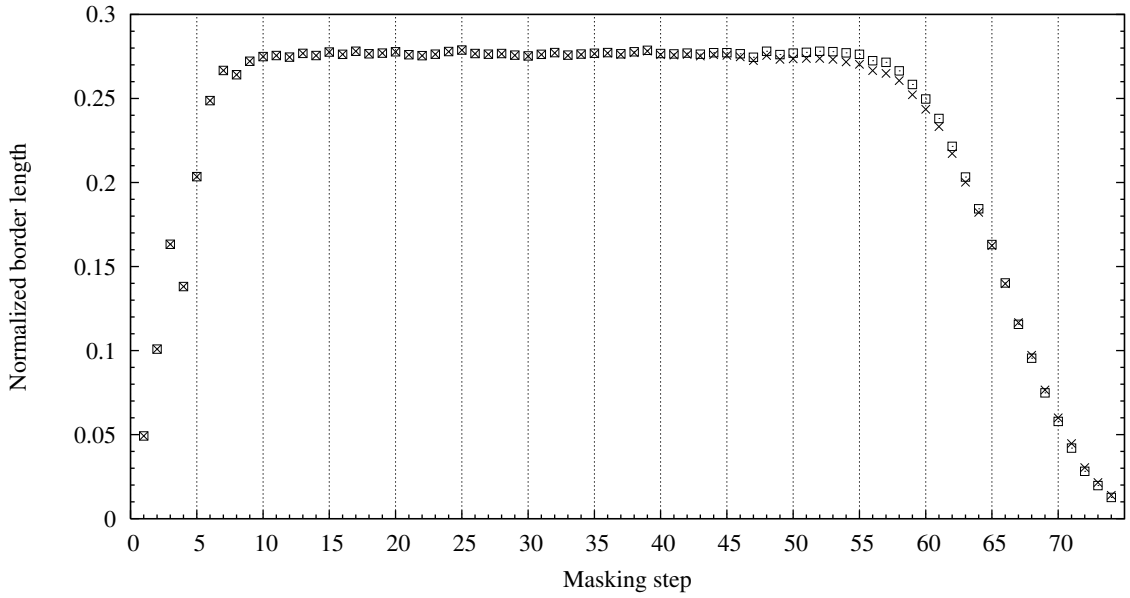


Figure 5.3: Normalized border length per masking step of a 500 × 500 chip before (⊡) and after (×) a re-embedding phase with Sequential for border length minimization. Layout was produced by the Greedy placement algorithm for border length minimization with 0-threading and $Q = 20\text{K}$.

The priority queue is used to retrieve the next spot s whose probe p should be re-embedded, according to the defined priority. Once a probe p is retrieved, it is optimally re-embedded in regard to its neighbors, and all four spots adjacent to s are added to the queue (if they have not been added previously).

We have implemented two different priorities: one based on the number of embeddings of each probe, and one based on the number re-embedded neighbors.

Priority I: Re-embed probes with fewer embeddings first.

The argument behind this priority is based on the observation that probes with more possible embeddings have a greater degree of freedom and can more easily "adapt" to their neighbors. Probes with a restricted number of embeddings, on the other hand, have fewer choices and should be re-embedded first.

In this priority, we examine each spot s with a probe $p_{k(s)}$ and compute $E(p_{k(s)}, N)$, the number of embeddings of $p_{k(s)}$ in N. A weight $w(s) := E(p_{k(s)}, N)$ is assigned for each spot s in the queue, and the spot with the highest weight in each iteration is retrieved.

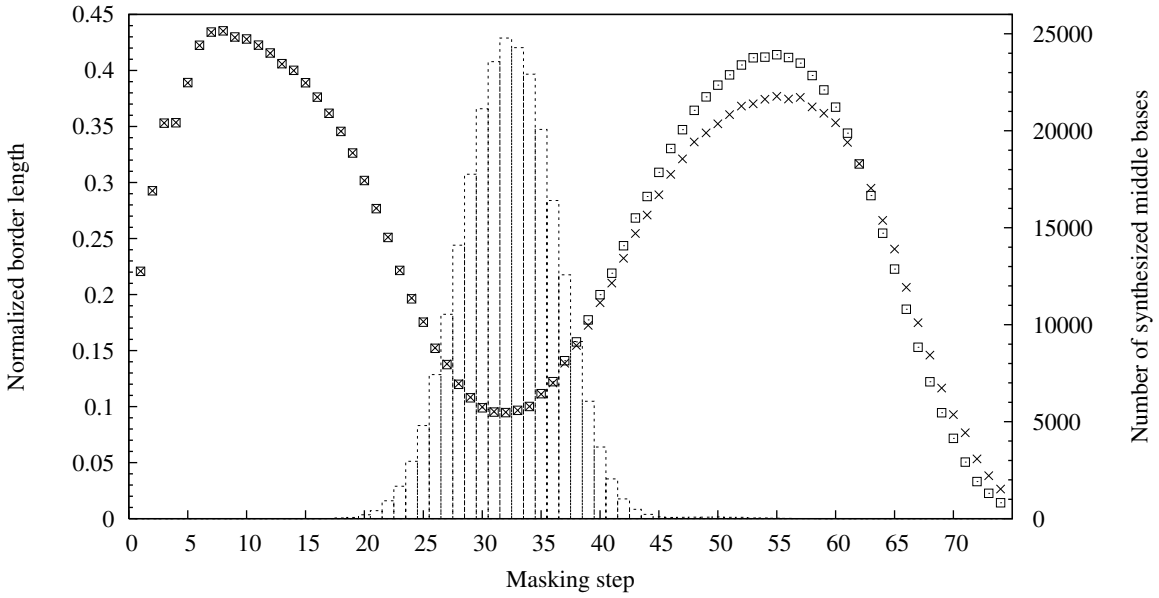


Figure 5.4: Normalized border length per masking step of a 500×500 chip before (⊡) and after (×) a re-embedding phase with Sequential for conflict index minimization. Layout was produced by the Greedy placement algorithm for conflict index minimization with 0-threading and $Q = 20\text{K}$. The number of middle bases synthesized at each step is shown in boxes (right y-axis)

Priority II: Re-embed probes with greater number of re-embedded neighbors first.

This priority tries to mimic the *seeded crystal growth* used by the Epitaxial placement algorithm (Section 3.3), giving preference to probes with a greater number of re-embedded neighbors. The argument behind this priority is that probes should not be re-embedded until a sufficient number of its neighbors have found their final embeddings.

In this priority, we also assign a weight $w(s)$ for each spot s in the queue, and the spot with the highest weight is retrieved. In case of border length minimization, $w(s)$ is set to the the number of immediate neighbors of s that have already been re-embedded in the current iteration.

In case of conflict index minimization, the algorithm looks at all 48 neighbors in the 7×7 region centered around s, and assigns a weight taking into account the distance-dependent function γ (Equation 2.5):

$$w(s) := \sum_{\substack{s' \text{: neighbor} \\ \text{of } s}} \mathbb{1}_{\{s' \text{ has been re-embedded}\}} \cdot \gamma(s, s'),$$

where s' ranges over all neighboring spots that are at most three cells away (hor-

izontally and vertically) from s, in accordance with the conflict index model (Section 2.3).

With Priority II, once a probe is re-embedded, it is necessary to update the weights of its neighbors that have been previously added to the queue (up to 4 with border length minimization, and 48 with conflict index minimization).

5.5.1 Results

Tables 5.3 and 5.4 show the results of using Priority re-embedding on the same set of arrays used for Sequential (Tables 5.1 and 5.2). In terms of BLM, both priorites resulted in negligible improvements when compared to Sequential (with Priority I giving the best results). The greatest difference was only 0.0032% (from 18.2121 with Sequential to 18.2115 with Priority I on 300×300 chips and Greedy placement with $Q = 5$K). Moreover, Priority I was between 8.8% and 12.7% slower than Sequential, whereas Priority II was between 2 to 5 times slower than Sequential.

Priority II is slower than Priority I because after it re-embeds a spot s, it needs to update the weights of all neighbors of s that have been previously added to the queue. With Priority I, the nuber of embeddings of each probe does not change, so they are computed only once, before the first iteration.

In terms of CIM, Priority I produced the worse layouts, whereas Priority II once again achieved negligible improvements when compared to Sequential — at most 0.0029% (from 412.5536 to 412.5418 on 300×300 chips and Greedy placement with $Q = 20$K). The difference in running times between Sequential and Priority dropped in comparison with the same difference in the BLM case. This is because OSPE is significantly slower with CIM, so the extra time spent re-embedding probes reduces the impact of the extra work with the priority queue. For this reason, Priority I was always within 0.1% of the time required by Sequential, whereas Priority II was at most 11.37% slower.

5.6 Summary

In this chapter, we have presented an extension of the Optimum Single Probe Embedding algorithm (OSPE) of Kahng et al. (2002) that is general enough to work with border length as well as conflict index minimization. We have also surveyed re-embeddings algorithms based on OSPE and presented experimental results with Sequential, the best known algorithm to date.

Table 5.3: Normalized border length (NBL) before and after an optimization phase with various re-embedding algorithms. Placement was produced by the Greedy placement algorithm with border length minimization, 0-threading, and number Q of candidates per spot set to 5K and 20K. In all cases, each re-embedding algorithm executed two passes before the threshold $W = 0.2\%$ was reached. Best results are highlighted in bold.

	Greedy placement		Sequential		Priority I		Priority II	
Dim.	Q	NBL	NBL	Time	NBL	Time	NBL	Time
300×300	5K	18.3182	18.2121	4.8	**18.2115**	5.4	18.2118	22.0
	20K	18.0576	17.9726	4.8	**17.9721**	5.4	17.9723	14.5
500×500	5K	17.5830	17.4851	12.7	**17.4848**	13.9	17.4849	76.7
	20K	17.3554	17.2779	12.6	**17.2776**	13.7	17.2777	63.9
800×800	5K	16.9124	16.8201	32.6	**16.8198**	36.1	16.8199	187.0
	20K	16.6980	16.6258	32.4	**16.6256**	35.3	16.6257	200.0

Table 5.4: Average conflict index (ACI) before and after an optimization phase with various re-embedding algorithms. Placement was produced by the Greedy placement algorithm with conflict index minimization, 0-threading, and number Q of candidates per spot set to 5K and 20K. In all cases, each re-embedding algorithm executed two passes before the threshold $W = 0.2\%$ was reached. Best results are highlighted in bold.

	Greedy placement		Sequential		Priority I		Priority II	
Dim.	Q	NBL	NBL	Time	NBL	Time	NBL	Time
300^2	5K	440.5166	436.8630	188.9	436.8881	190.7	**436.8626**	209.0
	20K	415.5003	412.5536	189.9	412.5613	190.0	**412.5418**	205.1
500^2	5K	432.3023	428.7410	527.3	428.7640	527.2	**428.7375**	581.6
	20K	401.4609	398.6096	528.3	398.6261	530.0	**398.6065**	569.5
800^2	5K	426.0757	422.6277	1357.9	422.6478	1357.9	**422.6223**	1512.2
	20K	392.1786	389.3929	1352.5	389.4075	1355.3	**389.3903**	1488.9

In our results, it is evident that there is little room for improvements by re-embedding probes once a placement is fixed. Nonetheless, we have also introduced a new re-embedding algorithm that attempts to obtain better results by changing the order of re-embeddings based on priorities. We have experimented with two priorities: probes with fewer embeddings first (Priority I) and probes with more re-embedded neighbors (Priority II).

Our results show that our algorithm can achive negligible improvements when compared to Sequential, with Priority I being the best for BLM and Priority II the best for CIM. However, because of the extra time required by Priority, Sequential offers a better trade-off between solution quality and running time, and it should still be the algorithm of choice unless when time is not constrained. The results with our new algorithm also give further indication that the improvements achievable in the re-embedding phase are rather small.

Chapter 6

Partitioning Algorithms

We mentioned earlier that the microarray layout problem is usually approached in two phases: placement and re-embedding. The placement, however, can be preceded by a *partitioning* phase that breaks the problem into smaller sub-problems that are easier to manage. A partitioning algorithm divides the set of probes $\mathcal{P}$ into smaller subsets, and assigns them to defined regions of the chip. Each region can then be treated as an independent chip (and processed by a placement algorithm) or be recursively partitioned. This is especially helpful for placement algorithms with non-linear time or space complexities that are otherwise unable to handle very large chips. Linear-time placement algorithms may also benefit from a partitioning since probes with similar embeddings are typically assigned to the same region — Greedy and Row-Epitaxial (Chapter 3), for instance, are more likely to find good candidates for filling the spots.

We describe four partitioning algorithms: 1-Dimensional Partitioning (1-DP), 2-Dimensional Partitioning (2-DP), Centroid-based Quadrisection (CQ), and Pivot Partitioning (PP). Like placement algorithms, they assume that an initial embedding of the probes is given. Pivot Partitioning is the only algorithm that modifies these embeddings. As we shall see, 1-DP and 2-DP generate a few masks with extremely few conflicts, but leave the remaining masks with many conflicts that are difficult to handle, whereas CQ and PP offer a more uniform optimization over all masks. Earlier results indicate that PP produces better layouts than CQ on large chips (de Carvalho Jr. and Rahmann, 2006a).

Partitioning is clearly a compromise in solution quality since it restricts the space of solutions and may lead to conflicts at partition borders, although it can improve solution quality when the placement algorithm cannot handle large regions well. Hence, it is not advisable to perform too many levels of partitioning because smaller sub-regions mean less freedom for optimization during placement. The right balance depends on the chip dimensions as well as on the placement and partitioning algorithms.

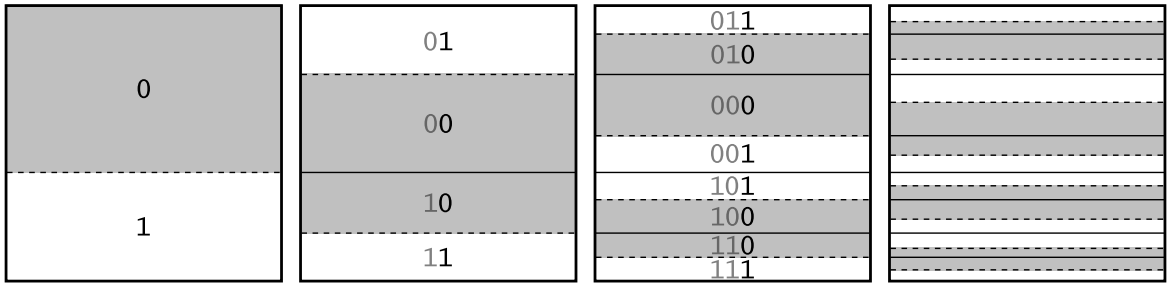


Figure 6.1: First four levels of 1-Dimensional Partitioning. Dashed lines show the divisions performed in each step; solid lines indicate regions delimited in previous steps (there are no border conflicts between spots separated by solid lines). Masked (shaded) regions are labeled "0", unmasked (white) regions are labeled "1". This labeling forms a binary Gray code (shown in the first three steps only).

6.1 1-Dimensional Partitioning

The 1-Dimensional Partitioning algorithm de Carvalho Jr. and Rahmann (2007) divides the set of probes based on the state of their embeddings at a particular synthesis step. It starts by creating two subsets of $\mathcal{P}$:

$$\mathcal{P}_0 = \{p_k \in \mathcal{P} | \varepsilon_{k,1} = 0\}, \qquad \mathcal{P}_1 = \{p_k \in \mathcal{P} | \varepsilon_{k,1} = 1\}.$$

In other words, $\mathcal{P}_0$ contains all probes whose embeddings are unproductive during the first synthesis step, whereas $\mathcal{P}_1$ contains probes with productive embeddings. The chip is then divided into two horizontal (or vertical) bands, proportionally to the number of probes in $\mathcal{P}_0$ and $\mathcal{P}_1$, so each band accommodates one subset of $\mathcal{P}$.

This procedure is recursively applied to each band, using the the next synthesis steps to further divide each subset of probes. For instance, the following subsets of $\mathcal{P}_0$ and $\mathcal{P}_1$ are created during step $t = 2$:

$$\mathcal{P}_{00} = \{p_k \in \mathcal{P}_0 | \varepsilon_{k,2} = 0\}, \qquad \mathcal{P}_{01} = \{p_k \in \mathcal{P}_0 | \varepsilon_{k,2} = 1\},$$

$$\mathcal{P}_{10} = \{p_k \in \mathcal{P}_1 | \varepsilon_{k,2} = 0\}, \qquad \mathcal{P}_{11} = \{p_k \in \mathcal{P}_1 | \varepsilon_{k,2} = 1\}.$$

The next assignments of subsets to the upper or lower band of their regions are made in such a way that regions with the same "state" — productive (unmasked) or unproductive (masked) — are joined as far as possible, resulting in masks that consist of alternating layers of masked and unmasked spots. This process is illustrated in Figure 6.1, where at each step t, a band is labeled "0" when its embeddings are unproductive, and "1" when its embeddings are productive. The resulting binary numbers from top to bottom form a binary Gray code, that is, a permutation of the binary

numbers between 0 and $2^n - 1$ such that neighboring elements differ in exactly one bit, as do the first and last elements (Kreher and Stinson, 1999).

The Gray code highlights an interesting property of 1-DP. After d levels of partitionings (based on steps 1 to d), the embeddings of any two immediate neighbors differ among the first d steps in at most one step. As a result, masks M_1 to M_d exhibit a layered structure that effectively reduces border conflicts. The Gray code is disrupted as soon as a region cannot be divided (because all probes of that region are masked at a particular step, for instance). This will certainly happen as several binary numbers are unlikely to be substrings of embeddings (for example, numbers containing long runs of zeros).

Moreover, 1-DP can optimize only a limited number of masks because the sub-regions soon become too narrow to be further divided. The maximum *partitioning depth* d_{max} is primarily limited by the number of rows (or columns) on the chip. In practice, since regions are likely to be unevenly divided, d_{max} varies between regions. The algorithm can also be configured to stop partitioning a region once its height drops below a given threshold H_{max} (i.e., the maximum height of any final region will not exceed H_{max}).

1-DP is easier to implement if the partitionings always produce rectangular regions (i.e., splitting a row between two regions is not allowed). In order to force an exact division of a region, however, it might be necessary to move a few probes from one subset of probes to the other one.

For example, imagine that a chip with $|\mathcal{P}| = 900$ probes, $n_r = 30$ rows and $n_c = 30$ columns is to be partitioned based on the state of the embeddings at the first synthesis step, resulting in sub-sets $\mathcal{P}_0$ and $\mathcal{P}_1$ with, say, 638 and 262 probes, respectively. The chip must thus be divided into two sub-regions, the upper one containing $[30 \cdot 638/900] = 21$ rows and the lower one with $[30 \cdot 262/900] = 9$ rows (where $[x]$ denotes the nearest integer of x). The problem is that the upper region then contains $21 \cdot 30 = 630$ spots but it has to accommodate 638 probes, whereas the lower region contains $9 \cdot 30 = 270$ spots but only 262 probes. The solution is to (arbitrarily) move 8 probes from $\mathcal{P}_0$ to $\mathcal{P}_1$, which results in some imperfections in the layers of the corresponding mask (a few masked spots in a region of unmasked spots and vice-versa).

6.2 2-Dimensional Partitioning

The 2-Dimensional Partitioning algorithm de Carvalho Jr. and Rahmann (2007) extends the idea of 1-DP to two dimensions, with the potential of optimizing twice as many masks. The algorithm is similar: $\mathcal{P}$ is divided into subsets based on the state of the embeddings at a particular synthesis step. The differences are that 2-DP alternates horizontal and vertical divisions of regions, and that the assignments of probes to regions obey a two-dimensional binary Gray code (Figure 6.2). In a 2-D Gray code,

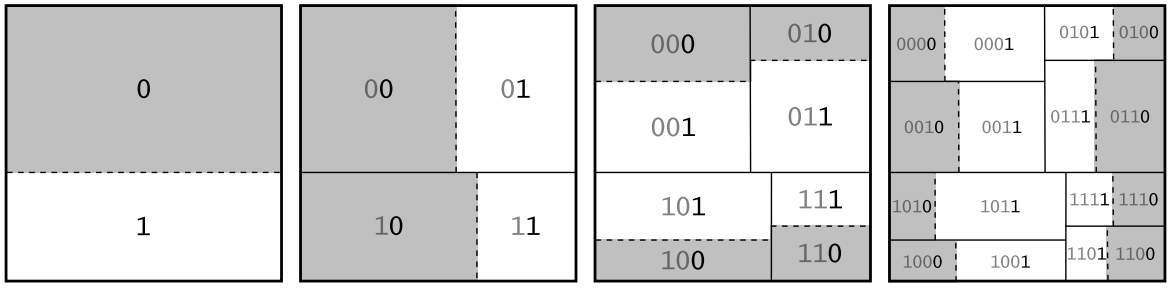


Figure 6.2: First four levels of 2-Dimensional Partitioning. Dashed lines show the divisions performed in each step; solid lines indicate regions delimited in previous steps. Masked regions are labeled with "0", unmasked regions with "1"; this labeling forms an approximation to a two-dimensional binary Gray code.

the binary numbers are arranged in a matrix in such a way that two neighboring numbers differ in at most one bit. As a result, regions whose embeddings are at the same state (productive or unproductive) are joined as far as possible.

If regions were always equally divided, 2-DP would have the same property as 1- DP: After d levels of partitionings, the embeddings of any two immediate neighbors would differ among the first d steps in at most one step. However, this is not always the case since 2-DP is likely to create regions with different dimensions, forcing some regions to share a border with more than its four natural neighbors. For instance, in Figure 6.2 region "0010" borders with "0000", "1010", and "0011", but also with "0001" and "1011".

Like 1-DP, the maximum partitioning depth, d_{max}, is limited by the number of rows and columns on the chip, and it varies since regions are likely to be unevenly divided. 2-DP can also be configured to stop partitioning a region as soon as its dimensions (height and width) drop bellow a given threshold L_{max} (the largest final region will contain at most ${L_{max}}^2$ spots).

Figure 6.3 shows the normalized border length per masking step of layouts produced by 2-DP for a random $1\,000 \times 1\,000$ chip. With maximum partitioning depth ($L_{max} = 1$), 2-DP produced a layout with the best masks for the first 22 synthesis steps. However, because the chip is partitioned until all regions contain a single probe, the placement algorithm has no freedom for reducing border conflicts in the remaining masks. As a result, after step 32, the levels of border conflicts are as high as in the random layout.

With $L_{max} = 10$, there is more room for optimization during placement since the final regions can be as large as 10×10. In this case, we used the Greedy placement algorithm (Section 3.6) with $Q = 100$ so that all probes of a region were considered for filling its spots. This resulted in a reduction of about 13.4% in normalized border length compared to the layout produced with $L_{max} = 1$ (from 21.5588 to 18.6670, data

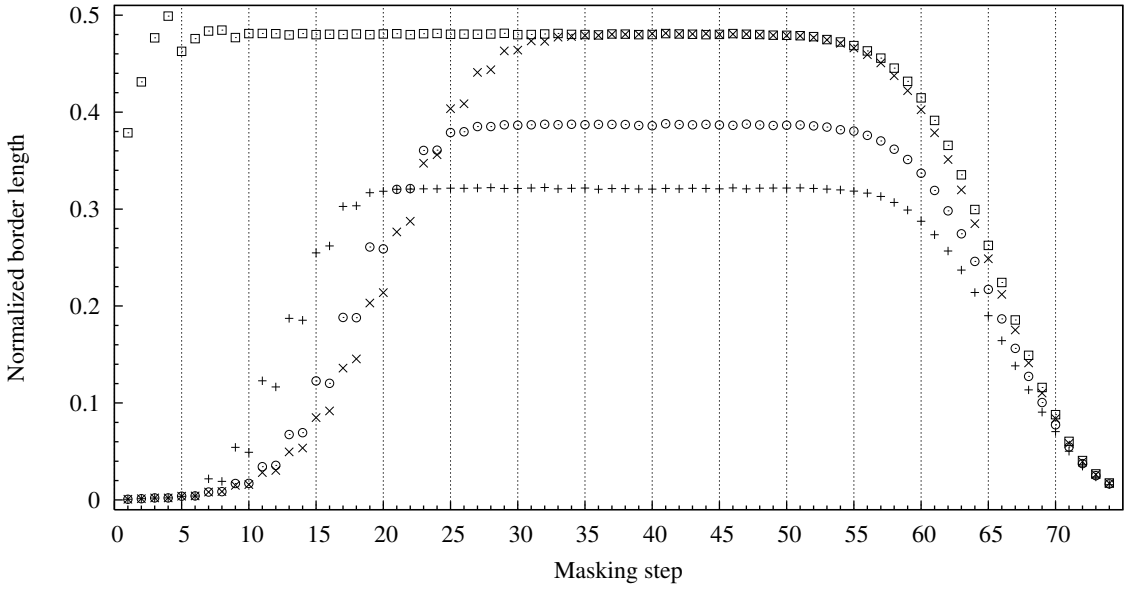


Figure 6.3: Normalized border length per masking step of several layouts for a 1 000 × 1 000 chip with random probes left-most embedded in the standard Affymetrix depostion sequence: random layout (⊡); 2-D Partitioning with $L_{max} = 1$ (×); 2-D Partitioning with $L_{max} = 10$ and Greedy placement with $Q = 100$ (⊙); 2-D Partitioning with $L_{max} = 50$ and Greedy placement with $Q = 2\,500$ (+).

not shown), although we observed an increase of border conflicts in the first 24 masks. Increasing L_{max} even further to 50 and using Greedy with $Q = 2\,500$ resulted in a reduction of 8.1% in normalized border length compared to $L_{max} = 10$ (from 18.6670 to 17.1629) but, again, this came at the expense of an increase of border conflicts in the first 20 masks.

Figure 6.4 compares the results obtained by 1-DP and 2-DP on the same $1\,000 \times 1\,000$ chip of Figure 6.3. We first compare both algorithms with their maximum partitioning depths ($H_{max} = 1$ for 1-DP and $L_{max} = 1$ for 2-DP). With $L_{max} = 1$, 2-DP produces 1×1 regions and leaves no room for optimization during placement. In constrast, 1-DP with $H_{max} = 1$ produces regions with a single row but, in this case, with 1 000 columns (and 1 000 probes), leaving a considerable degree of freedom for the placement algorithm. To be fair, we thus compare 1-DP and 2-DP using a placement algorithm that places probes randomly inside each final region, so that the results are only due to the partitionings (and not to the placement algorithm). In our results, with maximum partitioning depths, 1-DP and 2-DP produced layouts with similar levels of border conflicts in masks M_{33} to M_{74}, although the layout produced by 2-DP was slightly better in masks M_{58} to M_{69}. However, while 1-DP was able to produce masks with relatively few conflicts in the first 17 steps, 2-DP achieved even greater reductions of border conflicts in the first 32 steps. The normalized border length of these layouts

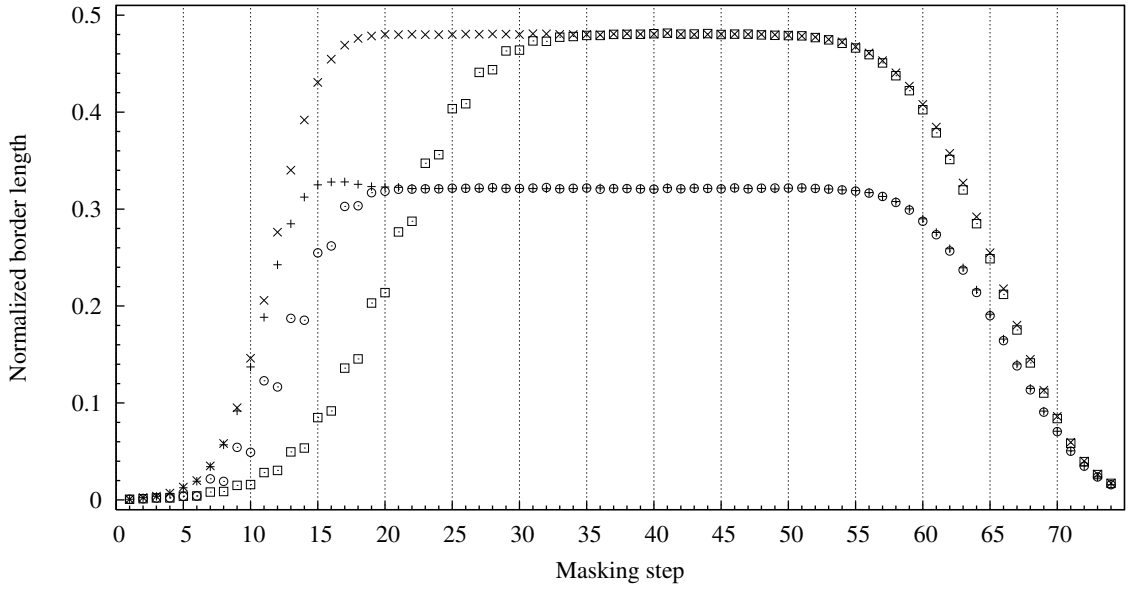


Figure 6.4: Normalized border length per masking step of layouts produced by 1-D and 2-D Partitioning for a $1\,000 \times 1\,000$ chip with random probes left-most embedded in the standard Affymetrix depostion sequence: 1-DP with $H_{max} = 1$ and random placement ($\times$); 2-DP with $L_{max} = 1$ ($\boxdot$); 1-DP with $H_{max} = 1$ and Greedy placement with $Q = 1\,000$ ($+$); 2-DP with $L_{max} = 50$ and Greedy placement with $Q = 2\,500$ ($\odot$).

are 25.8543 (with 1-DP) and 21.5588 (with 2-DP).

In Figure 6.4 we also compare 1-DP with $H_{max} = 1$ and 2-DP with $L_{max} = 50$ using Greedy for the placement. With $L_{max} = 50$, 2-DP produces regions containing at most 2 500 probes. For this particular chip, 2-DP produced 1 005 regions, containing 995.02 probes on average (the largest region contained 2 209 and the smallest 210 probes), so Greedy had about the same degree of freedom provided by 1-DP with $H_{max} = 1$. We used a sufficiently large number Q of candidates per spot so that all probes of a region were considered for filling its spots. With these settings, the layouts produced by 1-DP and 2-DP have similar levels of border conflicts in masks M_{20} to M_{74}. In the first 18 synthesis steps, however, 2-DP produced better masks, especially after step 5. The normalized border length of these layouts are 18.0078 (1-DP) and 17.1629 (2-DP).

A representation of selected photolithographic masks generated by 2-DP for a 300×300 chip are shown in Figure 6.5. The resulting rectangular regions can be clearly seen up to mask M_{18}. In the first eight masks it is possible to see some "imperfections" (unmasked spots on masked regions or vice-versa) that result from arbitrarily moving probes between regions in order to force exact divisions. On a chip of this size, 2-DP can usually reduce conflicts up to the 25^{th} synthesis step, although this is not noticeable in M_{25} of Figure 6.5.

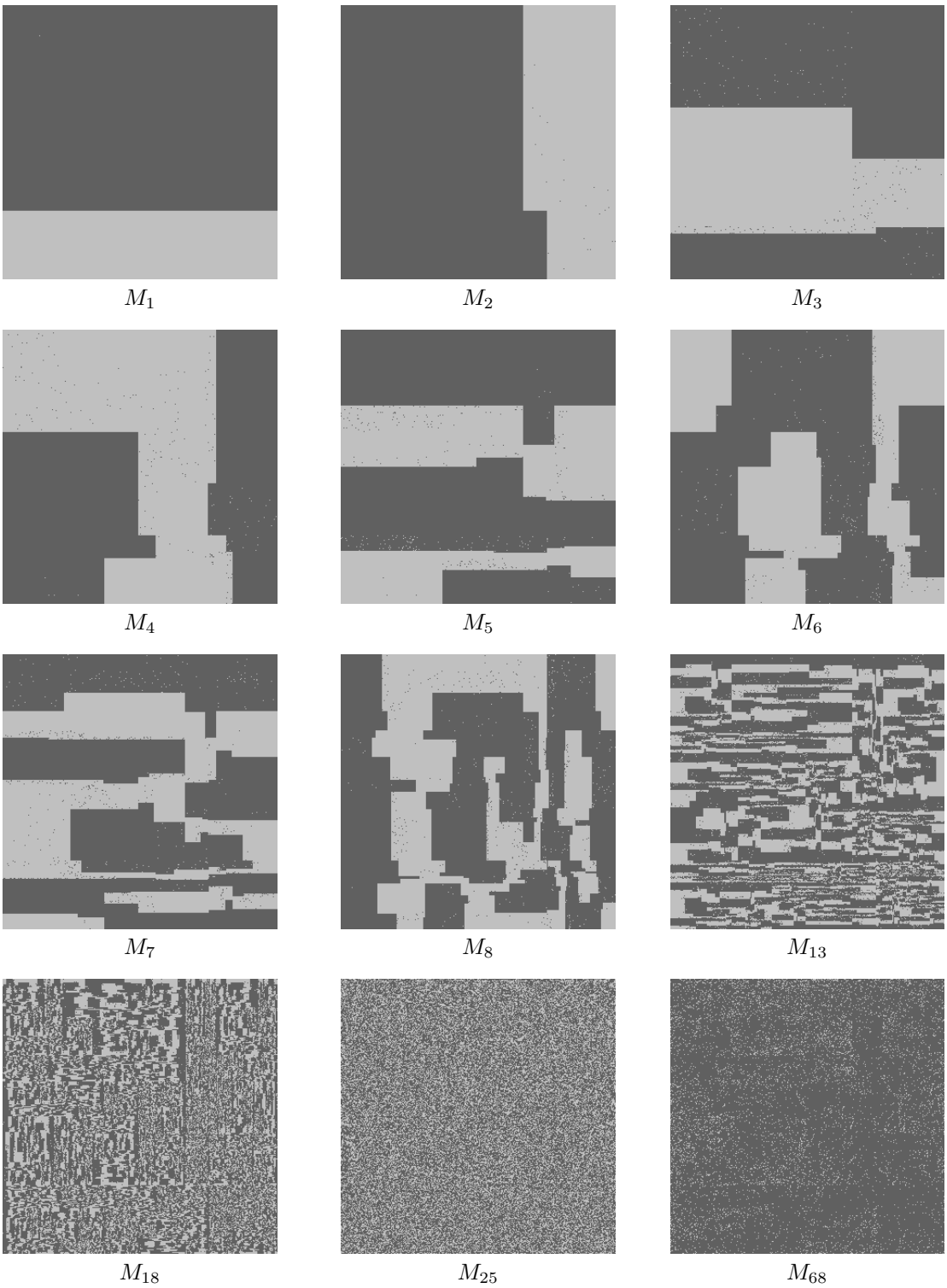


Figure 6.5: Selected masks generated by 2-Dimensional Partitioning with $L_{max} = 1$ for a random 300×300 chip with 25-mer probes left-most embedded into the standard Affymetrix deposition sequence. Unmasked (masked) spots are represented by light (dark) dots.

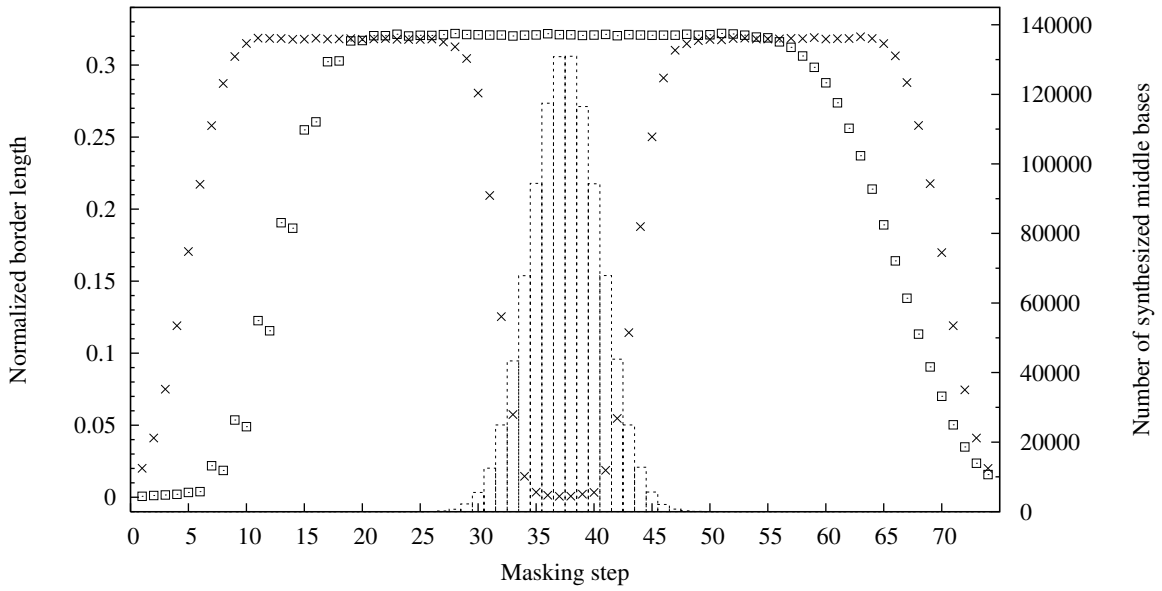


Figure 6.6: Normalized border length per masking step (on the left y-axis) of two layouts produced by 2-Dimensional Partitioning with $L_{max} = 50$ and Greedy placement with border length minimization and $Q = 2.5$K for a $1\,000 \times 1\,000$ chip with random probe sequences: left-most mask optimization with left-most embeddings (⊡); centered mask optimization with centered embeddings (×). The number of middle bases synthesized at each step (with centered embeddings) is shown in boxes (right y-axis).

So far we have described both 1-DP and 2-DP using the state of the first d synthesis steps to divide the set of probes. The result of this approach is that, while the first masks are optimized, the remaining masks are left with high levels of border conflicts; we call this a *left-most mask optimization*.

However, a defect in the middle of the probe is more harmful than in its extremities, so it is more important to optimize the central masks that are more likely to add the middle bases. Fortunately, it is possible to reduce conflicts in the central masks using 1-DP and 2-DP by partitioning the probe set based on the following sequence of synthesis steps, assuming that T is even and d is odd: $T/2, (T/2)\pm1, (T/2)\pm2, \ldots, (T/2)\pm\lfloor d/2 \rfloor$; we call this a *centered mask optimization*.

For left-most optimization, it makes sense to embed the probes in a left-most fashion in order to reduce conflicts in the last masks (which are not optimized by the partitioning). The left-most embeddings reduce the number of unmasked spots in the last steps, resulting in masks that largely consist of masked spots and consequently low levels of border conflicts. In contrast, centered mask optimization produces better results with *centered* embeddings. A centered embedding is constructed by shifting a left-most embedding to the right until the number of masked steps to the left of the first productive step is approximately equal to the number of masked steps to the

Table 6.1: Average conflict index (ACI) of layouts produced by Greedy placement and 2-D Partitioning on random 800×800 chips with left-most and centered embeddings. 2-DP was configured for centered mask optimization and used Greedy for the placement. In all cases, Greedy was configured for conflict index minimization and used 0-threading. Results are averages over a set of five arrays and running times are reported in minutes.

Embeddings	Algorithm	ACI	Time
Left-most	Greedy with $Q = 20$K	392.1786	186.4
Left-most	Greedy with $Q = 40$K	378.3110	357.0
Left-most	Greedy with $Q = 80$K	366.8446	680.9
Centered	Greedy with $Q = 20$K	387.5974	205.1
Centered	2-DP with $L_{max} = 10$ and Greedy with $Q = 100$	345.9908	0.6
Centered	2-DP with $L_{max} = 20$ and Greedy with $Q = 400$	342.2031	1.3
Centered	2-DP with $L_{max} = 30$ and Greedy with $Q = 900$	**341.2786**	2.3
Centered	2-DP with $L_{max} = 40$ and Greedy with $Q = 1\,200$	341.6185	4.0
Centered	2-DP with $L_{max} = 50$ and Greedy with $Q = 2\,000$	341.7515	6.1
Centered	2-DP with $L_{max} = 60$ and Greedy with $Q = 3\,600$	341.8634	8.4

right of the last productive step.

Figure 6.6 shows the results of using 2-D Partitioning with $L_{max} = 50$ on a $1\,000 \times 1\,000$ chip with left-most and centered mask optimization. With left-most mask optimization, we obtain a normalized border length of 17.1629 (up to approximately 0.32 per step). With centered mask optimization, the normalized border length improves by 1.03% to 16.9855 (not shown in the figure). The average conflict index, however, is reduced by as much as 34.89% (from 577.3353 to 375.9232) because of the higher weight of the middle bases in the conflict index measure.

When carefully used, 1-DP and 2-DP can improve placement by producing a few masks with very low levels of border conflicts, and breaking the problem into smaller subproblems that are easier to handle. Table 6.1 shows results on 800×800 arrays using 2-DP with centered mask optimization and Greedy with conflict index minimization for the placement, in comparison to using Greedy alone (results with Greedy as shown on Table 3.2 and Figure 3.5). Results of Greedy with centered embeddings are also shown. In our results, the layouts produced by 2-DP are even better than the ones produced by Greedy with $Q = 80$K. This is a consequence of the importance of the middle bases in the conflict index measure. Moreover, while Greedy required about 680.9 minutes with $Q = 80$K, the combination of 2-DP and Greedy required at most 8.4 minutes because the partitioning restricts the number of candidates Greedy can look at for each spot.

Increasing L_{max} provides more room for optimization during placement but worsens the central masks, while reducing L_{max} improves the central masks at the expense of an increase of conflicts in the remaining masks (in this case, reducing L_{max} also improves running time as Greedy has fewer candidates available for each spot). The best trade-off depends on several aspects of the problem such as chip dimension, probe

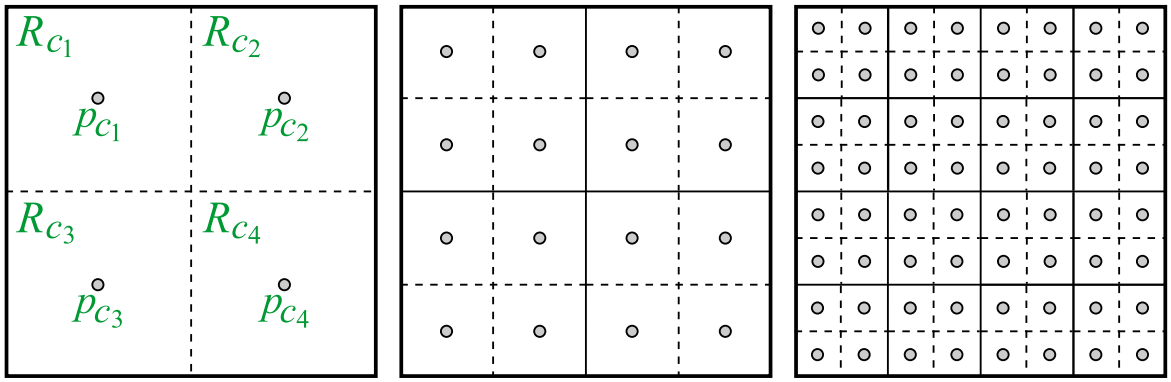


Figure 6.7: First three levels of Centroid-based Quadrisection. Dashed lines show the divisions performed in each step; solid lines indicate regions delimited in previous steps. The centroids of each partition R_{c_1} to R_{c_4} are represented by small circles (labeled with p_{c_1} to p_{c_4} in the first step).

embeddings, type of optimization (border length or conflict index), and placement algorithm. For this case, the best results were achieved with $L_{max} = 30$.

6.3 Centroid-based Quadrisection

Centroid-based Quadrisection (Kahng et al., 2003b), CQ for short, employs a different criterion for dividing the probe set and a different approach for partitioning. At each iteration, a region R is quadrisectioned into R_{c_1}, R_{c_2}, R_{c_3}, and R_{c_4}. Each sub-region R_{c_i} is associated with a selected probe $p_{c_i} \in \mathcal{P}$, called *centroid*, that is used to guide the assignment of the remaining probes to the sub-regions.

A centroid is a representative of its region: It should symbolize the "average embedding" in that region. The remaining probes $p_k \in \mathcal{P} \setminus \{p_{c_1}, p_{c_2}, p_{c_3}, p_{c_4}\}$ are compared to each centroid and assigned to the sub-region R_{c_i} whose centroid minimize the Hamming distance $H(k, c_i)$ (as defined in Section 2.2).

The authors argue that, in order to improve the "clustering" of similar probes, the four centroids should be as different from each other as possible. The following heuristic is proposed: First, a probe index c_1 is randomly selected from $\{1, \ldots, |\mathcal{P}|\}$. Then, a probe index $c_2 \neq c_1$ maximizing $H(c_2, c_1)$ is selected. Similarly, c_3 maximizing $H(c_3, c_1) + H(c_3, c_2)$ and c_4 maximizing $H(c_4, c_1) + H(c_4, c_2) + H(c_4, c_3)$ are selected. The assignment of centroids to the quadrisections of the chip is arbitrary.

Since the partitioning must always produce four regions of the same size, sometimes it is necessary to make non-optimal assignment of probes to regions. In order to recover from a possibly bad choice of centroids, a "multi-start heuristic" is used, running the centroid selection procedure several times with different "seeds" for c_1 and keeping the

centroids that lead to the best partitioning. For measuring partitioning quality, the algorithm uses the sum of Hamming distances between the embeddings of the probes and the embedding of the centroid (the partitioning that results in the least sum is selected).

The maximum partitioning depth d_{max} of CQ is $\log_2 n_r$, assuming that n_r is a power of 2 and that $n_c = n_r$ (n_r and n_c are the number of rows and columns on the chip, respectively). In practice, the partitioning continues until a pre-defined depth D has been reached.

Although CQ was developed for border length minimization (BLM), it can be adapted for conflict index minimization (CIM) by using the *conflict index distance* $C(k, k')$ (as defined in Section 2.3) instead of the Hamming distance $H(k, k')$ for selecting the centroids as well as for deciding which partition a probe should be assigned to.

As mentioned in Section 3.6, placement algorithms such as Row-Epitaxial and Greedy have the drawback of treating the last $Q - 1$ filled spots unfairly since fewer than Q probe candidates are available to fill them. This issue is aggravated by a partitioning because in each final partition $Q - 1$ spots have fewer than Q probe candidates. In order to attenuate this problem, a *borrowing heuristic* was implemented in CQ to allow the placement algorithm (Row-Epitaxial, in the original implementation) to look at Q probes "in the current and the next region". Although the authors did not specify the exact meaning of "next region", it can be, for instance, the next region to be processed by the placement algorithm. Borrowing probes from a region R_{c_i} to fill spots of R_{c_j} obviously requires using the unplaced probes of R_{c_j} to fill spots of R_{c_i}.

6.4 Pivot Partitioning

Pivot Partitioning (de Carvalho Jr. and Rahmann, 2006a), PP for short, is to a certain extent similar to CQ: Sub-regions are recursively associated with special probes, here called *pivots* instead of centroids, that are used to guide the assignment of the other probes to the sub-regions. The main differences between PP and CQ are as follows.

Instead of quadrisectioning the chip, PP creates sub-regions by alternating horizontal and vertical divisions (like 2-D Partitioning). At each iteration, a region R is partitioned into sub-regions R_{c_1} and R_{c_2} associated with pivots q_{c_1} and q_{c_2}, respectively. The advantage of alternating horizontal and vertical divisions over the quadrisectioning approach of CQ is that regions are not required to have the same size. Instead, regions are divided proportionally to the size of each subset of probes, which reduces the need for making non-optimal assignments, although it may still be necessary to move some probes from one sub-region to the other in order to obtain rectangular regions. Moreover, for each partitioning, only two pivots need to be selected.

Algorithm 1 PivotPartitioning

Input: rectangular region R consisting of all rows and columns of the chip,
set of probes $\mathcal{P} = \{p_1, p_2, \ldots p_n\}$,
deposition sequence N,
and requested partitioning depth D
Output: set of assignments $\mathcal{A} = \{a_1, a_2, \ldots a_{2^D}\}$
where $a_i = (\mathcal{P}_i, R_i)$, $\mathcal{P}_i \subset \mathcal{P}$, and R_i is a sub-region of the chip

1. (Select pivot candidates.) Select probes $p \in \mathcal{P}$ with minimum number of embeddings $E(p)$ as pivot candidates:
   a) Let $\mathcal{Q} = \{p \in \mathcal{P} \mid E(p, N) \text{ is minimum}\}$
   b) Set $\mathcal{P} \leftarrow \mathcal{P} \setminus \mathcal{Q}$
2. (Call RecursivePartitioning.) Call recursive procedure with initial partitioning depth 1 and return:
   a) Return RecursivePartitioning $(1, D, R, \mathcal{Q}, \mathcal{P})$

Another distinction is motivated by the same observation that inspired the development of the Priority re-embedding algorithm (Section 5.5), i.e., that different probes have different numbers of embeddings, ranging from a single one to several millions on a typical Affymetrix GeneChip array. Probes with more embeddings can more easily adapt to the other probes, that is, they are more likely to have an embedding with fewer conflicts to fill a particular spot than a probe that has only a limited number of embeddings. PP uses probes with a single embedding (or few embeddings) as pivots, and chooses the other probes' embeddings and region assignments accordingly. Indeed, the most important feature of PP is the simultaneous embedding and assignment of probes to sub-regions.

The first part of the algorithm consists of selecting a sub-set of probes that will be used as pivots (Algorithm 1). First, it examines each probe $p \in \mathcal{P}$ and computes $E(p, N)$, the number of embeddings of p in the deposition sequence N; this can be done in $O(\ell \cdot T)$ time with dynamic programming, where ℓ is the length of the probe and T is the length of the deposition sequence. The set of pivot candidates $\mathcal{Q}$ then consists of all probes p with $E(p, N) = 1$. In practice, this usually results in a sufficient number of pivots. For instance, around 6% of the probes in a randomly generated chip have a single embedding. If this is not the case, we can set a threshold e for the maximum number of embeddings of a pivot in such a way that the number of probes p with $E(p, N) \leq e$ is at least 2^D, where D is the requested partitioning depth (a user-defined parameter).

Using probes with fewer embeddings as pivots has two advantages. First, less time is spent choosing the pivots in each iteration since fewer candidates need to be examined. Second, probes with fewer embeddings are usually better "representatives" to drive the partitioning. The problem is that some embeddings may have their productive steps concentrated in one part of the deposition sequence. For instance, some Affymetrix

Algorithm 2 RecursivePartitioning with conflict index minimization

Input: current partitioning depth d,
requested partitioning depth D,
rectangular region R of the chip,
set of pivot candidates $\mathcal{Q}$,
and set of probes $\mathcal{P}$,
Output: set of assignments $\mathcal{A} = \{a_1, a_2, \ldots a_{2^{(D-d)}}\}$
where $a_i = (\mathcal{P}_i \cup \mathcal{Q}_i, R_i)$, $\mathcal{P}_i \subset \mathcal{P}$, $\mathcal{Q}_i \subset \mathcal{Q}$, and R_i is a sub-region of R

1. (Stopping condition.) When $d = D$:
   a) Re-embed each $p \in \mathcal{P}$ optimally with respect to all $q \in \mathcal{Q}$
   b) Return $\{(\mathcal{P} \cup \mathcal{Q}, R)\}$
2. (Choose pivot pair.) Select $q_{c_1}, q_{c_2} \in \mathcal{Q}$ such that $C(c_1, c_2)$ is maximal
3. (Partition set of pivot candidates.) Assign each pivot candidate $q_k \in \mathcal{Q}$ to sub-set $\mathcal{Q}_{c_j}$ associated with pivot q_{c_j} such that $C(k, c_j)$ is minimal; in case of ties, make assignments heuristically in an attempt to achieve balanced partitionings:
   a) $\mathcal{Q}_{c_1} = \{q_k \in \mathcal{Q} \mid C(k, c_1) < C(k, c_2)\}$
   b) $\mathcal{Q}_{c_2} = \{q_k \in \mathcal{Q} \mid C(k, c_1) > C(k, c_2)\}$
4. (Partition probe set.) Assign each probe $p_k \in \mathcal{P}$ to sub-set $\mathcal{Q}_{c_j}$ such that $M_C(k, c_j)$ is minimal; in case of ties, make assignments heuristically in an attempt to achieve balanced partitionings:
   a) $\mathcal{P}_{c_1} = \{p_k \in \mathcal{P} \mid M_C(k, c_1) < M_C(k, c_2)\}$
   b) $\mathcal{P}_{c_2} = \{p_k \in \mathcal{P} \mid M_C(k, c_1) > M_C(k, c_2)\}$
5. (Partition chip region.) Partition R into sub-regions R_{c_1} and R_{c_2} (vertically if d is even, horizontally otherwise) proportionally to the number of probes in $\mathcal{P}_{c_1} \cup \mathcal{Q}_{c_1}$ and $\mathcal{P}_{c_2} \cup \mathcal{Q}_{c_2}$
6. (Proceed recursively.) Partition each sub-problem recursively and return:
   a) Return RecursivePartitioning $(d + 1, D, R_{c_1}, \mathcal{Q}_{c_1}, \mathcal{P}_{c_1})$
   $\cup$ RecursivePartitioning $(d + 1, D, R_{c_2}, \mathcal{Q}_{c_2}, \mathcal{P}_{c_2})$

probes, when left-most embedded, are synthesized in the first 37 masking steps, thus using only half of the total 74 steps. Such probes are not good choices for pivots. In our experience, probes with fewer embeddings are better pivots because they cover most (if not all) cycles of the deposition sequence.

Once the pivot candidates are selected, the main recursive procedure is called (Algorithm 2). The output of this procedure is a set of assignments $\mathcal{A} = \{a_1, a_2, \ldots a_{2^D}\}$, where each $a_i = (\mathcal{P}_i \cup \mathcal{Q}_i, R_i)$, i.e., a_i consists of a set of probes (pivots and non-pivots) and a defined sub-region R_i of the chip. Each assignment can then be processed, independently, by a placement algorithm.

At Step 2 of Algorithm 2, a pair of pivots q_{c_1} and $q_{c_2} \in \mathcal{Q}$ is selected such that the conflict index distance between their embeddings $C(c_1, c_2)$ is maximal; in case of BLM, the Hamming distance $H(c_1, c_2)$ is used. Instead of checking every possible pair of pivots, the following heuristic is applied: First, a probe index c_1 is randomly selected from $\{1, \ldots, |\mathcal{Q}|\}$. Then, a probe index $c_2 \neq c_1$ maximizing $C(c_2, c_1)$ is selected. This

procedure is repeated for a fixed number of times, and the pair with maximum $H(c_1, c_2)$ is used in this iteration.

Step 3 partitions the set of pivot candidates $\mathcal{Q}$ into sub-sets $\mathcal{Q}_{c_1}$ and $\mathcal{Q}_{c_2}$ associated with pivots q_{c_1} and q_{c_2}, respectively. This is done by comparing each of the remaining pivot candidates $q_k \in \mathcal{Q}$ with q_{c_1} and q_{c_2} and assigning it to the sub-set $\mathcal{Q}_{c_j}$ whose pivot results in minimum $C(k, c_j)$ over $j = 1, 2$, or minimum $H(k, c_j)$ in case of BLM.

A similar approach is used to partition the set of non-pivot probes $\mathcal{P}$ into sub-sets $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_2}$ (Step 4). The difference is that a non-pivot probe p_k is assigned to a sub-set $\mathcal{P}_{c_j}$ considering all valid embeddings of p_k with respect to the embedding of pivot q_{c_j}. This is done by computing the *minimum conflict index distance* $M_C(k, c_j)$ or the *minimum Hamming distance* $M_H(k, c_j)$ in case of BLM. $M_C(k, c_j)$ is defined as the minimum conflict index distance $C(x, c_j)$ between any embedding ε_x of p_k and a fixed embedding ε_{c_j} (see Section 2.3 for the definition of conflict index distance). Similarly, $M_H(k, c_j)$ is defined as the minimum Hamming distance $H(x, c_j)$ between any embedding ε_x of p_k and ε_{c_j} (see Section 2.2 for the definition of Hamming distance).

$M_C(k, c_j)$ and $M_H(k, c_j)$ are computed with the OSPE algorithm of Section 5.1. However, since at this point the probes have not yet been assigned to spots, we use a variant of OSPE that ignores the location of the probes (and thus the distance-dependent weights γ) by setting the U_t and $M_{i,t}$ costs (Equations 5.1 and 5.2), in the CIM case, as follows:

$$U_t := \mathbb{1}_{\{\varepsilon_{c_j},t=0\}} \cdot \omega(\varepsilon_{c_j}, t),$$

$$M_{i,t} := c \cdot \exp(\theta \cdot (1 + \min\{i, \ell - i\})) \cdot \mathbb{1}_{\{\varepsilon_{c_j},t=1\}}.$$

At Step 5, the region R is partitioned into sub-regions R_{c_1} and R_{c_2} proportionally to the number of probes in $\mathcal{P}_{c_1} \cup \mathcal{Q}_{c_1}$ and $\mathcal{P}_{c_2} \cup \mathcal{Q}_{c_2}$. The algorithm alternates between vertical (if current partitiong depth d is even) and horizontal (if d is odd) divisions.

Pivot Partitioning continues recursively up to a pre-defined maximum partitioning depth D. When $d = D$, it returns an assignment of all probes of $\mathcal{P} \cup \mathcal{Q}$ (pivots and non-pivots) to region R (Step 1). Before that, however, the algorithm re-embeds each probe $p_k \in \mathcal{P}$ optimally with respect to all pivots $q_j \in \mathcal{Q}$ using another variant of OSPE with costs U_t and $M_{i,t}$, in case of CIM, set as follows:

$$U_t := \sum_{q_j \in \mathcal{Q}} \mathbb{1}_{\{\varepsilon_{j,t}=0\}} \cdot \omega(\varepsilon_j, t),$$

$$M_{i,t} := c \cdot \exp(\theta \cdot (1 + \min\{i, \ell - i\})) \cdot \sum_{q_j \in \mathcal{Q}} \mathbb{1}_{\{\varepsilon_{j,t}=1\}}.$$

Table 6.2: Comparison between Pivot Partitioning (PP) and Centroid-based Quadrisection (CQ) on chips containing random probes sequences of length 25 embedded in a 100-step deposition sequence (probes are, initially, synchronously embedded). Chip dimensions range from 100×100 to 500×500. Partitioning depths vary from $D = 1$ to $D = 3$ for CQ and, equivalently, from $D = 2$ to $D = 6$ for PP. Both partitionings use Row-Epitaxial for the placement with 1-threading and $Q = 20\,000$, and are followed by the Sequential re-embedding algorithm with threshold $W = 0.1\%$. The data shows the normalized border length of chips produced by CQ as reported by Kahng et al. (2003b), and the results of using PP on similar input. The relative diffence between the two algorithms is shown in percentage.

	100×100	200×200	300×300	500×500
	NBL	NBL	NBL	NBL
CQ $D = 1$	19.8595	19.1558	19.4735	19.1310
PP $D = 2$	**19.7414**	**18.6572**	**17.9959**	**17.3154**
Relative	−0.60%	−2.60%	−7.59%	−9.49%
CQ $D = 2$	**20.1673**	19.4199	19.0263	18.7480
PP $D = 4$	20.4057	**19.1756**	**18.4533**	**17.6462**
Relative	+1.18%	−1.26%	−3.01%	−5.88%
CQ $D = 3$	**20.7378**	**19.7625**	19.1470	18.6523
PP $D = 6$	21.1305	19.8459	**19.0458**	**18.1701**
Relative	+1.89%	+0.42%	−0.53%	−2.59%

6.4.1 Results

Table 6.2 shows a comparison between Pivot Partitioning and Centroid-based Quadrisection. For this comparsion, we reproduce the results of Kahng et al. (2003b), which used chips with random probes of length $\ell = 25$ that were, initially, synchronously embedded in a cyclic deposition sequence of length $N = 100$. We run PP on similar input and report the results with equivalent partitioning depths (two levels of PP are equivalent to one level of CQ). Both algorithms were configured for BLM and used 1-threading and Row-Epitaxial for the placement with $Q = 20\,000$. Since PP also modifies the probes' embeddings, we compare the results obtained by both algorithms after a re-embedding phase with Sequential (Section 5.4) using threshold $W = 0.1\%$.

Our results show that PP produced layouts with less border conflicts than CQ except on the smaller chips with higher partitioning depths. On 500×500 chips, for instance, PP with $D = 2$ produced a layout with 9.49% less border conflicts than CQ with $D = 1$, on average. With $D = 6$ (respectively, $D = 3$ for CQ), this difference droped to 2.60%. On 100×100 chips, however, PP produced worse layouts, with up to 1.89% more border conflicts with $D = 6$. We suspect that this disadvantage is due to the "borrowing heuristic" used by CQ (and not implemented in PP) that permits, during placement, borrowing probes from neighboring partitions in order to maintain a high number of probe candidates for filling the last spots of a quadrant.

We also report results of similar experiments using PP and the Greedy placement

Table 6.3: Normalized border length (NBL) and average conflict index (ACI) of layouts produced by the Greedy placement algorithm and Pivot Partitioning (PP) with varying partitioning depths D on chips containing random probes embedded in a deposition sequence of length 100 (probes are, initially, synchronously embedded). PP uses Greedy for placement inside final regions. In all cases, Greedy uses $Q = 20\,000$ and 0-threading, and placement is followed by a re-embedding phase with Sequential using threshold $W = 0.1\%$. Total time (including partitioning, placement and re-embedding) is reported in seconds.

	200 × 200		300 × 300		500 × 500	
	NBL	Time	NBL	Time	NBL	Time
Greedy	20.7696	173.8	20.2921	560.5	19.5884	2 214.3
PP $D = 2$	18.6572	50.3	17.9959	335.8	17.3154	1 921.2
Relative	−10.17%	−71.0%	−11.32%	−40.1%	−11.60%	−13.2%
PP $D = 4$	19.1756	26.6	18.4533	92.2	17.6462	913.6
Relative	−7.67%	−84.7%	−9.06%	−83.6%	−9.92%	−58.7%
PP $D = 6$	19.8459	23.3	19.0458	60.2	18.1701	254.4
Relative	−4.45%	−86.6%	−6.14%	−89.3%	−7.24%	−88.5%

	200 × 200		300 × 300		500 × 500	
	ACI	Time	ACI	Time	ACI	Time
Greedy	469.6163	1 077.8	454.7646	2 780.5	440.8775	8 151.0
PP $D = 2$	410.9014	533.2	396.1600	1 799.2	380.6258	6 940.4
Relative	−12.50%	−50.5%	−12.89%	−35.3%	−13.67%	−14.9%
PP $D = 4$	426.4966	406.3	409.6784	1 024.0	389.2871	4 505.6
Relative	−9.18%	−62.3%	−9.91%	−63.2%	−11.70%	−44.7%
PP $D = 6$	444.0277	366.1	425.2855	891.5	403.9497	3 038.1
Relative	−5.45%	−66.0%	−6.48%	−67.9%	−8.38%	−62.7%

algorithm compared to using Greedy alone. For these experiments, we used versions of PP and Greedy for border length as well as conflict index minimization (Table 6.3). In all cases, we run the Sequential re-embedding algorithm with threshold $W = 0.1\%$ after placement.

Our results show that PP improves the quality of layouts in both measures at the same time that it significantly reduces running time. The best layouts were invariably achived with $D = 2$ and the improvements were higher on larger chips. The reduction in normalized border length was up to 11.60% (from 19.5884 to 17.3154) on 500 × 500 chips with $D = 2$ when compared with no partitioning. In this particular case, there was also a reduction of 13.2% in running time (from 2 214.3 to 1 921.2 seconds). With CIM, the reduction in average conflict index was up to 13.67% (from 440.8775 to 380.6258) on 500 × 500 chips with $D = 2$ when compared with no partitioning. Increasing the partitioning depth up to $D = 6$ still resulted in better layouts, although with relatively less reduction in normalized border length and average conflict index when compared to $D = 2$. In terms of running time, however, we observed a reduction of as much as 89.3% in the BLM case (from 560.5 to 60.2 seconds) and 67.9% in the CIM case (from 2 780.5 to 891.5 seconds) on 300 × 300 chips with $D = 6$ when

Table 6.4: Normalized border length (NBL) of layouts produced by the Greedy placement algorithm and Pivot Partitioning (PP) with varying partitioning depths D on chips containing random probes embedded in the standard Affymetrix deposition sequence (of length 74; probes are, initially, left-most embedded). PP uses Greedy for placement inside final regions. In all cases, Greedy uses Q as indicated and 0-threading, and placement is followed by a re-embedding phase with Sequential using threshold $W = 0.2\%$. Total time (including partitioning, placement and re-embedding) is reported in seconds.

		300×300		500×500		800×800	
Q	Alg.	NBL	Time	NBL	Time	NBL	Time
5K	Greedy	18.2121	103.3	17.4851	358.4	16.8201	949.4
	PP $D = 2$	18.4376	87.0	17.8102	315.3	17.1683	922.0
	Relative	+1.24%	−15.7%	+1.86%	−12.0%	+2.07%	−2.9%
	PP $D = 4$	18.6193	58.7	17.9299	267.9	17.3763	885.2
	Relative	+2.24%	−43.2%	+2.54%	−25.3%	+3.31%	−6.8%
	PP $D = 6$	19.1262	31.2	18.2090	149.8	17.5295	671.6
	Relative	+5.02%	−69.8%	+4.14%	−58.2%	+4.22%	−29.3%
20K	Greedy	17.9726	582.7	17.2779	2 012.4	16.6258	5 782.1
	PP $D = 2$	18.1954	295.6	17.5494	1 612.5	16.9620	5 083.1
	Relative	+1.24%	−49.3%	+1.57%	−19.9%	+2.02%	−12.1%
	PP $D = 4$	18.6124	61.2	17.7584	696.6	17.1114	3 924.4
	Relative	+3.56%	−89.5%	+2.78%	−65.4%	+2.92%	−32.1%
	PP $D = 6$	19.1262	31.3	18.2083	150.5	17.4450	1 158.8
	Relative	+6.42%	−94.6%	+5.38%	−92.5%	+4.93%	−80.0%

compared with no partitioning.

It should be noted that the results shown on Tables 6.2 and 6.3 use a deposition sequence of length $T = 100$, which allows a considerable degree of freedom for embedding probes of length $\ell = 25$; these experiments were mainly performed to compare PP with previous results on CQ. In practice, the production of commercial microarrays is likely to use shorter deposition sequences. Affymetrix chips, for instance, are synthesized in 74 synthesis steps. For this reason, we also show the results of using Pivot Partitioning on chips with random 25-mer probes left-most embedded in the standard Affymetrix deposition sequence. In these experiments we use the Greedy placement algorithm with $Q = 5\,000$ and $Q = 20\,000$, and we report the results of PP compared with layouts produced with no partitioning (using Greedy alone).

With BLM (Table 6.4), we observed that partitioning the chip always resulted in worse layouts than without partitioning, although there was always a reduction in running time. Again, increasing the partitioning depth from $D = 2$ to $D = 6$ worsened the results. For instance, the percentual increase in normalized border length on 800×800 arrays in comparison with no partitioning raised from 2.02% with $D = 2$ to 4.93% with $D = 6$ (with $Q = 20$ K), although the percentual reduction in running time also raised from 12.1% to 80.0%. The reduction in running time was higher on the smaller arrays and with higher values of Q because, in these cases, the restriction on

Table 6.5: Average conflict index (ACI) of layouts produced by Greedy and Pivot Partitioning (PP) with varying partitioning depths D on random chips with probes left-most embedded in the Affymetrix deposition sequence. PP uses Greedy for placement inside final regions. In all cases, Greedy uses Q as indicated and 0-threading, and placement is followed by Sequential re- embedding with $W = 0.2\%$. Total time is reported in seconds.

		300 × 300		500 × 500		800 × 800	
Q	Alg.	ACI	Time	ACI	Time	ACI	Time
5K	Greedy	436.8630	511.3	428.7410	1 479.8	422.6277	3 870.0
	PP $D = 2$	432.8319	621.8	419.9128	1 863.0	410.8418	4 865.1
	Relative	−0.92%	+21.6%	−2.06%	+25.9%	−2.79%	+25.7%
	PP $D = 4$	441.2177	510.6	418.1961	1 724.1	403.9992	4 781.8
	Relative	+1.00%	−0.1%	−2.46%	+16.5%	−4.41%	+23.6%
	PP $D = 6$	459.5480	378.7	429.4306	1 356.5	407.4338	4 275.3
	Relative	+5.19%	−25.9%	+0.16%	−8.3%	−3.60%	+10.5%
20K	Greedy	412.5536	2 008.5	398.6096	4 555.5	389.3929	12 535.3
	PP $D = 2$	423.0404	1 184.5	400.7174	4 837.2	386.0881	13 898.2
	Relative	+2.54%	−41.0%	+0.53%	+6.2%	−0.85%	+10.9%
	PP $D = 4$	440.4754	539.6	411.0308	2 940.8	388.3189	11 656.7
	Relative	+6.77%	−73.1%	+3.12%	−35.4%	−0.28%	−7.0%
	PP $D = 6$	459.5725	378.6	428.7111	1 461.4	402.3157	6 629.7
	Relative	+11.40%	−81.2%	+7.55%	−67.9%	+3.32%	−47.1%

the number of probe candidates per spot is more significant.

With respect to CIM (Table 6.5), however, the partitioning resulted in improved layouts in some cases, especially for the larger chips. With $D = 4$ and $Q = 5$K, we observed a reduction of 4.41% in average conflict index on 800×800 arrays, although that also resulted in an increase of 23.6% in running time. On 500×500 chips, PP with $D = 6$ and Greedy with $Q = 20$K produced, in approximately the same time, a layout that was slightly better than the layout produced by Greedy with $Q = 5K$ and no partitioning (428.7111 ACI in 1 461.4 seconds versus 428.7410 ACI in 1 479.8 seconds, respectively). In some cases, the extra time needed for the partitioning (choosing pivots, comparing probes to pivots, etc.) exceeded the reduction in running time due to limiting Q and, as a result, the total time with partitioning was higher than without it. Only in one case we observed a reduction of running time combined with an improvement in solution quality: On 800×800 arrays, PP with $D = 4$ and Greedy with $Q = 20$K achieved reductions of 0.28% in ACI and 7.0% in running time when compared to Greedy alone.

6.5 Summary

We described several partitioning algorithms that are able to break the microarray layout problem into smaller sub-problems and showed that a partitioning can indeed

be used to improve solution quality and/or reduce running time. However, several aspects of the problem such as chip size, placement algorithm, type of embeddings, deposition sequence length and type of optimization (BLM or CIM), must be taken into account when choosing the partitioning algorithm and its parameters.

While Centroid-based Quadrisection and Pivot Partition offer more homogeneous improvements over all synthesis steps, 1-DP and 2-DP are able to achieve significant reduction of conflicts for a few selected masks, which can be benefical for the conflict index measure where a conflict in the middle of the probe is penalized more severely.

On chips with a 100-step cyclic deposition sequence, Pivot Partitioning outperformed previous results of CQ on larger chips because the approach of simultaneously re-embedding and assigning probes to regions better exploits the extra freedom on the probes' embeddings provided by the long deposition sequence. We believe that the comparatively worse results achieved by PP on the smaller chips with higher partitioning depths are due to the borrowing heuristic implemented in CQ that allows the placement algorithm to keep a high number of probe candidates per spot when the last sites of a quadrant are being filled.

With shorter deposition sequences, we have showed that the restriction in number of candidates per probe during placement of the last spots of a region often impacts the solution quality more significantly than the gains due to grouping similar probes together. As a result, in terms of BLM, PP failed to improve the quality of layouts produced by the Greedy placement algorithm. In terms of CIM, however, PP was able to reduce running time as well as ACI, probably because there is more room for optimization in this measure. Again, the borrowing heuristic implemented in CQ could improve the results of PP in both measures. It should be noted, however, that the effects of a partitioning on Greedy and Row-Epitaxial are mainly due to a particularity of their placement strategies; other placement algorithms such as Sliding-Window Matching (Section 3.4) are not expected to be impaired in the same way.

Chapter 7

Merging Placement and Re-embedding

In the previous chapters we have described several algorithms that deal with the microarray layout problem in the traditional way: partitioning, placement and re-embedding. The problem with the "place and re-embed" approach is that once the placement is fixed, there is usually little freedom for optimization by re-embedding the probes. Intuitively, better results should be obtained when the placement and embedding phases are considered simultaneously instead of separately. However, because of the generally high number of embeddings of each single probe, it is not easy to design algorithms that efficiently use the additional freedom and run reasonably fast in practice. In Chapter 6, we have shown how Pivot Partitioning successfully exploits this extra freedom to outperform previous partitioning algorithms.

In this chapter, we describe the first placement algorithm that simultaneously places and re-embeds the probes. Our goal was to design an algorithm that is similar to the Greedy placement algorithm (Section 3.6), so that we can make a better assessment of the gains resulting from merging the placement and re-embedding phases.

7.1 Greedy+

Greedy+ de Carvalho Jr. and Rahmann (2007) is similar to Greedy in many respects. Spots are filled in a greedy fashion, sequentially, using a user-configurable k-threading pattern. For each spot s, Greedy+ looks at Q probe candidates and chooses the one that can be placed at s with minimum cost. The main difference is that Greedy+ considers all possible embeddings of a candidate p instead of only p's given embedding. This is done by temporarily placing p at the spot s and using OSPE (Section 5.1) to compute p's optimal embedding with respect to the already-filled neighbors of s. (Naturally, OSPE can be used to compute the optimal embedding with respect to border length or conflict index.) Another difference is that, unlike Greedy and Row-Epitaxial, Greedy+ does not assume that an initial embedding of the probes is given.

Compared to Greedy, Greedy+ spends more time evaluating each probe candidate p for filling a spot s. While Greedy takes $O(T)$ time to compute the conflict index or the border length resulting from placing p at s, Greedy+ requires $O(\ell \cdot T)$ time since it uses OSPE (recall that ℓ is the probe length and T is the deposition sequence length). We must therefore use lower numbers Q of candidates per spot to achieve a running time comparable to Greedy.

There are three observations that significantly reduce the time spent with OSPE computations when several probe candidates are considered in succession for filling the same spot. First, we note that the U_t and $M_{i,t}$ costs of OSPE (Equations 5.1 and 5.2, respectively) need to be computed only once for a given spot s since they do not depend on the probe placed at s but rather on the probes placed at neighbors of s: U_t depends solely on the neighbors of s, whereas $M_{i,t}$ depends on the neighbors of s and on the number i of bases probe p already contains at synthesis step t (if all probes have the same length ℓ, then c and θ in Equation 5.2 are constants).

Second, once we know that a probe candidate p can be placed at the spot s with minimum cost κ, we can stop the OSPE computation for another candidate p' as soon as all values in a row of OSPE's dynamic programming matrix are greater than or equal to κ.

Finally, we note that if two probe sequences p and p' share a common prefix of length r, the first $r+1$ rows of OSPE's matrix D will be identical. Hence, if we have previously calculated the minimum cost of p, we can speed up the calculation of the minimum cost of p' by skipping the first $r+1$ rows of D. In order to fully exploit this fact, we must examine the probes in lexicographical order so that we maximize the length of the common prefix between two consecutive probe candidates. For this reason, Greedy+ uses the same technique used by Greedy: Initially, the probe sequences are sorted lexicographically and stored in a doubly-linked list. Once a probe p is selected to fill the current spot, it is removed from the list. For the next spot to be filled, Greedy+ looks at Q probes in the list around p's former position, e.g., at $\lfloor Q/2 \rfloor$ probes to the left and at $\lceil Q/2 \rceil$ probes to the right of p (the list is traversed from left to right).

7.2 Results

We first examine how the amplitude of the k-threading and the number Q of candidates per spot affect the results of Greedy+. In the case of BLM (Table 7.1), the best results were always achieved with surprisingly high values of k (this is in contrast to Greedy, which always produced the best results with $k = 0$). The reason is not yet clear, especially because only conflicts between adjacent spots count in the border length model. It should also be noted that for a sufficiently large value of k, a "row-wise" k-threading can be seen as a "column-wise" 0-threading.

Table 7.1: Normalized border length (NBL) of layouts produced by Greedy+ on random chips with varying number Q of candidades per spot and amplitude of k-threading. Running times are reported in minutes.

		$Q = 500$		$Q = 1\,000$		$Q = 2\,000$	
Dim.	k	NBL	Time	NBL	Time	NBL	Time
300×300	0	17.9356	5.4	17.7136	10.6	17.5460	20.6
	1	18.0922	5.4	17.8988	10.5	17.7501	20.4
	2	17.9886	5.4	17.7905	10.5	17.6342	20.5
	3	17.9339	5.7	17.7406	10.5	17.5799	20.5
	4	17.8978	5.7	17.7155	11.1	17.5506	20.5
	5	17.8862	5.7	17.7013	10.6	17.5359	20.5
	6	17.8749	5.4	17.6908	10.6	17.5225	20.5
	7	17.8641	5.5	17.6807	10.6	17.5223	20.6
	8	17.8605	5.4	17.6711	10.6	17.5141	20.6
	9	17.8519	5.4	17.6685	10.6	17.5083	20.6
	10	17.8518	5.4	17.6657	10.6	17.5067	20.6
	11	17.8427	5.5	17.6705	10.6	17.5066	20.6
	12	17.8431	5.4	17.6643	10.6	17.5070	20.6
	13	17.8455	5.4	**17.6628**	10.6	**17.5021**	20.6
	14	**17.8423**	5.4	17.6629	10.6	17.5053	20.5
500×500	0	17.3240	14.9	17.0576	29.1	16.8707	57.0
	1	17.4648	14.8	17.2483	28.9	17.0761	56.5
	2	17.3372	14.9	17.1318	29.0	16.9650	56.4
	3	17.2732	14.9	17.0785	29.0	16.9135	56.5
	4	17.2371	14.9	17.0436	29.0	16.8855	56.8
	5	17.2143	14.9	17.0264	29.3	16.8676	57.2
	6	17.1990	15.0	17.0141	29.3	16.8557	57.2
	7	17.1812	15.0	17.0049	29.3	16.8420	57.2
	8	17.1774	15.0	16.9965	29.3	16.8398	57.0
	9	17.1704	15.0	16.9921	29.4	16.8346	57.3
	10	17.1666	15.8	16.9876	29.2	16.8332	59.7
	11	17.1629	15.0	16.9814	29.1	16.8294	56.8
	12	17.1594	14.9	16.9821	29.3	16.8280	56.7
	13	17.1549	15.8	16.9767	29.1	**16.8240**	56.8
	14	**17.1503**	14.9	**16.9737**	29.1	16.8261	56.8
800×800	0	16.7983	38.0	16.4944	73.8	16.2640	144.4
	1	16.8849	37.7	16.6615	73.3	16.4780	143.3
	2	16.7420	37.8	16.5377	73.5	16.3626	143.6
	3	16.6693	37.9	16.4775	73.9	16.3070	143.9
	4	16.6266	38.0	16.4375	73.8	16.2707	144.2
	5	16.5938	38.1	16.4096	74.2	16.2497	145.1
	6	16.5700	38.2	16.3919	74.3	16.2334	145.2
	7	16.5543	38.2	16.3801	74.6	16.2237	145.2
	8	16.5435	38.1	16.3691	74.5	16.2171	145.3
	9	16.5379	38.2	16.3646	74.7	16.2115	145.8
	10	16.5297	38.0	16.3586	74.0	16.2094	144.5
	11	16.5229	38.0	16.3539	74.0	16.2039	144.5
	12	16.5210	38.2	16.3518	74.1	16.2022	144.6
	13	16.5194	38.1	16.3474	74.1	16.1971	144.7
	14	**16.5118**	38.0	**16.3456**	74.1	**16.1968**	144.8

Table 7.2: Average conflict index (ACI) of layouts produced by Greedy+ on random chips with varying number Q of candidades per spot and k-threading's amplitude. Running times are reported in minutes.

		$Q = 500$		$Q = 1\,000$		$Q = 2\,000$	
Dim.	k	ACI	Time	ACI	Time	ACI	Time
300×300	0	**462.3882**	5.8	**443.3786**	10.5	**425.9132**	19.8
	1	468.6485	5.8	449.1931	10.6	431.1021	19.9
	2	472.3753	5.8	452.5054	10.6	434.1209	19.9
	3	474.3210	5.8	454.6870	10.6	436.2880	20.0
	4	474.2031	5.8	454.6782	10.6	436.2529	19.9
500×500	0	**457.3329**	15.8	**437.3920**	28.8	**419.2114**	54.2
	1	463.6259	16.0	443.7018	30.4	424.5009	54.7
	2	467.3461	15.9	447.5021	29.0	428.3882	54.8
	3	469.2554	16.6	449.4136	29.1	430.4992	55.0
	4	468.9371	16.0	449.5197	29.1	430.4662	58.0
800×800	0	**451.8074**	40.0	**431.8977**	73.0	**413.3451**	144.3
	1	458.1598	40.3	437.8440	73.5	418.9562	138.4
	2	461.6418	40.3	441.6484	73.3	423.0075	145.9
	3	463.5349	40.3	443.7868	73.6	425.2302	138.9
	4	463.1225	40.3	443.7802	73.7	425.3695	139.0

With BLM, increasing the amplitude from $k = 0$ to $k = 1$ always worsened the results. Increasing it further, however, improved the layouts and eventually resulted in less conflicts than with $k = 0$ up to a point when it started to make little difference. The greatest difference between the worst and the best layouts due to the amplitude k was at most 2.26% (from 16.5118 with $k = 14$ to 16.8849 with $k = 1$ on 800×800 chips and $Q = 500$). In case of CIM (Table 7.2), the best results were always achieved with $k = 0$, and increasing it up to $k = 3$ always resulted in more conflicts, although increasing it to $k = 4$ often resulted in slightly better layouts than with $k = 3$.

In both cases, doubling the number Q of candidates per spot roughly doubled the running time. In contrast with Greedy, Greedy+ requires approximately the same time with CIM and BLM, sometimes being even slightly faster with the former. This can be explained as follows. The major difference the quality measure makes for OSPE, in terms of running time, is when the U_t and $M_{i,t}$ costs of OSPE are computed. While for BLM at most four neighbors of a spot s need to be examined, for CIM we must look at up to 48 neighbors of s. However, since the U_t and $M_{i,t}$ costs are computed only once for a spot s and are reused for each of the Q candidate probes, the greater the number Q, the less impact the quality measure makes in total running time. The fact that Greedy+ is sometimes slightly faster with CIM than with BLM could be because, with the former, it more quickly finds a probe candidate with a low minimum cost κ that allows it to stop computing the cost of other candidates sooner (when all entries in a row of OSPE's matrix are greater than κ).

We now compare the results obtained by Greedy and Greedy+ when both algorithms

Table 7.3: Normalized border length (NBL) of layouts produced by Greedy and Greedy+ on random chips with the number Q of candidades per spot of Greedy+ set in such a way that it does not exceed the time spent by Greedy. Total time including placement and re-embedding is reported in minutes. Both algorithms use 0-threading and are followed by two passes of re-embedding optimization with Sequential. The relative difference in NBL and time between the two approaches is shown in percentage.

	Greedy and Sequential			Greedy+ and Sequential			Relative	
Dim.	Q	NBL	Time	Q	NBL	Time	NBL	Time
300×300	10 000	18.0900	6.2	300	**17.9807**	4.2	−0.60%	−31.21%
	20 000	17.9725	12.1	700	**17.6746**	9.2	−1.66%	−23.85%
500×500	10 000	17.3809	20.8	450	**17.2216**	16.0	−0.92%	−23.30%
	20 000	17.2779	41.9	950	**16.9382**	30.4	−1.97%	−27.42%
800×800	10 000	16.7143	57.9	500	**16.6549**	41.7	−0.36%	−28.00%
	20 000	16.6259	121.6	1 130	**16.3175**	97.7	−1.85%	−19.68%

Table 7.4: Average conflict index (ACI) of layouts produced by Greedy and Greedy+ (with 0-threading) on random chips in approximately the same amount of time (total time in minutes including two passes of Sequential re-embedding optimization). The relative difference in ACI between the two approaches is shown in percentage.

	Greedy and Sequential			Greedy+ and Sequential			
Dim.	Q	ACI	Time	Q	ACI	Time	Relative
300×300	10 000	**423.1330**	13.9	1 070	438.4015	14.0	+3.61%
	20 000	**412.5536**	24.1	2 180	420.8863	24.2	+2.02%
	80 000	402.4365	54.3	5 500	**401.7005**	54.0	−0.18%
500×500	10 000	**412.5468**	43.2	1 225	428.5082	43.7	+3.87%
	20 000	**398.6096**	77.0	2 580	409.6446	76.9	+2.77%
	140 000	375.5428	352.2	13 500	**374.9914**	351.9	−0.15%
800×800	10 000	**405.3133**	113.9	1 315	421.2380	113.7	+3.93%
	20 000	**389.3929**	207.9	2 790	401.7969	208.5	+3.19%
	300 000	350.8412	2 056.7	32 000	**350.6951**	2 050.8	−0.04%

are given the same amount of time (the parameter Q is chosen differently for both algorithms so that the running time is approximately comparable). To be fair, since Greedy is a traditional placement algorithm that does not change the embeddings of the probes, we need to compare the layouts obtained by both algorithms after a re-embedding phase. For this task we use the Sequential algorithm (Section 5.4) performing two passes of re-embedding optimization. For this experiment we use probes of length $\ell = 25$ left-most embedded in the standard Affymetrix deposition sequence.

Table 7.3 compares both algorithms in terms of border length minimization. In all cases, Greedy+ produced better layouts than Greedy in the same amount of time (or less) while looking at fewer probe candidates. For instance, on 800×800 chips Greedy+ with $Q = 1\,130$ produced layouts with 1.85% less border conflicts than Greedy with $Q = 20\,000$ in 19.68% less time, on average.

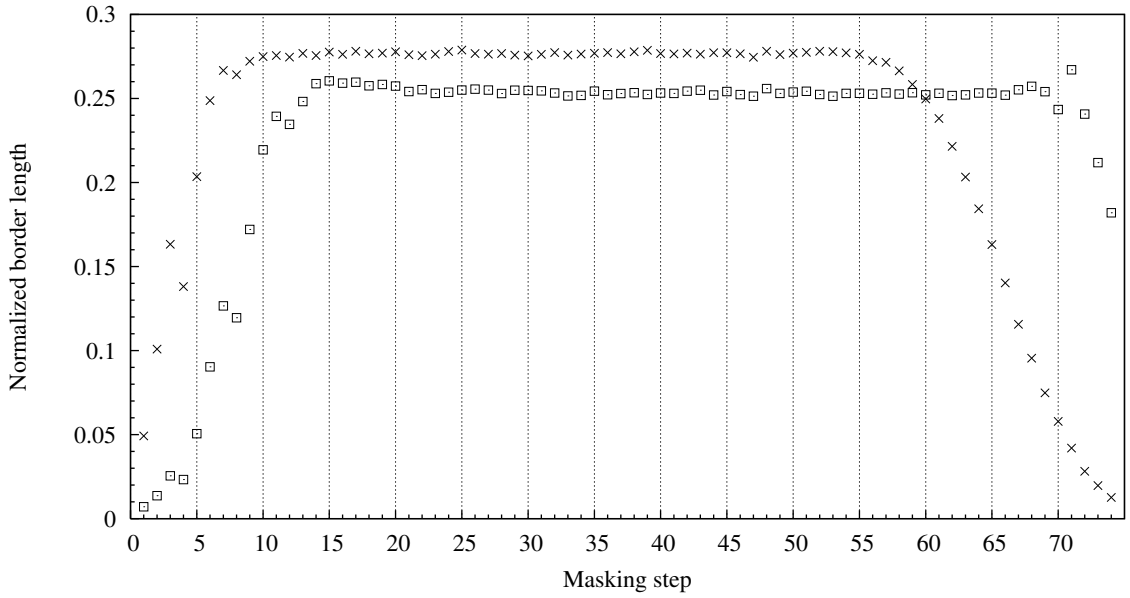


Figure 7.1: Normalized border length per masking step of layouts produced by Greedy with $Q = 20\,000$ (×) and Greedy+ with $Q = 950$ (⊡) for a 500×500 chip with border length minimization. Both algorithms used 0-threading and were followed by two passes of re-embedding optimization with Sequential.

In terms of CIM (Table 7.4), Greedy is not so easily outperformed by Greedy+. With $Q = 10\,000$ and $Q = 20\,000$ Greedy produced better layouts than Greedy+ in approximately the same time. For instance, on 800×800 chips, Greedy+ with $Q = 2\,790$ produced layouts with 3.19% more conflicts than Greedy with $Q = 20\,000$. However, Greedy+ has an advantage over Greedy since it needs to examine fewer candidates to achieve similar results and, for sufficiently large values of Q, it is usually possible to achieve better results with Greedy+ in the same amount of time. For instance, on 300×300 chips, Greedy+ with $Q = 13\,500$ produced layouts with only 0.18% less conflicts than Greedy with $Q = 80\,000$. After this point, however, the difference in ACI between Greedy and Greedy+ tends to increase (data not shown). We also observed that the larger the chip, the less advantage Greedy+ has over Greedy. On 500×500 chips, Greedy+ starts to outperform Greedy when $Q = 13\,500$ (with running times in the order of 6 hours), approximately, and on 800×800 chips around $Q = 32\,000$ (with more than 34 hours of running time per array).

One advantage of Greedy+ is that, unlike Greedy, it is not influenced by the initial embeddings of the probes. Figure 7.1 shows the normalized border length of layouts produced by Greedy and Greedy+ with border length minimization for a selected 500×500 chip with equivalent numbers Q of candidates per spot (in accordance with Table 7.3). Because the probes were initially left-most embedded, Greedy produced a layout in which the border conflicts are concentrated between steps 7 and 58; start-

Table 7.5: Normalized border length (NBL) of layouts produced by Row-Epitaxial and Greedy+ (both with 0-threading) on random chips in approximately the same amount of time. (total time in minutes including two passes of Sequential re-embedding optimization). The relative difference in NBL and time between the two approaches is shown in percentage.

	Row-Epitaxial and Sequential			Greedy+ and Sequential			Relative	
Dim.	Q	NBL	Time	Q	NBL	Time	NBL	Time
300×300	10 000	18.0524	4.3	300	**17.9807**	4.2	−0.40%	−1.24%
	20 000	17.9430	9.5	700	**17.6746**	9.2	−1.50%	−2.85%
500×500	10 000	17.3584	16.0	450	**17.2216**	16.0	−0.79%	−0.40%
	20 000	17.2502	34.7	950	**16.9382**	30.4	−1.81%	−12.51%
800×800	10 000	16.7176	45.6	500	**16.6549**	41.7	−0.38%	−8.51%
	20 000	16.6012	100.1	1 130	**16.3175**	97.7	−1.71%	−2.41%

ing on step 59, the normalized border length drops steadily as the embeddings reach their last productive steps. In contrast, Greedy+ produces a layout with a more uniform distribution of conflicts in the final synthesis steps. In both cases the first masks have relatively few conflicts as a result of lexicographically sorting the probes. A representation of selected masks for the layout produced by Greedy+ is shown in Figure 7.2. Layers of masked and unmasked regions in masks M_1 to M_9 are similar to the ones shown in Figure 3.7, although the masks produced by Greedy are "noisier". The normalized border length of these layouts are 17.3182 (Greedy) and 16.9451 (Greedy+).

Finally, we also compare Greedy+ with Row-Epitaxial (Section 3.5), which, in terms of border length minimization, achieves results comparable to Greedy in less time. Table 7.5 shows that Greedy+ also outperforms Row-Epitaxial in the same amount of time (or less). The larger values of Q are used, the greater is the advantage of Greedy+. According to the results of Table 7.1, the difference in NBL between Greedy+ and Row-Epitaxial could be even greater if the former used higher k-threading amplitudes.

7.3 Summary

We have presented a new placement algorithm, called Greedy+, that for the first time places and re-embeds the probes simultaneously. Our results have shown that Greedy+ outperforms the previously best placement algorithms — Row-Epitaxial for border length minimization and Greedy for conflict index minimization. In terms of CIM, Greedy produces better results when time is limited but, otherwise, Greedy+ should be the placement algorithm of choice. In fact, Greedy+ achieves similar results to Greedy by examining fewer probe candidates per spot and, for this reason, it has the potential for producing better layouts.

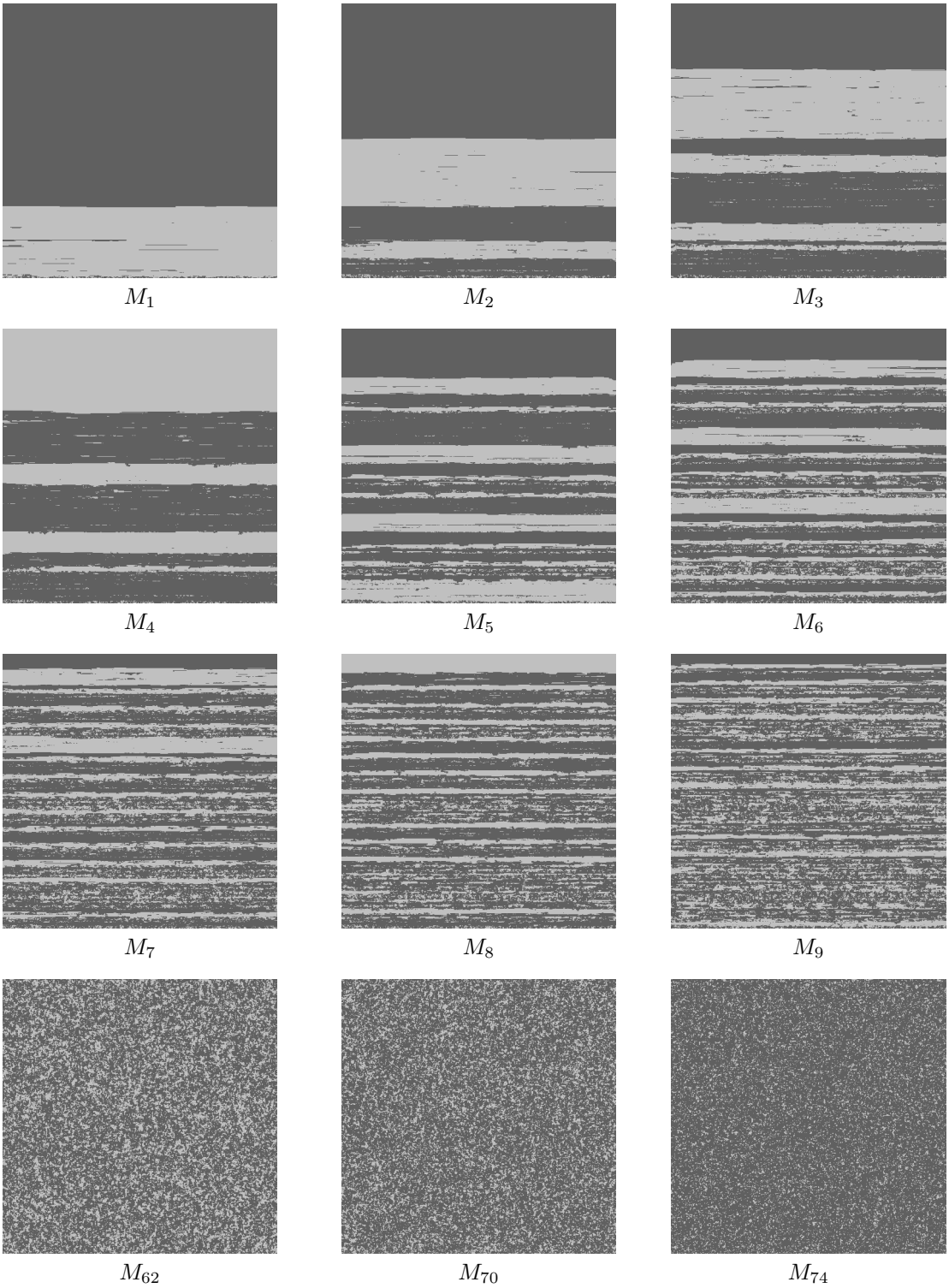


Figure 7.2: Selected masks generated by Greedy+ with border length minimization for a 500×500 chip with 25-mer probes embedded in the standard Affymetrix deposition sequence. Unmasked (masked) spots are represented by light (dark) dots.

7.3.1 Future work

The fact that Greedy+ does not outperform Greedy so easily in terms of CIM as it does in terms of BLM could be explained by the fact that probes are sorted lexicographically, which increases the chances of finding candidates that have similar prefixes but not good "matches" for the middle part of the embeddings. Greedy has an advantage since it looks at more candidates in the lexicographically sorted list of probes. One possibility that could improve the results of Greedy+ is to sort the list of probes with an emphasis on the middle bases. Although this is technically possible, with our current implementation of OSPE it would result in an increase in running time because consecutive candidates would then be unlikely to have a common prefix, requiring the dynamic programming matrix to be entirely re-computed for each probe considered. We leave as an open problem the question of finding an ordering of the probes with an emphasis on the middle bases and an implementation of OSPE in such a way that consective candidates can be examined quickly by skipping the computation of identical rows of the matrix.

Chapter 8

Analysis of Affymetrix Microarrays

Affymetrix GeneChip arrays are considered the industry standard in terms of high-density oligonucleotide microarrays. In this chapter, we analyze the layout of several GeneChip arrays with respect to the quality measures defined in Chapter 2, i.e., border length and conflict index. We then use some of the algorithms presented in previous chapters to create alternative layouts for two commercially available microarrays.

8.1 Introduction

We obtained the specification of several GeneChip arrays containing the list of probe sequences and their positions on the chip from Affymetrix's web site[1]. As discussed below, we have to make a few assumptions because some details such as the deposition sequence used to synthesize the probes, the probe embeddings, and the contents of "special" spots are not publicly available.

Some of the special spots are used to help the mechanical alignment of the chip with the scanner that captures the image with the hybridization signals. Others contain *quality control probes* used to detect failures during the production of the chip (Affymetrix, Inc., 2002; Hubbell and Pevzner, 1999). Not knowing the contents of these special spots did not interfere with our analysis because, in all arrays we examined, they amount to at most 1.22% of the total number of spots.

What could interfere with our analysis is the fact that some arrays have a significant number of empty spots (as much as 11.94% on the Chicken Genome array). The physical location of some empty spots suggest that they might be used as "spacers" to separate regions of the chip. Others might be empty simply because the number of spots exceeds the number of probes. A high number of empty spots result in a relatively low normalized border length (as defined in Section 2.4) since we divide the total number of border conflicts by the number of internal borders of the chip (an empty spot contributes to the number of internal borders but obviously not to the

[1] `http://www.affymetrix.com/support/technical/byproduct.affx?cat=arrays`

N	TGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATG
ε_p	G AT TG A G A A C C GCA G T A C G A C C C G T
$\varepsilon_{\bar{p}}$	G AT TG A G A A C C G G A G T A C G A C C G T
	++ +++ ++ ++ +++ +++ ++ ++ +
ε_p	G AT TG A G A A C C GC A G T A C G A C C G T
$\varepsilon_{\bar{p}}$	G AT TG A G A A C C G G A G T A C G A C C G T
	+ +

Figure 8.1: Left-most (above) and pair-wise left-most (below) embeddings ε_p and $\varepsilon_{\bar{p}}$ of perfect match (PM) and mismatch (MM) probes $p =$ GATTGAGAACCGCAGTACGACCCGT and $\bar{p} =$ GATTGAGAACCGGAGTACGACCCGT, respectively, in the standard Affymetrix deposition sequence $N = (\texttt{TGCA})^{18}\texttt{TG}$. Conflicts between the embeddings are highlighted with plus signs (+) in the corresponding synthesis steps.

number of border conflicts). Thus, to better compare chips with different amounts of empty spots we also use the *average number of border conflicts per probe*, defined as the total border length divided by the number of probes. As we shall see, an array with many empty spots might still have an advantage depending on how the empty spots are distributed on the chip.

It has been reported that a fixed 74-step deposition sequence is used by Affymetrix (Kahng et al., 2004). An analysis with the algorithms presented in Chapter 9 revealed that all GeneChip arrays, regardless of their size, can be synthesized in $N = (\texttt{TGCA})^{18}\texttt{TG}$, i.e., 18.5 cycles of TGCA, and that a shorter deposition sequence is indeed unlikely. This suggests that only sub-sequences of this particular deposition sequence can be used as probes on Affymetrix chips. In principle, this should not be a problem as this sequence covers about 98.45% of all 25-mers (Rahmann, 2006).

Probes of GeneChip arrays always appear in pairs: the perfect match (PM), which perfectly matches its target sequence, and the mismatch (MM) probe, which is used to quantify cross-hybridizations and unpredictable background signal variations (Affymetrix, Inc., 2001). The MM probe is a copy of the PM probe except for the middle base (position 13 of the 25-mer), which is exchanged with its Watson-Crick complement. The layout of GeneChip arrays alternate rows of PM probes with rows of MM probes in such a way that probes of a pair are always adjacent on the chip. Moreover, PM and MM probes are *pair-wise left-most embedded* (Kahng et al., 2004). Informally, a pair-wise left-most embedding is obtained from left-most embeddings by shifting the second half of one embedding to the right until the two embeddings are "aligned" in the synthesis steps that follow the mismatched middle bases (Figure 8.1). This approach reduces border conflicts between the probes of a pair, although it leaves a conflict in the steps that add the middle bases.

The fact that probes must appear in pairs restricts even more which sequences can be used as probes on GeneChip arrays because both PM and MM probes must "fit"

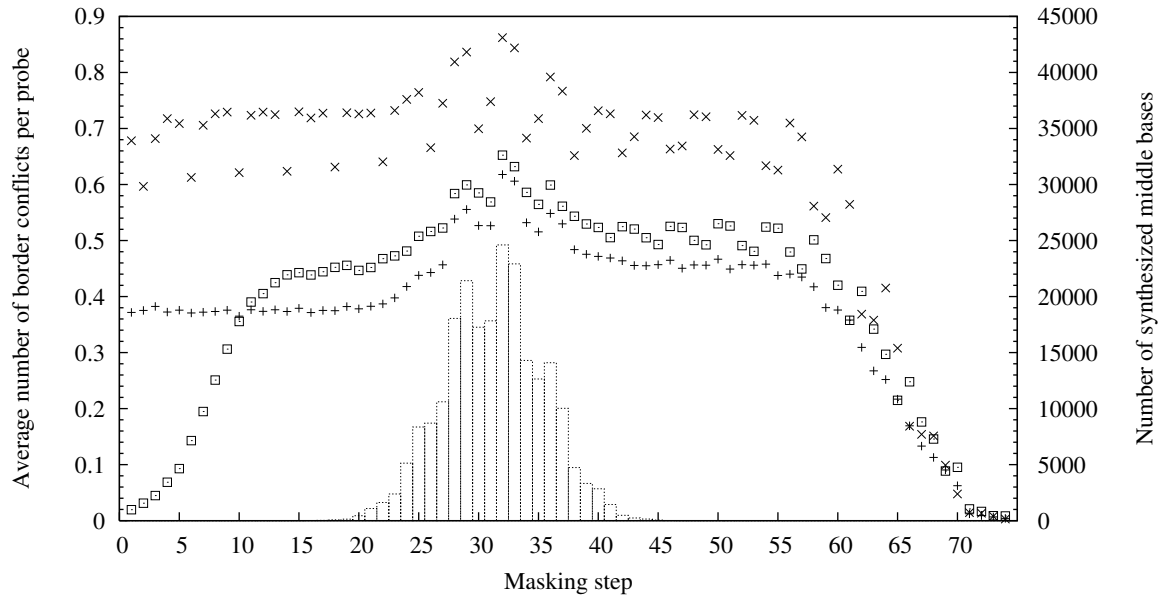


Figure 8.2: Average number of border conflicts per probe per masking step (on the left y-axis) of three GeneChip arrays, assuming pair-wise left-most embeddings: Yeast Genome S98 (×), Human Genome U95A2 (+), and E. coli Genome 2.0 (⊡). The number of middle bases synthesized at each step on the E. coli Genome 2.0 is shown in boxes (on the right y-axis).

in the deposition sequence. For example, although $p = \texttt{CGTAGGTACGTTATAAGTCACTAAA}$ has an embeddeding in $N = (\texttt{TGCA})^{18}\texttt{TG}$, it cannot be used as a probe because its corresponding mismatch probe $\bar{p} = \texttt{CGTAGGTACGTTTTAAGTCACTAAA}$ cannot be embedded in N, as shown below.

N	TGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATG
ε_p	C G T A G G T A C G T T AT A A G T CA C T A A A
$\varepsilon_{\bar{p}}$	C G T A G G T A C G T T T T A A G T CA C T A A A

8.2 Layout Analysis

Figure 8.2 shows the average number border conflicts per probe per masking step of three GeneChip arrays. We assume that the probes are pair-wise left-most embedded in $N = (\texttt{TGCA})^{18}\texttt{TG}$, and we consider all spots whose contents are not available as empty spots. In all chips we analyzed, the number of border conflicts are higher in the steps that add the middle bases, a result of placing PM and MM probes in adjacent spots. The Yeast Genome S98 array has the worst layout in terms of border conflicts and most of the earlier GeneChip arrays such as the E. coli Antisense Genome have similar levels of conflicts. The layout of the Human Genome U95A2 array has signficantly less

Table 8.1: Average number of border conflicts per probe (ABC), normalized border length (NBL) and average conflict index (ACI) of selected GeneChip arrays (assuming pair-wise left-most embeddings). The dimension of the chip, the percentage of spots with unknown content and the percentage of empty spots are also shown.

GeneChip Array	Dimension	Unknown	Empty	ABC	NBL	ACI
Yeast Genome S98	534×534	1.22%	1.70%	44.8168	21.7945	669.0663
E. coli Antisense Genome	544×544	1.17%	3.12%	43.3345	20.7772	663.7353
Human Genome U95A2	640×640	0.96%	1.83%	28.2489	13.7517	510.3418
E. coli Genome 2.0	478×478	1.08%	0.46%	29.2038	14.4079	550.2014
Chicken Genome	984×984	0.46%	11.94%	28.2087	12.3680	540.5022
Wheat Genome	$1\,164 \times 1\,164$	0.38%	0.08%	27.6569	13.7771	539.9632

border conflicts than the Yeast array, suggesting that it was designed with a better placement strategy. The curve of the E. coli Genome 2.0 array, with very low levels of conflicts in the first 10 masks, is typical of the latest generation of GeneChip arrays, including the Chicken Genome and the Wheat Genome (one of the largest GeneChip arrays currently available with $1\,164 \times 1\,164$ spots), and suggest yet another placement strategy.

Figure 8.3 shows a representation of selected masks for the E. coli Genome 2.0. The low levels of conflicts in the first synthesis steps are a result of the pattern of masked and unmasked layers that can be seen in masks M_1 to M_9. This pattern is similar to the ones produced by Greedy (Figure 3.7) and Greedy+ (Figure 7.2). A more careful examination, however, reveals that the layers are arranged in a way that resembles the Gray-code–based arrangement employed by 1-Dimensional Partitioning (Figure 6.1). This does not necessarily mean that the layout was produced by such a partitioning. In fact, a similar effect could be produced by a placement algorithm such as Greedy or Greedy+ if the probes were ordered in such a way that a prefix of their binary embeddings formed an approximation of a Gray code.

Table 8.1 confirms that the layout of the Human Genome U95A2 array is the best in terms of normalized border length and average conflict index. This, however, has more to do with empty spots than with the placement strategy as this chip has about 1.83% of empty spots that are evenly distributed on the chip surface (Figure 8.4, left). In contrast, the Chicken Genome array has an exceptionally high percentage of empty spots (11.94%) that contribute to lower the normalized border length but that does not result in a lower average number of border conflicts per probe in comparison with the Human Genome array because the empty spots are concentrated in the lower part of the chip (Figure 8.4, right).

GeneChip arrays exhibit relatively low levels of border conflicts when compared to layouts produced by the best algorithms for random arrays of similar dimensions. This can be explained by the fact that each probe has a nearly identical copy next to it. Not surprisingly, these arrays have relatively high average conflict indices when

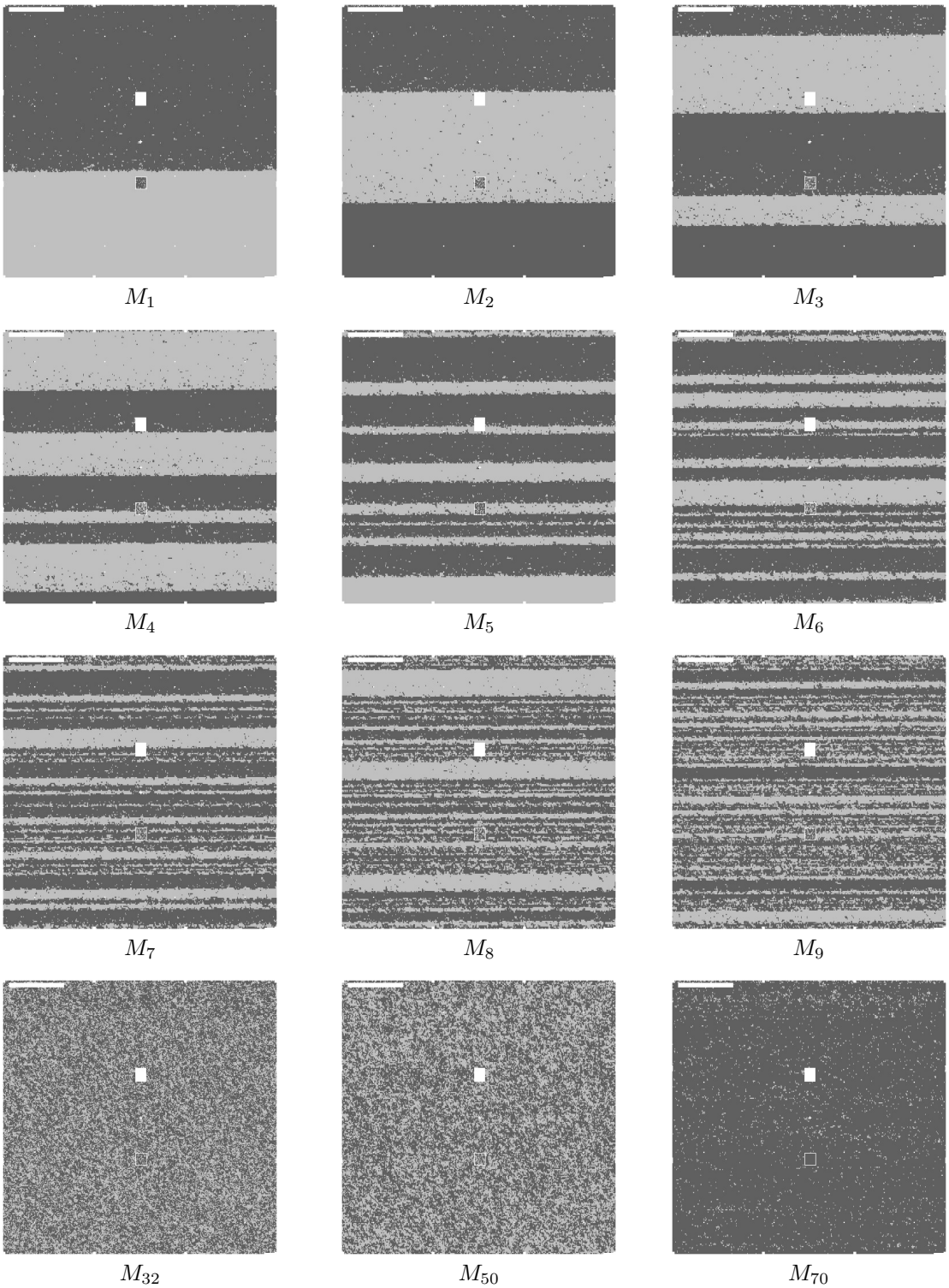
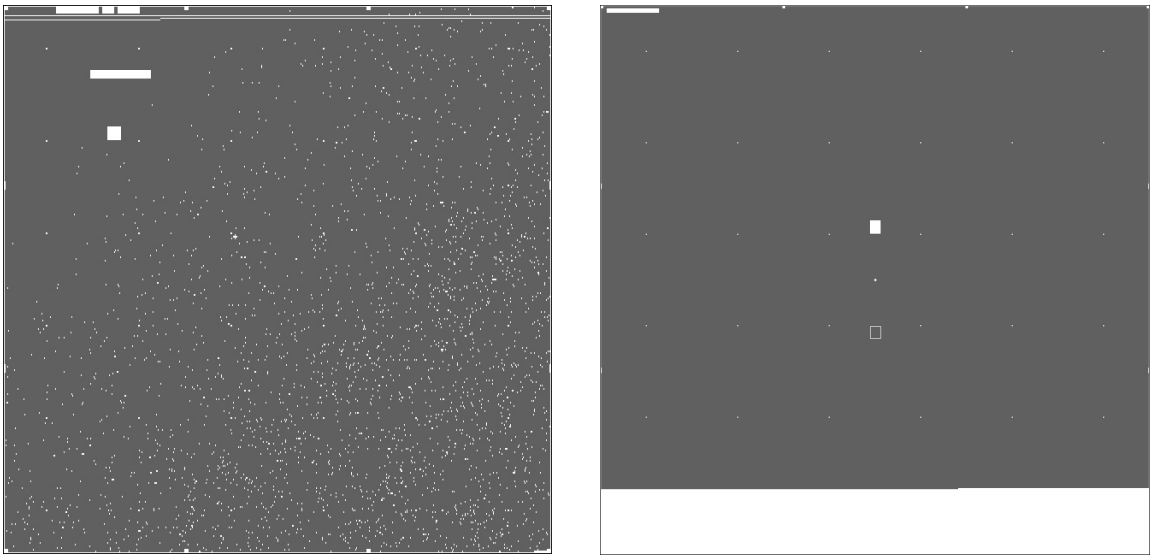


Figure 8.3: Selected masks of Affymetrix's E. coli Genome 2.0 GeneChip array, assuming pair-wise left-most embeddings. Unmasked (masked) spots are represented by light (dark) dots. White regions represent spots whose contents are not publicly available.



Human Genome U95A2 (640 × 640) Chicken Genome (984 × 984)

Figure 8.4: Distribution of empty spots on two GeneChip arrays. Chip dimensions are indicated in parenthesis (images were scaled differently). Non-empty spots are represented by dark dots. White dots represent empty spots or spots whose contents are not publicly available.

compared to random arrays because the conflicts are concentrated on the synthesis steps that add the middle bases.

8.3 Alternative Layouts

We used Greedy+ (Chapter 7) and Sequential re-embedding (Section 5.4) to create alternative layouts for two of the latest generation of GeneChip arrays: E. coli Genome 2.0 and Wheat Genome. Greedy+ was modified to avoid placing probes on special spots or empty spots that we believe might have a function on the chip.

For each chip we run the algorithms with border length as well as conflict index minimization. The main difference between our layouts and the original ones is that we do not require the arrays to alternate rows of PM and MM probes; hence, probes of a pair are not necessarily placed on adjacent spots. This is especially helpful for conflict index minimization since it avoids conflicts in the middle bases. With border length minimization, we observed that Greedy+ placed between 90.70% and 95.16% of the PM probes adjacent to their corresponding MM probes. With conflict index minimization, this rate dropped to between 12.89% and 21.25%.

Figure 8.5 shows the normalized border length per masking step of the layout produced by Greedy+ and Sequential for the E. coli Genome 2.0 array in comparison with the

Table 8.2: Normalized border length (NBL) and average conflict index (ACI) of several layouts for the E. coli Genome 2.0 and Wheat Genome GeneChip arrays. Greedy+ and Sequential run with border length and conflict index minimization (BLM and CIM, respectively) as indicated. Greedy+ used k-threading with $k = 5$ for BLM and $k = 0$ for CIM. Running times are reported in minutes and include placement (Greedy+) and 2 passes of re-embedding optimization with Sequential.

Array	Layout	NBL	ACI	Time
E. coli 2.0	Affymetrix with pair-wise left-most	14.4079	550.2014	—
	Affymetrix after "pair-aware" Sequential (BLM)	13.5005	541.0954	—
	Greedy+ with $Q = 2$K and Sequential (BLM)	13.3774	529.8129	46.9
	Greedy+ with $Q = 10$K and Sequential (BLM)	13.2406	515.5917	218.3
	Greedy+ with $Q = 2$K and Sequential (CIM)	17.6935	394.9905	54.9
	Greedy+ with $Q = 10$K and Sequential (CIM)	17.5575	361.4418	225.7
Wheat	Affymetrix with pair-wise left-most	13.7771	539.9632	—
	Affymetrix after "pair-aware" Sequential (BLM)	12.9151	531.2692	—
	Greedy+ with $Q = 2$K and Sequential (BLM)	12.7622	519.0869	279.2
	Greedy+ with $Q = 5$K and Sequential (BLM)	12.6670	511.7193	676.0
	Greedy+ with $Q = 2$K and Sequential (CIM)	17.1047	387.8430	322.7
	Greedy+ with $Q = 5$K and Sequential (CIM)	17.1144	366.6045	704.7

original Affymetrix layout. For comparison, we also show the result of running a "pair-aware" version of Sequential on the original layout (this version ensures that the embeddings of PM-MM pairs remain pair-wise "aligned"). The normalized border length and average conflict indices of these layouts are shown on Table 8.2, together with several layouts for the Wheat Genome array. Greedy+ with $Q = 10$K produced a layout with 8.10% less border conflicts than the original layout for the E. coli array (13.2406 versus 14.4079) in 218.3 minutes. With $Q = 2$K, this difference was 7.15%, although that required only 46.9 minutes. For the Wheat array, Greedy+ with $Q = 2$K generated a layout with 7.36% less border conflicts than the original layout (12.7622 versus 13.3771). It is not fair to compare the layouts in terms of CIM since the original layouts were probably designed to minimize border conflicts (and not conflict indices). Nevertheless, the results produced by Greedy+ and Sequential are comparable to the results on random chips presented on Chapter 7.

8.4 Summary

We have analyzed the layout of several commercial microarrays with respect to border length and conflict index. It is clear that placing perfect match (PM) and mismatch (MM) probes on adjacent spots reduces the incidence of border conflicts. However, this also has the disadvantage of concentrating the conflicts on the synthesis steps that add the middle bases, precisely where the probes are most likely to be damaged.

We have also showed that two algorithms presented in earlier chapters, Greedy+ and

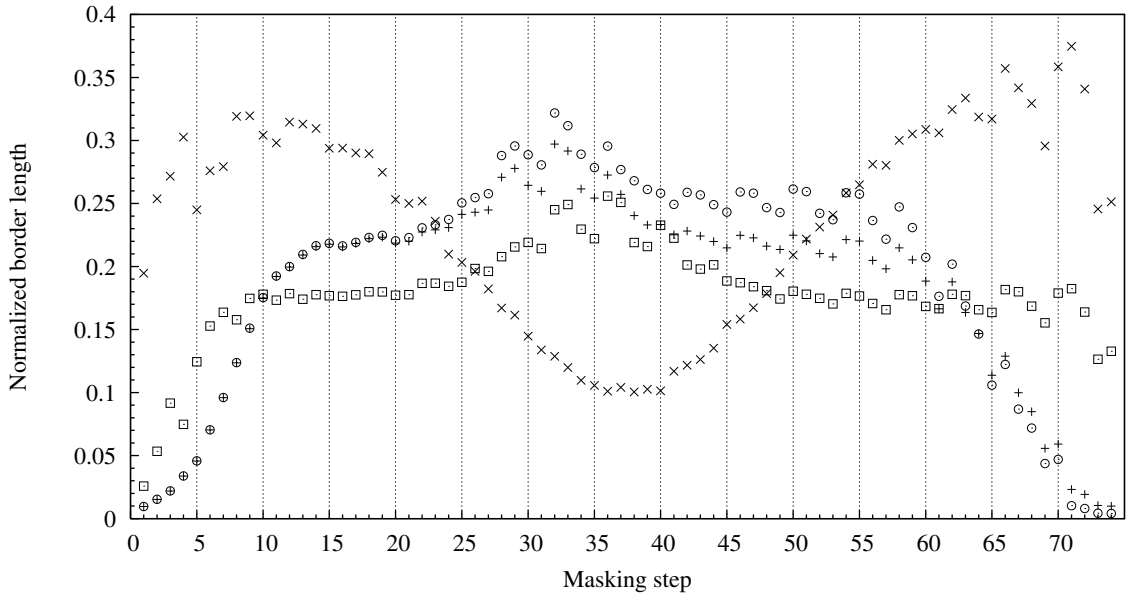


Figure 8.5: Normalized border length per masking step of several layouts for the E. coli Genome 2.0 GeneChip array: original Affymetrix layout with pair-wise left-most embeddings (⊙), original Affymetrix layout after running two passes of a "pair-aware" version of Sequential re-embedding (+), layout produced by Greedy+ with $Q = 10\text{K}$ and Sequential with border length minimization (⊡), and layout produced by Greedy+ with $Q = 10\text{K}$ and Sequential with conflict index minimization (×).

Sequential re-embedding, performed well on real microarrays, including one of the largest GeneChip arrays available, producing layouts with up to 8.10% less border conflicts than the original layouts in reasonable time, and layouts with average conflict index comparable to results on random arrays. In general, we believe that the quality of currently available GeneChip arrays can be significantly improved with respect to the problem of unintended illumination.

Chapter 9

The Shortest Deposition Sequence Problem

Several interesting computational problems emerge in the design and production of DNA microarrays. The shortest common supersequence (SCS) problem is one such a problem, which arises during the synthesis of probes on the array. It is an classical problem in computer science that has already been proved to be NP-complete. The aim of this report is to analyze the feasibility of finding shortest common supersequences for the production of DNA microarrays. Several different strategies are considered. Because of time and space constraints, we propose a branch-and-bound depth-first search. This strategy relies on computing upper and lower bounds on the length of the SCS to prune the search space. A computer program was implemented to analyze its performance in practice. The implementation is described in detail, along with several attempts to reduce its running time. Finally, we describe some experiments and conclude with an evaluation of the implementation as well as the feasibility of such an approach.

9.1 Introduction

In the arrays manufactured by Affymetrix, the probe sequences are synthesized on the chip, in parallel, nucleotide by nucleotide. The process consists of a series of steps. In each step the same nucleotide is appended to all probes located on selected spots. This selection occurs by exposure to light, with a mask that is placed over the chip to ensure that only the appropriate spots are activated to receive the nucleotide. After that, a solution containing the nucleotides is flushed on the chip and allowed to hybridize.

The sequence of nucleotides used in the synthesis process is called the *deposition sequence*. Clearly, each probe is a (non-contiguous) subsequence of the deposition sequence, which is, in turn, a supersequence of the set of probes. It is common to use a deposition sequence that consists of repetitions of `ACGT`. This sequence is used because of its simplicity and relative effectiveness Rahmann (2004).

Number of sequences:	$10,000 \leq n \leq 1,000,000$
Length of sequences:	$20 \leq m \leq 30$
Alphabet:	$\Omega = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$
Alphabet size:	$z = \lvert\Omega\rvert = 4$

Table 9.1: Variable constraints in the DNA chip production setting.

Ideally, the deposition sequence should be as short as possible so that the number of steps is reduced and, consequently, the manufacturing time is shortened. Moreover, the masking process is subject to errors, i.e., there is a risk of having the light shining over a spot that was not supposed to be activated. Hence, if the number of steps is reduced, the chance of unwanted spot activation is also reduced, and the overall quality of the chip is improved.

The issue of accidental activation can also be diminished by carefully arranging the probes on the chip. It is evident that the spots more likely to be affected by stray light are those which lie immediately next to an unmasked spot. More precisely, the chances are higher on the *borders* shared with an unmasked spot. For this reason, the risk of errors can be reduced by minimizing the number of borders shared by masked and unmasked spots in each step of the synthesis process.

This report, however, will concentrate on minimizing the length of the deposition sequence. More precisely, we are interested in finding the shortest supersequence of a given set of sequences such that all probes of a microarray chip can be synthesized. This is, of course, a well-known problem in the computer science literature and is usually referred as the shortest common supersequence (SCS) problem. It was shown to be NP-complete for strings over an alphabet of size greater than or equal to 2. Räihä and Ukkonen (1981). For this reason, several heuristics have been devised to compute approximate solutions more efficiently — see Fraser (1995) for a survey. But, to this day, finding an exact solution seems to be limited to small sets of sequences and reduced alphabet sizes.

The objective of this report is, therefore, to analyze the feasibility of computing a solution to the SCS problem in the context of the production of DNA microarrays.

Formally we have a set of n sequences $\mathcal{P} = \{p_1, p_2, p_3, \ldots, p_n\}$, drawn from the alphabet Ω, whose size is $z = |\Omega|$. The length of any sequence p_i is $m = |p_i|$ (for simplicity, all sequences are assumed to have the same length, but this assumption is not necessary). Given this input, we want to find s, the shortest common supersequence for the set $\mathcal{P}$. The DNA chip production setting constrains these variables to the values shown in table 9.1.

9.2 Alternatives

Several approaches may be tried in order to find an exact solution to the shortest supersequence problem. Naively, one can generate all possible sequences with length k, checking whether each of them is a viable supersequence. The initial value of k can be set to m (the length of the sequences) since a supersequence must be, at least, m characters long. If no supersequence is found with k characters, the value of k is increased and all sequences of this new length are generated and examined. The value of k denotes the length of the shortest common supersequence as soon as a supersequence is found.

Clearly, m is not a good initial value for k. The shortest supersequence of a typical DNA chip can be expected to be between 55 and 95 characters long (see Rahmann (2003)). Fortunately, reasonably good lower bounds on the length of the SCS can easily be computed and the search can start with a higher value for k.

With the methods described in section 9.4, one can expect to find a lower bound of around 60 characters for a typical set of $n = 10,000$ probes of length $m = 25$. Starting from $k = 60$ (and not $k = 25$) alone saves a significant amount of time. But this is not enough. For $k = 60$, about $1.33 \cdot 10^{36}$ candidate sequences are generated. This approach is obviously not feasible since, for each candidate, we still need to check whether it is a supersequence or not.

Efficient algorithms for the SCS problem can be found in the computer science literature. Most of them are based on dynamic programming — such as those developed by Itoga Itoga (1981), Foulser at al. Foulser et al. (1992) and the improvements suggested by Fraser Fraser (1995). However, they were not developed for problem sets in the order we are investigating here. In fact, they are not feasible in this context due to their $O(m^n)$ space complexity (recall that the number of sequences n is usually in the order of 10^5). In practice these algorithms can only be used to solve problem instances with small n.

The only viable approach to compute an exact solution to the SCS problem of such magnitude seems to be a branch-and-bound search. The reason is that its space complexity is merely $O(nm)$ for simple implementations (and this memory is used only for storing the input sequences).

In practice, as it will be demonstrated in the next section, the approach taken here employs techniques that speed up the algorithm at the expense of greater memory requirements. This approach is similar to the one used by Fraser (1995), although he worked on much smaller problem instances (the number of sequences was not greater than 24).

9.2.1 Searching for the SCS

As stated earlier, the approach adopted here to solve the SCS problem for the DNA chip production setting is based on a branch-and-bound search strategy. Essentially, this approach is an improvement on the naive algorithm described in the previous section.

Consider what happens when the naive algorithm searches for a SCS. For the sake of simplicity, suppose the alphabet from which the sequences are draw consists of only three letters, say $\Omega = \{A, B, C\}$. As k increases from 1 to 3, the following sequences are generated, in this order (increasing k, from left to right):

$k = 1 \Rightarrow A$, B, C
$k = 2 \Rightarrow AA$, AB, AC, BA, BB, BC, CA, CB, CC
$k = 3 \Rightarrow AAA$, AAB, AAC, ABA, ABB, ABC, ACA, ..., CCC

If no supersequence of length 3 is found, three sequences are generated when $k = 1$, nine when $k = 2$ and 27 when $k = 3$, totaling 39 candidate sequences. These sequences can be arranged in a (complete) tree $\mathcal{T}$ of height $h = 3$ in such a way that each candidate sequence is represented by a path in $\mathcal{T}$ (and each path in $\mathcal{T}$ yields a candidate). Each node has precisely three children, one for each possible letter of the alphabet, and the root node represents an empty sequence.

It is not difficult to see that the naive algorithm examines each node of $\mathcal{T}$ in a *breadth-first* fashion. It first investigates all nodes in the first level of $\mathcal{T}$ (directly reached from the root node), then all nodes of the second level (reached from first-level nodes), and finally all nodes of the third level.

Alternatively, the nodes of $\mathcal{T}$ could be explored in a *depth-first* fashion, which searches "deeper" in the graph whenever possible Cormen et al. (2001). In this way, the sequences are visited in the following order: A, AA, AAA, AAB, AAC, AB, ABA, ABB, ABC, AC, ACA, ACB, ACC, B, BA, BAA, ..., CCC.

Changing from a breadth-first to a depth-first search does not make the algorithm any better. The key point is that it can be more easily combined with a branch-and-bound strategy to produce an efficient way of exploring the search space.

9.2.2 A Branch-and-Bound Strategy

A branch-and-bound strategy means that, before exploring a node further down, we check whether the node has a chance of leading to a better solution than the best solution found so far Horowitz et al. (1996). If it does not, the node is ignored and the search proceeds to the next node. The implications of this definition are two-fold.

First, it implies that we already have a solution, although we are looking for a better one — or, ultimately, the best solution. Moreover, we need an *initial* solution, even before the search starts. An initial solution means one proper supersequence of the set, preferably, one that is relatively short and that can be found relatively quickly. For this purpose, we can use the heuristics algorithms described in section 9.3. These algorithms are able to quickly produce an approximate solution to the SCS. (In practice, we can run several of those methods and pick the shortest sequence among them.) This approximate solution is an upper bound that defines the limits of the search-space that needs to be explored. During the search, we keep track of the best solution found so far, i.e., the shortest known supersequence of the set. The shorter it is, the more branches of the tree can be skipped.

Second, we need a way of checking whether a node can lead to a better solution or not. More precisely, we must be able to predict the length of the shortest supersequence that can be found from the node, i.e. a lower bound on the length of any supersequence reachable from it. Again, an exact answer is not necessary, but only a good and quickly computed estimate. For this purpose we can use the heuristic algorithms described in section 9.4. These algorithms compute the minimum length that a sequence must have in order to be a proper supersequence of the set; in other words, they give a lower bound on the length of the supersequence.

Note that, when the search is at a node x of the tree, its corresponding sequence d_x is a prefix of a set of candidate sequences. Moreover, for each sequence p_i in the set $\mathcal{P}$, d_x is a supersequence of a (possibly empty) prefix of p_i. Let c_i be the longest prefix of p_i which is a subsequence of d_x, and r_i be the remainder of p_i in such a way that p_i is a concatenation of c_i and r_i. In order to be a proper supersequence of $\mathcal{P}$, d_x must be extended with a suffix u in such a way that it "consumes" the remainders of each sequence. In other words, u must be a supersequence of the set $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$. Naturally, the algorithms of section 9.4 can again be used to estimate the minimum length of u. And since we know the length of d_x, we can estimate the minimum length of any sequence that has d_x as a prefix must have in order to be a proper supersequence of $\mathcal{P}$. At this point, if this minimum length is already greater than the length of the shortest known supersequence, the node can safely be ignored.

In this way, the algorithm proceeds with the goal of finding a sequence that is shorter than the best supersequence already found (known as the search's *bounding condition*). And whenever a better solution is encountered, the goal is updated and, hopefully, more parts of the tree can be pruned off.

This is precisely the approach taken here, a branch-and-bound depth-first search. Note that a branch-and-bound strategy could also be used in conjunction with a breadth-first search. However, doing so would require keeping track of the branches which are still "alive" — i.e. the nodes in the current level of the tree that need to be further investigated — and this would consume an enormous amount of memory as the search reached deeper levels of the tree.

A depth-first search, on the other hand, does not require such bookkeeping. Each child of a node is reached by a different letter of the alphabet, and can be explored in *any* order, e.g., their alphabetical order, and the order can be different for each node. When the search backtracks to a certain node, the next child is selected. When a node is skipped, the search backtracks until it finds a node whit an estimate that is greater than the shortest sequence already found.

9.3 Upper bound algorithms

In this section, two heuristic algorithms used to compute an approximate solution to the shortest common supersequence problem will be briefly described: MajorityMerge and AlphabetCycle. These algorithms are used to set an upper bound on the length of the SCS for the branch-and-bound search algorithm described in details in section 9.5. Refer to Fraser (1995) for other approximation algorithms.

9.3.1 AlphabetCycle

The AlphabetCycle algorithm is essentially the trivial AlphabetLeftmost algorithm analyzed in more detail by Rahmann in Rahmann (2003). Let λ be any permutation of letters of the alphabet Ω. If m is the length of the longest input sequence and s is an m-fold repetition of λ, s is a supersequence of the set. Moreover, it is a $|\Omega|$-factor approximation of the SCS. The sequence s can be further improved by considering the left-most embedding of the input sequences in s (where "useless" characters are removed).

This algorithm can be easily implemented in the following way. First set s to the empty sequence. Given a permutation λ, the algorithm proceeds by constructing s in a series of steps. At each step, it takes a character c of λ. Then it removes the first character of all input sequences that have c as a prefix, and append c to s. If no sequence has c as a prefix at a given step of the algorithm, the character is simply ignored and not appended to s. The character c assumes values from λ in a cyclical way. The algorithm terminates when all sequences have been reduced to empty sequences.

According to Rahmann and to our own empirical results, this algorithm is hard to be outperformed Rahmann (2003). The choice of the permutation λ is not really important but if the alphabet is small, it is worth trying all possible permutations of Ω and select the best result.

9.3.2 MajorityMerge

This algorithm was proposed by Jiang and Li Jiang and Li (1995), and it can be seen as a variation of the AlphabetCycle. The idea is that the supersequence is also built in a series of steps until all input sequences have been reduced to empty sequences.

The choice of the character c, however, is not taken from a fixed permutation. Instead, at each step, the algorithm examines the first character of all sequences and sets c to the most frequent one. Ties are broken arbitrarily or other measure is used such as considering the average length of the remaining sequences.

MajorityMerge typically performs much worse on typical microarray-scale problems than AlphabetCycle Rahmann (2003).

9.4 Lower Bound Algorithms

This section will briefly describe some heuristic algorithms that can be used to compute a lower bound on the length of the SCS.

9.4.1 Simple algorithm

Perhaps the most simple way of computing a lower bound on the length of the SCS is to simply take the length of the longest sequence in the set. If the longest sequence in $\mathcal{P}$ has length x, the SCS must be, at least, x characters long. Clearly, when all sequences have the same length (like in the context this work is focused on, where all sequences have length m), this result is not really interesting. On the other hand, if the sequences are short, have different lengths and are restricted to a small alphabet, this easy-to-compute algorithm can be an option to consider.

9.4.2 Counting occurrences of single letters

A stronger result is achieved by finding the maximum number of occurrences of the letter c over all sequences, $\mathcal{N}(c)$, for all letters of the alphabet. Obviously, the shortest common supersequence must have, at least, $\mathcal{N}(c)$ occurrences of c (figure 9.1).

$x =$	A	B	C	Sum
CABBABAC	3	3	2	8
CCABBABC	2	3	3	8
BBBBAACC	2	4	2	8
$\mathcal{N}(x) =$	3	4	3	10

Figure 9.1: Counting occurrences of single letters. A lower bound on the length of the SCS is computed for a set of three sequences of length 8. The alphabet is $\Omega = \{A, B, C\}$. After the number of As, Bs and Cs are counted, the maxima over all sequences are taken and summed up. The SCS must have, at least, 3 As, 4 Bs and 3 Cs. Its length, therefore, cannot be shorter than 10.

9.4.3 Counting pairs and triples

The previous algorithm can be naturally extended to count occurrences of pairs or even triples instead of single letters, and the same reasoning can be used to compute a lower bound as illustrated in figure 9.2.

It should be clear that the algorithm of figure 9.2 produced a tighter lower bound (11) than the one produced with the method described in figure 9.1 (10). That is, therefore, a better result. While the previous method proved that the SCS must have at least 10 characters, figure 9.2 goes even further by showing that in fact, it can be no shorter than 11 characters long.

It might be somewhat intuitive to think that this algorithm is likely to produce better results than simply counting single letters because its prediction is based on "more information". However, much to ones surprise, this method rarely produces better results in the context of DNA chip production. In fact, the example of figures 9.1 and 9.2 were carefully designed to produce a tighter bound with the second method. Furthermore, this method clearly has a higher cost in terms of time complexity. While the method of section 9.4.2 can be implemented in linear time, this method has $O(n^2)$ time complexity.

9.4.4 Looking for Better Estimations

As it will become clear in the next section, computing a good lower bound on the length of the SCS is key to the success of the approach developed in this work. After some of the previous algorithms had been tried, it was obvious that a better method was needed. In the pursuit of such a method, some relations were investigated, like the following one:

$$|w|_{xy} + |w|_{yx} = |w|_x \times |w|_y \tag{9.1}$$

$x =$	AA	AB	AC	BA	BB	BC	CA	CB	CC	Sum
CABBABAC	3	4	3	5	3	3	3	3	1	28
CCABBABC	1	4	2	2	3	3	4	6	3	28
BBBBAACC	1	0	4	8	6	8	0	0	1	28
$\mathcal{N}(xy) =$	3	4	4	8	6	8	4	6	3	46

Figure 9.2: Counting pairs. This is the same example shown in figure 9.1. Each sequence has length 8 which gives room for 28 possible pairs. After the number of occurrences of each possible pair are counted, the maxima over all sequences are taken and summed up. The SCS must have, at least, 3 AAs, 4 ABs, 4 ACs and so on, totaling 46 distinct pairs. Since it must accommodate 46 pairs, the SCS can be no shorter than 11 (a sequence of length 10 has only $10 choose 2 = 45$ pairs).

This relation is valid for any $w \in \Omega^*$ and $x, y \in \Omega, x \neq y$. The notation $|w|_{xy}$ refers to the number of occurrences of the pair xy in w.

It is not difficult to see that this relation holds for any sequence, and hence it must also hold for the supersequence. The question is: what happens when we take the maximum number of occurrences of single letters and pairs over the set of sequences?

$$\mathcal{N}(xy) + \mathcal{N}(yx) \; ? \; \mathcal{N}(x) \times \mathcal{N}(y) \tag{9.2}$$

Here $\mathcal{N}(x)$ is the maximum number of occurrences of the letter x in the input sequences for $x \in \Omega$. Similarly, $\mathcal{N}(xy)$ is the maximum number of occurrences of the pair xy for $x, y \in \Omega$. Again, the SCS must have at least $\mathcal{N}(x)$ occurrences of x, and at least $\mathcal{N}(xy)$ occurrences of the pair xy.

Given the input sequences, we can easily compute $\mathcal{N}(x)$ and $\mathcal{N}(xy)$ for all $x, y \in \Omega$. It seemed intuitive that a "greater than" would be found in relation (9.2) since counting pairs would "carry more information" than counting single letters. If this was the case, we could estimate the length of the SCS in the following way. First, we would compute $\mathcal{N}(x)$ and $\mathcal{N}(x)$ for all $x, y \in \Omega$. Then, with these values in hand, we would create several relations in the form of (9.2) and increase the values on the right-hand side of these relations (the values of $\mathcal{N}(x)$ and $\mathcal{N}(x)$) until we had equalities as in (9.1).

However, the intuition turned out to be wrong. It is still not clear why this is the case but, in practice, it was observed that the relation (9.2) have a "less than" or an "equal" sign in the majority of the cases, which prevented it to be used to compute a tighter lower bound on the length of the SCS.

Another interesting relation that seemed promising in the beginning was the Cauchy inequality Salomaa (2003) Mateescu et al. (2004):

$$|w|_y \times |w|_{xyz} \leq |w|_{xy} \times |w|_{yz} \tag{9.3}$$

Once more, the notations $|w|_y$ and $|w|_{xyz}$ refers to the number of occurrences of the sequence y and xyz in w, respectively, for $x, y, z \in \Omega^*$. It is clear that this inequality also holds when x, y and z are not sequences but single letters ($x, y, z \in \Omega$. Now consider a similar relation concerning the maximum number of occurrences of single letters, pairs and triples over all sequences:

$$\mathcal{N}(y) \times \mathcal{N}(xyz) \; ? \; \mathcal{N}(xy) \times \mathcal{N}(yz) \tag{9.4}$$

Contrary to the case with relation (9.2), there is no intuitive notion to predict how it behaves in practice. However, if a "greater than" was the case, we could estimate the length of the SCS by increasing the values of $\mathcal{N}(yz)$ and $\mathcal{N}(xy)$ until we had a "less than or equal" sign, agreeing with relation (9.3).

But, unfortunately, this is not the case. When we check this relation in real-world examples, we find a "less than or equal" sign, already in accordance with the Cauchy inequality. Then again, it was not possible to conceive a better method to compute a lower bound on the length of the SCS.

9.5 Implementation

So far we have only outlined the approach used in developing a computer program to search for the shortest common supersequence in the context of DNA chip production. In this section, the actual implementation will be described in more detail.

The program takes as input a text file with n sequences of length m, one sequence per line. Each sequence consists of a series of characters draw from a fixed (and small) alphabet Ω which must be known beforehand. The sequences are read and stored in memory.

As explained earlier, in order to start the search, an upper bound on the length of the SCS is needed, and both MajorityMerge and AlphabetCycle (described in section 9.3) are used for this purpose. In fact, the AlphabetCycle is run with every possible permutation of the alphabet and the best result is kept in a variable $\mathcal{U}$. (Since these algorithms are relatively fast and run only once, their influence on the total running time is negligible.) In fact, at any given time, $\mathcal{U}$ will store the length of the shortest known supersequence.

Recall that the approach taken here is a branch-and-bound depth-first search. This means that the program will look for the SCS in a tree which represents all sequences that can be constructed with the characters of Ω.

The search starts from the root node and proceeds down the tree as far as it can. At every node x, the program computes a lower bound L_x on the length of the shortest supersequence that can be found from x. For this purpose one algorithm of those described in section 9.4 is used (more on the topic later). Again, a node x in this tree represents a sequence d_x of Ω^*. If d_x is not a proper supersequence of the set, the program must search for a supersequence further down the tree. However, the search proceeds to a child node of x only if $L_x + |d_x| < \mathcal{U}$. Otherwise node x is skipped and the search proceeds to a sibling node of x or, if there is no sibling node of x which has not been visited yet, the search goes back to the parent node of x.

Whenever the search finds a sequence d_x which is a supersequence of the set, it knows that d_x is shorter than the previously known supersequence (otherwise the search would have never reached the corresponding node). In this case, $\mathcal{U}$ is set to $|d_x|$. Then, with a new value for $\mathcal{U}$, the search backtracks to a node y in the tree where $L_y + |d_y| < \mathcal{U}$.

The program proceeds in this way until the tree has been completely traversed. In the end, the program can assure that the length of the shortest common supersequence for the input sequences is $\mathcal{U}$ since it can be proven that all nodes which have not been examined can not be a prefix of a supersequence whose length is shorter than $\mathcal{U}$.

9.5.1 Bounding condition

With regard to the evaluation of a node (checking if it can lead to a supersequence that is shorter than the one already found), several approaches have been tried as described in section 9.4. Obviously, unlike the initial upper bound estimation, we cannot afford to run all methods to choose the best result because this estimation is done at every single node of the search space. Therefore, one strategy must be chosen.

Initial experiments quickly revealed that the best alternative is to compute the lower bound based solely on the number of occurrences of single characters. The reasons are two-fold. First, this strategy produced the best results in the majority of cases. Second, it is significantly faster and consumes less memory than the others. The experiments also showed that counting pairs to compute the lower bound produced tighter bounds in only 10% of the cases. However, the extra costs could not compensate for the slightly tighter bounds. In other words, although the tighter bounds meant more branches of the tree could be pruned off, the time spent in estimating the lower bound and maintaining the necessary data structures was greater than the time saved for not traversing those branches.

9.5.2 Computing lower bounds

Recall that when the search is at one node x of the tree, the sequence d_x represented by x is a prefix of a set of candidate sequences. Moreover, for each sequence p_i in the input set $\mathcal{P}$, d_x is a supersequence of a (possibly empty) prefix of p_i. Let c_i be the longest such prefix and r_i be the remainder of p_i in the same way defined earlier. Recall that, in order to be a proper supersequence of $\mathcal{P}$, d_x must be extended with a suffix u in such a way that it "consumes" the remainders of each sequence. In other words, u must be a supersequence of the set $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$. This means that the value L_x of a node is computed by finding the SCS of $\mathcal{R}$.

As the search proceeds from one node to the other, the program keeps track of the longest prefix of each sequence p_i in $\mathcal{P}$ that is a subsequence of d_x, with the help of a table containing n index values, I_1 to I_n, one for each input sequence. Before the search starts, the index values are initialized to zero. When the search proceeds from a parent node to one of its children, (incrementing the sequence d_y with character c to produce d_x) the program examines every input sequence p_i at the position indicated by the corresponding index I_i. If $p_i[I_i]$ contains the character c, I_i is incremented by 1. Otherwise I_i is left unchanged.

Clearly, when the search proceeds from the child node to its parent, a similar procedure must also be executed to update the index values. But what may not be so easy to realize is the fact that, with the index table alone, this mechanism is not reversible.

In fact, the program needs to maintain a second table R, called reverse index table, whose size is equal to $n \times m$. This table is updated whenever an index I is updated. If an index I_i is incremented the program records the length of the sequence d_x in $R_i[I_i]$. This indicates that the character at the position I_i of p_i, $p_i[I_i]$, corresponds to the character at the position $R_i[I_i]$ of d_x. Note that the sequence d_x will be a prefix of any sequence d_z as the search progresses down the tree and, therefore, I_i will always point to a valid position.

Now, when the search goes from child node x to parent node y, it is possible to update table I with the help of table R. The index I_i is decremented only if a) $p_i[I_i]$ is equal to the character c corresponding to the edge that is being traversed back and b) $R_i[I_i]$ is equal to $|d_x|$.

In this way, the program can quickly determine the set $\mathcal{R}$ for which it needs to find the SCS. But this is not enough. The most critical part of the search is, precisely, computing the lower bounds at every node of the tree. Hence, it is not feasible to run the algorithm of section 9.4.2 from scratch for every instance of $\mathcal{R}$. A cleverer way is needed. Indeed, if we carefully observe how the search traverses the tree, we can note that it is possible to compute the lower bound L_x of node x "reusing" information gained when computing L_y for y, the parent node of x. This is true due to the fact

that very little changes are observed in the set $\mathcal{R}$ when the search goes from nodes y to x and vice-versa.

Hence, the program also keeps track of, for each input sequence, the number of occurrences of each letter of the alphabet, $N_i[c]$. This table is updated together with the index table I. Whenever the index I_i is incremented, the number of occurrences of the character $p_i[I_i]$, $N[p_i[I_i]]$ is decremented by 1, and vice-versa.

However, in order to compute the lower bound on the length of the SCS of $\mathcal{R}$, the program needs to find the maximum value of $N_i[c]$ over all sequences, $Max[c]$, for all letters of the alphabet. Fortunately, this information can also be maintained as the search goes from one node to the other.

When going from one node to its parent, if a particular $N_i[c]$ is incremented and becomes greater than $Max[c]$, $Max[c]$ is updated. Unfortunately, the reverse procedure is not so simple. In fact, when going from one node to one of its children, the program must first update the $N_i[c]$ values for all sequences and all letters of the alphabet. Only after that, it can update the $Max[c]$ values by scanning all $N_i[c]$ values and taking the maximum among them.

Finally, with the Max table on hand, the lower bound computation is reduced to summing up its values. Precisely, the lower bound on the length of any supersequence represented by a given node x, L_x is equal to $\Sigma\, Max[c]$.

One last table is used for speeding up the search process. It is used to store the estimate values, L_i, computed for every node i in the tree. This table avoids the need to recompute these values when the search backtracks to a certain point in the tree — either because one of its branches has been fully traversed or a shorter supersequence has been found.

However, once a node has been skipped or fully traversed (all of its branches have already been examined), its L value is not needed anymore. This is true because the search will never go back to this node or any of its children again. In fact, at any given time, the only L values needed are those corresponding to the nodes in the path to the current node. Hence, the maximum size the L table can take is equal to $\mathcal{U}$ since the length of the longest path to be traversed in the tree will be no greater than the shortest supersequence already found.

Although the final implementation used the method described in section 9.4.2 to compute lower bounds, it should be noted that this mechanism can be easily extended to keep track of the number of occurrences of pairs and triples so that the other methods described in section 9.4 can be used. However, it must be now clear that keeping track of pairs and triples not only requires more memory space but also increases the time spent at each node.

9.5.3 Visiting Order

In the initial implementations of the search program, the order in which the child nodes were traversed was the lexicographical order, i.e. if the alphabet is $\Omega = \{A, C, G, T\}$, the first child to be visited was the one reached with an A, followed by the one reached with a C, and so on.

However, as mentioned earlier, the sooner a shorter supersequence is found, the higher is the chance of pruning off more branches of the search space. With this in mind, it was soon realized that the search could be speeded up by changing the order in which the child nodes are visited. If the order is always the same, say $ACGT$, the search will initially dive into the tree in a series of As until there is no more As in the input sequences or the estimate of a node exceeds the current upper bound. In fact, it takes a long time until the search backtracks to a point where the candidate sequence is note prefixed by a long run of As.

For this reason, different orders have been tried. In the final implementation, the order in which the children of a node are visited was set to depend on the character that led to the current node or, in other words, on the last character appended to the current sequence. Given a fixed permutation of the alphabet, say $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, if the last appended character was a `G`, the first child node to be visited would be the one reached with a `T`, followed by the one reached by an `A` and so on.

In this way, the very first dive into the tree produces a candidate sequence which is a repetition of $ACGT$. It is not difficult to see that this sequence has a much higher chance of reaching a shorter sequence more quickly, which, as mentioned earlier, will increase the chances of pruning off more branches of the tree.

9.5.4 Code Tuning

The program was coded initially in Perl because of its natural suitability for quick prototyping. It made it particularly easy to modify the program and try different approaches, especially with regard to the lower bound estimation. When the program reached its maturity, it was translated into C for maximum performance.

Coding in C made it possible to fine-tune some parts of the program that used implicit structures in Perl. For instance, it was possible to store the sequences in a more compact fashion and to play with its memory lay-out in order to favor cache locality.

The compact representation took advantage of the fact that the size of the alphabet is rather small in the context this program is focused on. The number of actual bits used for each character in memory depends on the size of the alphabet. For instance, if the alphabet size is not larger than four (as in the case of a DNA alphabet), only two bits are used. This means that one byte can be used to store four characters.

Similarly, if the alphabet contains only two different characters, only one bit is used to represent a character. The implementation determines the number of bits according to the alphabet in use (which can be configured in the code).

This not only reduces the memory requirements but also increases the efficiency of the memory cache since more data can fit in a cache block. For small inputs, this compact representation gives little or no improvement at all, but as the size of the input (number and size of sequences) gets larger, significant improvements are observed.

Another small improvement was gained by rearranging the sequences in memory. Initially, the sequences were stored in the obvious way, with one complete sequence after the other. However, if we observe how these sequences are accessed in the searchs main loop we can see that when the search goes from one node to the other, only one character of each sequence is actually needed, precisely those pointed by the I index values. In the general case, the values of those index values tend to have similar values most of the times. For instance, in the beginning of the search, the first character of all sequences will be accessed. If the sequences are organized in the obvious way, the program will access virtually all memory blocks.

Therefore, a different memory arrangement was devised. Instead of having complete sequences, one after the other, the program stores the first characters of all sequences, followed by the second characters, and so on. In this way, it is very likely that, in most of the times, very few blocks of memory will be accessed. But, although this alternative memory lay-out may favor cache locality, only a limited improvement could be observed in practice.

A last effort towards improving the performance of the program came from converting the characters of the alphabet to numbers. This has to do with the table N, which keeps track of the number of occurrences of letters in the sequences. Whenever a new node is reached, the program must read the characters in the appropriate positions in the sequences and update the corresponding counters for those characters. In practice there is a table which is indexed by the sequence number and the character which is being counted. For instance, $N_i[c]$ gives the number of occurrences of the character c in the input sequence p_i. In C, the `char` type can be used to index an array but, since the size of the alphabet is much smaller than the possible values of a char variable, we must map each letter of the alphabet to a number that will be used to index this table. And since this operation is repeated several times as the search goes from one node to the other, even such a cheap operation (of mapping a character to a number via another table) can, in the end, account for a few seconds.

Hence, in order to avoid this problem, the sequences are converted — during the reading process — to numbers according to the position of each character in a fixed permutation of the alphabet. For instance, if $\Omega = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, then a sequence `CTTGTA` is stored as 133230.

9.6 Results

This section will analyze how the implementation of the branch-and-bound search algorithm described in the previous section performs in practice.

Three variables determine how much time is necessary to completely traverse the search space with the approach outlined in the previous sections.

The length of the sequences, m, will ultimately affect the length of the shortest supersequence and, therefore, how deep in the tree the program must go. The size of the alphabet, z, influences the breadth of the tree. Finally, the number of sequences, n, does not influence the number of explored nodes but determines how much time is spent at each node.

It is not difficult to predict that z is the most critical factor as it increases exponentially the size of the search-space. (The number of nodes in the h^{th} level of the tree is $z^{(h-1)}$.) As empirical results showed, even small variations in z can drastically change the total running time. The value of n, on the other hand, is the less critical one since the work done at each node is nearly proportional to $O(n)$.

Fortunately, since this work is focused on a DNA chip production setting, the values taken by z and m are relatively small, although n is way greater than in any other known previous study. As described earlier, m is restricted to between 20 and 30, z is equal to 4 (DNA alphabet) and n is between 10,000 and 1,000,000. These values give a glimmer of hope that it might be possible to compute an exact solution to the SCS.

Table 9.2 shows the running times of several experiments performed on a computer equipped with a Pentium III 850 GHz processor and 256 MB of memory.

Several interesting conclusions can be drawn from these results. Comparing the first two experiments we can conclude that, in fact, the impact of the number of sequences (n) in the running time is very small. Although the number of sequences in the second experiment was 10 times as larger, the total time was increased by a factor of only 2,5.

On the other hand, comparing experiments 2 and 3 we can verify that increasing the size of the alphabet (z) by only 1 character can increase the time by a factor of 170. A similar comparison between experiments 2 and 4 reveals that the impact of increasing the length of the sequences (m) is also significant.

It is not difficult to see that, as the values of z, n and m approach the values stated earlier, the time spent by the program to search for the SCS becomes impractical. Even with an alphabet of size three and a reduced input of only 1,000 sequences of length 20, the program was not able to finish the search after several hours (experiment

#	z	n	m	Heuristic	SCS	Time	Notes
1	3	1,000	10	28	27	5 sec.	
2	3	10,000	10	29	28	13 sec.	
3	4	10,000	10	36	36	37 min.	
4	3	10,000	15	40	39	6.25 min.	
5	3	100	17	40	39	34 min.	†
6	3	1,000	20	53	?	? hours	§

†: SCS found in less than 1 min.
§: 50-char sequence found in less than 1 min.

Table 9.2: Running times with random sequences. The first four columns show the experiment number, the size of the alphabet (z), the number of sequences (n) and their length (m). The fifth column shows the length of the shorter supersequence found by the heuristics methods, whereas the sixth column displays the length of the shortest common supersequence. The sixth column gives the total running time.

6). In fact, from the point where it was interrupted it was possible to conclude that it would take more than one day to terminate.

9.7 Disscussion

In order for this program to be deemed practical, we could accept, in the extreme case, running times of up to one or two days, perhaps even a week. An extreme case would be, for instance, when a newly DNA microarray chip is designed to be produced in large scale. In this case, it would make sense to spend a few days before starting the manufacturing of several thousand chips since even a reduction of one step in the synthesis process could, in the end, save a significant amount of money in the production line. And it does not seem naive to believe that some days may be arranged for this computation during the design phase or perhaps in parallel with other activities in the project of such a chip.

However, judging from how an increase in the three variables z, n and m, affect the total time, we can conclude that this implementation would take several days or maybe years to terminate with a real-world input data and, therefore, in its current state, it cannot be applied to solve the shortest common supersequence problem in the context of DNA microarrays.

Having said that, one good feature of this approach must be pointed out. From the results of table 9.2 we can see that improvements of one, two characters or even mode are usually possible for small inputs. We cannot prove that this is also the case for real-world problem instances simply because, at the moment, the program cannot terminate in a reasonable amount of time. However, it is not unusual to find, even

for very large inputs, an improvement of one or two characters in the first minutes of execution.

For instance, experiment 5 shows that the SCS, which is one character shorter than the one found by the heuristic methods, was found in the first minute of execution. Similarly, in experiment 6, only 1 minute after a 53-character long sequence was found with the heuristics, the program could find a supersequence with only 50 characters. Of course, this is not always the case, but the frequency in which this happens is not rare.

There is an intuitive explanation for this behavior. Most of the supersequences found by the heuristic methods have a prefix which is just a repetition of a certain permutation of the alphabet — this characteristic is the result of how these methods build such sequences. This prefix is then followed by suffix which is constructed in a more or less accidental way.

As described earlier, the order in which the child nodes are visited in the final implementation of the program also produces a sequence whose prefix has the same characteristic, i.e. a repetition of a certain permutation of the alphabet. When the input is large, within a couple of hours, the program can only investigate a certain branch of the tree which corresponds to a set of sequences with such a prefix. However, within this time, the program is able to investigate and try all different possibilities of suffixes and, therefore, it is usually able to shorten the supersequence. The problem is that, again, the program takes too much time to come back to a point where it can try different prefixes.

Finally, it should be mentioned that this approach also has the ability to prove very quickly that no sequence of a given length exists for the input set, if this length is very short. Say a given input set is known to have a supersequence of length 50. If we want to know whether there exists a supersequence of length 40, for instance, we can search in the tree with an initial value of $\mathcal{U}$ set to 40. This will prune off all branches whose root node has an estimate larger than 40 and, consequently, the program will terminate much faster. Of course, if there is a sequence of length 40 (or less), such a sequence will be found.

Other code-tuning techniques may be applied to make the current implementation faster but clearly, in order to make this whole approach feasible, a better way of computing the lower bounds for each node of the search tree must be devised, one that not only finds tighter bounds — so that more branches of the tree can be pruned off — but also one that can be computed quickly — so that less time is spent at each node.

As discussed in Chapter 8, there is evidence that Affymetrix design their arrays by selecting probes that fit a fixed deposition sequence, regardless of how large the probe set is. It is thus unlikely that a shorter deposition sequence could be found for any

GeneChip array. This approach clearly restricts the sequences that can be used as probes

Instead of fixing the probe sequences and looking for the shortest supersequence, Tolonen et al. (2002) proposed a different approach to reduce the length of the deposition sequence. Their method consists of initially defining a set of probe sequences that could be used to query each gene of interest. Then, iteratively, a single probe or a sub-set of probes that require the minimum number of synthesis step is selected.

Chapter 10

Discussion

We have focused on two computational problems related to the production of oligonucleotide microarrays: the microarray layout problem (MLP) and the shortest deposition sequence problem (SDSP). With respect to the former, this thesis constitutes a detailed study of strategies and algorithmic approaches that can be used to design the layout of high-density microarrays. Because of the super-exponential number of possible layouts and the relation to the quadratic assignment problem (QAP), we cannot expect to find optimal solutions. Indeed, the algorithms we presented are heuristics with an emphasis on good scalability and, ideally, a user-controllable trade-off between running time and solution quality, albeit without any known provable guarantees. We have concentrated our work on algorithms that can handle, in reasonable time, relatively large chips with the 25-mer probes typically found on GeneChip arrays, presenting an extensive range of empirical results on the best known methods. We hope that this work will help improving the quality of the next generation of microarrays. In summary, we have made the following contributions.

Extended model for microarray layout evaluation. In Chapter 2 we gave a formal definition to the microarray layout problem and introduced the conflict index model for evaluating a microarray layout and estimating the risk of unintended illumination. This model extends the border length definition of Hannenhalli et al. (2002) by taking into account the position inside the probe where the conflict occurs and the distance between the spots.

Although adjusting this model to a particular fabrication technology is beyond the scope of this thesis, all algorithms discussed in later chapters make no assumption about the range of values returned by the weighting functions used in our definition of conflict index. Consequently, our empirical results should be reproducible using differents constants or even similarly-defined functions.

QAP formulation of MLP. In Chapter 4 we showed that the microarray layout problem can be formulated as a quadratic assigment problem (QAP). We then showed how a microarray can be designed using QAP heuristics, and reported experimental results using a QAP algorithm, known as GRASP, to design the layout of small artificial microarrays. Although GRASP was able to produce good layouts, there was clearly a problem of running time, and we do not expect any QAP algorithm to outperform the best known placement algorithms. Nevertheless, our formulation is of interest as there is a rich literature on QAP and numerous methods that can now be applied for the MLP. As a suggestion for further work, we discussed how an existing layout could be improved using our QAP approach, iteratively.

Algorithms. After describing all known placement algorithms in detail, we introduced a new algorithm, called Greedy (Section 3.6), in Chapter 3. In terms of border length minimization, Greedy achieved results comparable to Row-Epitaxial (Kahng et al., 2003a), the previously best known placement algorithm, although Greedy was slower in our results. In terms of conflict index minimization, however, Greedy clearly outperformed Row-Epitaxial.

Chapter 5 was devoted to the re-embedding phase that usually follows the placement in an attempt to further reduce conflicts. After describing all known algorithms of this kind, we introduced a new algorithm, called Priority re-embedding. In our results, Priority achieved marginal improvements compared to Sequential, the best re-embedding algorithm to our knowledge. Unfortunately, the extra complexity and slower performance of Priority make it hard to justify its use. In fact, we view these results as a further indication that there is little room for improvements on the re-embedding phase.

In Chapter 6, we first described, 1-Dimensional and 2-Dimensional Partitioning (de Carvalho Jr. and Rahmann, 2007). We demonstrated how these two algorithms can be used to generate a few masks with extremely low levels of conflicts, which can be especially helpful in case of conflict index minimization. We also described two partitioning algorithms, Centroid-based Quadrisection (Kahng et al., 2003b) and Pivot Partitioning (de Carvalho Jr. and Rahmann, 2006a), that offer a more uniform optimization over all synthesis steps. Earlier results on chips with relatively long deposition sequences suggested that Pivot Partitioning is better than Centroid-based Quadrisection, and that these algorithms improve solution quality and reduce running times.

Our new results on chips with the shorter deposition sequence used by Affymetrix, however, showed that the restriction in number of candidates per probe during placement of the last spots of a region (when algorithms such as Row-Epitaxial and Greedy are used for the placement) often impacts the solution quality more significantly than the gains due to grouping similar probes together. As a result, Pivot Partitioning improved solution quality only in terms of conflict index, although it often reduced

running time. Nevertheless, we believe that there is still room for improvements on partitioning algorithms.

Our new approach to the layout problem that merges the placement and re-embedding phases was discussed in Chapter 7, where we presented Greedy+ (de Carvalho Jr. and Rahmann, 2007). Our results showed that Greedy+ outperforms previous algorithms based on the traditional approach, such as Greedy and Row-Epitaxial, in terms of border length as well as conflict index minimization. Although Greedy might produce better results on large chips if time is restricted, we believe that Greedy+ has a greater potential for producing the best layouts in both quality measures because it needs to examine fewer probe candidates to achive similar results. Among all presented algorithms, Greedy+ and Pivot Partitioning and indicate that the traditional "place first and then re-embed" approach can be improved upon by merging the partitioning/placement and (re-)embedding phases.

As a suggestion for further work on placement algorithms, we note the possibility of improving the order in which probe candidates are considered for filling each spot by algorithms such as Row-Epitaxial, Greedy, and Greedy+. Sorting the probes lexicographically tends to improve the first synthesis steps more than the others. One possibility is to use the TSP-base approach described in Section 3.2. However, it is unlikely that the time-consuming TSP computation will pay off, especially for large chips — instead, we could use this extra time to look at more probe candidates. As discussed in the end of Chapter 7, sorting the probes with an emphasis on the middle bases is likely to improve the layouts in terms of conflict index. For Greedy+, however, it remains to be seen whether a different implementation of OSPE can be used in combination with such an ordering without incurring in increased running times.

Analysis of Affymetrix microarrays. In Chapter 8 we used the border length and conflict index quality measures to make, for the first time, an evaluation of the layout of several GeneChip arrays. Our analysis revealed that the design approach used by Affymetrix evolved since the first generation of chips, probably as a result of attempting to reduce border conflicts. We showed that the current approach of placing perfect match (PM) and mismatch (MM) probes on adjacent spots reduces border conflicts, but it also results in a concentration of conflicts on the synthesis steps where an error is more likely to damage the probes. This fact could add to the argument that the PM/MM pairing used by Affymetrix should be dropped altogether, as some researches have recently proposed (Lauren, 2003). Although the PM probe is expected to have a higher affinity for the specific target than the MM probe, it has been reported that sometimes the signals from the mismatch spots are stronger than the perfect match (Naef and Magnasco, 2003). The reliability of the PM/MM approach to account for nonspecific hybridizations has not yet been estabilished by published experiments, and some researches claim that comparable or better analysis are possible without the MM signals. In fact, there is a wide range of alternative methods for analyzing the gene

expression experiments obtained from Affymetrix chips (Irizarry et al., 2006; Millenaar et al., 2006).

Since the position of the probe on the chip bears no relation with its function, we proposed different layouts for two of the latest GeneChip arrays, where the PM and MM probes were allowed to occupy non-adjacent spots. Our results showed that the Affymetrix layouts can be significantly improved, especially in terms of conflict index. Even in terms of border length, we managed to produce layouts with as much as 8.10% less border conflicts using the algorithms presented in earlier chapters.

Shortest common supersequence. In Chapter 9, we studied the shortest deposition sequence problem as an instance of the shortest common supersequence problem (SCSP). Althoug several heuristic algorithms exist for the SCSP, our goal was to determine the feasibility of finding *the shortest* deposition sequence for a given set of probes. We employed a branch-and-bound algorithm, the only approach that seems feasible for our setting. Our results indicate that the problem remains intractable for a typical high-density microarray. This, however, does not seem to be a major problem for microarray production because, commonly, a deposition sequence is fixed even before the probe sequences are selected.

10.1 Outlook

Today, Affymetrix produces up to $1\,164 \times 1\,164$ arrays in large scale, and we have showed that good layouts for arrays of this size can be designed in a few hours. When the best results are required, one or two days are enough, with reasonable computing power. We expect to see larger microarrays being produced in the near future as there is an increasing need for widening the range of genes that can be monitored in a single experiment. Still, we belive that this should cause no major problems in terms of layout design, for two reasons. First, because a continuous increase in computing power should also be expected. Second, because it is possible to control the running time of the best algorithms presented here (Greedy and Greedy+), so they can be configured to compute the best layout in the available time.

For commercial microarrays, we believe that, even if an algorithm takes a week to complete, it is time well spent given that they are likely to be produced in large quantities. This is specially true if we consider that a week is a relatively short time compared to the the time required for the entire design process of an off-the-shelf microarray chip.

The fact that it is possible to control the running time of the best algorithms is also good news for custom microarray production, because, in this case, only a few units are usually produced, and there is an obvious need to design them as quickly

as possible. Custom chips produced today are still relatively small when compared to chips produced in large scale. This could change as technologies, such as the self-contained geniom platform of febit biotech GmbH, become increasingly more mature and affordable.

Bibliography

W. Adams, M. Guignard, P. Hahn, and W. Hightower. A level-2 reformulation-linearization technique bound for the quadratic assignment problem. *European Journal of Operational Research*, To appear.

Affymetrix, Inc. GeneChip arrays provide optimal sensitivity and specificity for microarray expression analysis. Technical Note, Santa Clara, CA, USA, 2001.

Affymetrix, Inc. Manufacturing quality control and validation studies of GeneChip arrays. Technical Note, Santa Clara, CA, USA, 2002.

N. Alon, C. J. Colbourn, A. C. H. Ling, and M. Tompa. Equireplicate balanced binary codes for oligo arrays. *SIAM Journal on Discrete Mathematics*, 14(4):481–497, 2001. doi: http://dx.doi.org/10.1137/S0895480101383895.

M. Atlas, N. Hundewale, L. Perelygina, and A. Zelikovsky. Consolidating software tools for dna microarray design and manufacturing. *Proceedings of the 26th Annual International Conference IEEE Engineering in Medicine and Biology Society (EMBS2004)*, 1:172–175, 2004. doi: 10.1109/IEMBS.2004.1403119. URL `http://dx.doi.org/10.1109/IEMBS.2004.1403119`.

M. Baum, S. Bielau, N. Rittner, K. Schmid, K. Eggelbusch, M. Dahms, A. Schlauersbach, H. Tahedl, M. Beier, R. Gimil, M. Scheffler, C. Hermann, J.-M. Funk, A. Wixmerten, H. Rebscher, M. Hnig, C. Andreae, D. Bchner, E. Moschel, A. Glathe, E. Jger, M. Thom, A. Greil, F. Bestvater, F. Obermeier, J. Burgmaier, K. Thome, S. Weichert, S. Hein, T. Binnewies, V. Foitzik, M. Mller, C. F. Sthler, and P. F. Sthler. Validation of a novel, fully integrated and flexible microarray benchtop facility for gene expression profiling. *Nucleic Acids Research*, 31(23):e151, Dec 2003.

K. R. Bhushan. Light-directed maskless synthesis of peptide arrays using photolabile amino acid monomers. *Organic & Biomolecular Chemistry*, 4(10):1857–1859, May 2006. doi: 10.1039/b601390b. URL `http://dx.doi.org/10.1039/b601390b`.

H. Binder and S. Preibisch. Specific and nonspecific hybridization of oligonucleotide probes on microarrays. *Biophys J*, 89(1):337–352, Jul 2005. doi: 10.1529/biophysj.104.055343. URL `http://dx.doi.org/10.1529/biophysj.104.055343`.

H. Binder, T. Kirsten, I. L. Hofacker, P. F. Stadler, and M. Loeffler. Interactions in oligonucleotide hybrid duplexes on microarrays. *Journal of Physical Chemistry B*, 108(46):18015–18025, 2004. ISSN 1520-6106. doi: 10.1021/jp049592o. URL `http://pubs3.acs.org/acs/journals/doilookup?in_doi=10.1021/jp049592o`.

E. Çela. *The Quadratic Assignment Problem: Theory and Algorithms*. Kluwer Academic Publishers, 1997.

P. Chase. Subsequence numbers and logarithmic concavity. *Discrete Mathematics*, 16: 123–140, 1976.

R. J. Cho, M. J. Campbell, E. A. Winzeler, L. Steinmetz, A. Conway, L. Wodicka, T. G. Wolfsberg, A. E. Gabrielian, D. Landsman, D. J. Lockhart, and R. W. Davis. A genome-wide transcriptional analysis of the mitotic cell cycle. *Molecular Cell*, 2 (1):65–73, Jul 1998.

H.-H. Chou, A.-P. Hsia, D. L. Mooney, and P. S. Schnable. Picky: oligo microarray design for large genomes. *Bioinformatics*, 20(17):2893–2902, Nov 2004. doi: 10.1093/bioinformatics/bth347. URL `http://dx.doi.org/10.1093/bioinformatics/bth347`.

C. J. Colbourn, A. C. H. Ling, and M. Tompa. Construction of optimal quality control for oligo arrays. *Bioinformatics*, 18(4):529–535, Apr 2002.

T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

S. A. de Carvalho Jr. and S. Rahmann. Searching for the shortest common supersequence. Technical Report 2005-03, Technische Fakultät der Universität Bielefeld, Apr 2005.

S. A. de Carvalho Jr. and S. Rahmann. Improving the layout of oligonucleotide microarrays: Pivot Partitioning. In P. Bucher and B. Moret, editors, *Proceedings of the 6th Workshop of Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 321–332. Springer, 2006a. doi: 10.1007/11851561. URL `http://www.springerlink.com/content/h9r4n673058032xm`.

S. A. de Carvalho Jr. and S. Rahmann. Microarray layout as a quadratic assignment problem. In D. Huson, O. Kohlbacher, A. Lupas, K. Nieselt, and A. Zell, editors, *Proceedings of the German Conference on Bioinformatics*, volume P-83 of *Lecture Notes in Informatics (LNI)*, pages 11–20. Gesellschaft für Informatik, 2006b.

S. A. de Carvalho Jr. and S. Rahmann. Modeling and optimizing oligonucleotide microarray layout. In I. Mandoiu and A. Zelikovsky, editors, *Bioinformatics Algorithms: Techniques and Applications*, Wiley Book Series on Bioinformatics. Wiley, 2007. To appear.

C. Debouck and P. N. Goodfellow. Dna microarrays in drug discovery and development. *Nature Genetics*, 21(1 Suppl):48–50, Jan 1999. doi: 10.1038/4475. URL `http://dx.doi.org/10.1038/4475`.

W. Feldman and P. Pevzner. Gray code masks for sequencing by hybridization. *Genomics*, 23(1):233–235, 1994. doi: 10.1006/geno.1994.1482. URL `http://dx.doi.org/10.1006/geno.1994.1482`.

T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

S. P. Fodor, J. L. Read, M. C. Pirrung, L. Stryer, A. T. Lu, and D. Solas. Light-directed, spatially addressable parallel chemical synthesis. *Science*, 251(4995):767–773, 1991.

S. P. Fodor, R. P. Rava, X. C. Huang, A. C. Pease, C. P. Holmes, and C. L. Adams. Multiplexed biochemical assays with biological chips. *Nature*, 364(6437):555–556, Aug 1993. doi: 10.1038/364555a0. URL `http://dx.doi.org/10.1038/364555a0`.

D. E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2–3):143–181, 1992.

D. S. Franzblau and D. J. Kleitman. An algorithm for covering polygons with rectangles. *Information and Control*, 63(3):164–189, 1986. doi: http://dx.doi.org/10.1016/S0019-9958(84)80012-1.

C. B. Fraser. *Subsequences and supersequences of strings*. PhD thesis, University of Glasgow, 1995.

H. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, 1976.

X. Gao, X. Zhou, and E. Gulari. Light directed massively parallel on-chip synthesis of peptide arrays with t-Boc chemistry. *Proteomics*, 3(11):2135–2141, Nov 2003. doi: 10.1002/pmic.200300597. URL `http://dx.doi.org/10.1002/pmic.200300597`.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

J. L. Gross and J. Yellen, editors. *Handbook of graph theory*. CRC Press, 2004.

Z. Guo, Q. Liu, and L. M. Smith. Enhanced discrimination of single nucleotide polymorphisms by artificial mismatch hybridization. *Nat Biotechnol*, 15(4):331–335, Apr 1997. doi: 10.1038/nbt0497-331. URL `http://dx.doi.org/10.1038/nbt0497-331`.

P. Hahn, T. Grant, and N. Hall. A branch-and-bound algorithm for the quadratic assignment problem based on the hungarian method. *European Journal of Operational Research*, 108:629–640(12), 1998. doi: doi:10.1016/S0377-2217(97)00063-5. URL `http://www.ingentaconnect.com/content/els/03772217/1998/00000108/00000003/art00063`.

S. Hannenhalli, E. Hubell, R. Lipshutz, and P. A. Pevzner. Combinatorial algorithms for design of DNA arrays. *Advances in Biochemical Engineering Biotechnology*, 77: 1–19, 2002.

E. Horowitz, S. Sahni, and S. Rajasckaran. *Computer Algorithms: C++*. W. H. Freeman & Co., 1996.

G. K. Hu, S. J. Madore, B. Moldover, T. Jatkoe, D. Balaban, J. Thomas, and Y. Wang. Predicting splice variant from dna chip expression data. *Genome Research*, 11(7): 1237–1245, Jul 2001. doi: 10.1101/gr.165501. URL `http://dx.doi.org/10.1101/gr.165501`.

E. Hubbell and P. A. Pevzner. Fidelity probes for dna arrays. *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 113–117, 1999.

E. A. Hubbell and L. Stryer. Lithographic masks design and synthesis of diverse probes on a substrate. United States Patent number 6,153,743, Apr 1998.

E. A. Hubbell, M. S. Morris, and J. L. Winkler. Computer-aided engineering system for design of sequence arrays and lithographic masks. United States Patent number 5,856,101, Jan 1999.

T. R. Hughes, M. J. Marton, A. R. Jones, C. J. Roberts, R. Stoughton, C. D. Armour, H. A. Bennett, E. Coffey, H. Dai, Y. D. He, M. J. Kidd, A. M. King, M. R. Meyer, D. Slade, P. Y. Lum, S. B. Stepaniants, D. D. Shoemaker, D. Gachotte, K. Chakraburtty, J. Simon, M. Bard, and S. H. Friend. Functional discovery via a compendium of expression profiles. *Cell*, 102(1):109–126, Jul 2000.

R. A. Irizarry, Z. Wu, and H. A. Jaffee. Comparison of affymetrix genechip expression measures. *Bioinformatics*, 22(7):789–794, Apr 2006. doi: bioinformatics/btk046. URL `http://dx.doi.org/bioinformatics/btk046`.

S. Y. Itoga. The string merging problem. *BIT Numerical Mathematics*, 21(1):20–30, 1981. doi: 10.1007/BF01934067. URL `http://www.springerlink.com/content/j333r8513r0u34t2`.

T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995. doi: 10.1137/S009753979223842X. URL `http://link.aip.org/link/?SMJ/24/1122/1`.

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

A. Kahng, I. Mandoiu, P. Pevzner, S. Reda, and A. Zelikovsky. Border length minimization in DNA array design. In R. Guigó and D. Gusfield, editors, *Algorithms in Bioinformatics (Proceedings of WABI)*, volume 2452 of *Lecture Notes in Computer Science*, pages 435–448. Springer, 2002. URL `http://www.springerlink.com/content/pqp7c7emyk7gmx3u`.

A. B. Kahng, I. Mandoiu, P. Pevzner, S. Reda, and A. Zelikovsky. Engineering a scalable placement heuristic for DNA probe arrays. In *Proceedings of the seventh annual international conference on research in computational molecular biology (RECOMB)*, pages 148–156. ACM Press, 2003a. doi: http://doi.acm.org/10.1145/640075.640095.

A. B. Kahng, I. Mandoiu, S. Reda, X. Xu, and A. Z. Zelikovsky. Evaluation of placement techniques for DNA probe array layout. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design (ICCAD)*, pages 262–269. IEEE Computer Society, 2003b. doi: http://dx.doi.org/10.1109/ICCAD.2003.65.

A. B. Kahng, I. I. Mandoiut, S. Reda, X. Xu, and A. Z. Zelikovsky. Design flow enhancements for dna arrays. In *Proceedings of the 21st International Conference on Computer Design*, pages 116–123, 2003c. doi: 10.1109/ICCD.2003.1240883.

A. B. Kahng, I. I. Mandoiu, P. A. Pevzner, S. Reda, and A. Z. Zelikovsky. Scalable heuristics for design of DNA probe arrays. *Journal Computational Biology*, 11(2–3):429–447, 2004. doi: 10.1089/1066527041410391. URL `http://dx.doi.org/10.1089/1066527041410391`.

A. B. Kahng, I. Mandoiu, S. Reda, X. Xu, and A. Z. Zelikovsky. Computer-aided optimization of DNA array design and manufacturing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(2):305–320, Feb 2006. doi: 10.1109/TCAD.2005.855940.

A. H. Khan, A. Ossadtchi, R. M. Leahy, and D. J. Smith. Error-correcting microarray design. *Genomics*, 81(2):157–165, Feb 2003.

T. C. Koopmans and M. J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

M. J. Kozal, N. Shah, N. Shen, R. Yang, R. Fucini, T. C. Merigan, D. D. Richman, D. Morris, E. Hubbell, M. Chee, and T. R. Gingeras. Extensive polymorphisms observed in hiv-1 clade b protease gene using high-density oligonucleotide arrays. *Nature Medicine*, 2(7):753–759, Jul 1996.

D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press, 1999.

P. D. Lauren. Algorithm to model gene expression on affymetrix chips without the use of mm cells. *IEEE Transactions on NanoBioscience*, 2(3):163–170, Sep 2003.

F. Li and G. D. Stormo. Selection of optimal dna oligos for gene expression arrays. *Bioinformatics*, 17(11):1067–1076, Nov 2001.

S. Li, D. Bowerman, N. Marthandan, S. Klyza, K. J. Luebke, H. R. Garner, and T. Kodadek. Photolithographic synthesis of peptoids. *Journal of the American Chemical Society*, 126(13):4088–4089, Apr 2004. doi: 10.1021/ja039565w. URL `http://dx.doi.org/10.1021/ja039565w`.

Y. Li, P. Pardalos, and M. Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. In P. M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 237–261. American Mathematical Society, 1994. URL `http://www.research.att.com/~mgcr/doc/grpqap.ps.Z`.

S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

K. Lindblad-Toh, E. Winchester, M. J. Daly, D. G. Wang, J. N. Hirschhorn, J. P. Laviolette, K. Ardlie, D. E. Reich, E. Robinson, P. Sklar, N. Shah, D. Thomas, J. B. Fan, T. Gingeras, J. Warrington, N. Patil, T. J. Hudson, and E. S. Lander. Large-scale discovery and genotyping of single-nucleotide polymorphisms in the mouse. *Nature Genetics*, 24(4):381–386, Apr 2000. doi: 10.1038/74215. URL `http://dx.doi.org/10.1038/74215`.

L. Liotta and E. Petricoin. Molecular profiling of human cancer. *Nature Reviews Genetics*, 1(1):48–56, Oct 2000. doi: 10.1038/35049567. URL `http://dx.doi.org/10.1038/35049567`.

R. J. Lipshutz, S. P. Fodor, T. R. Gingeras, and D. J. Lockhart. High density synthetic oligonucleotide arrays. *Nature Genetics*, 21(Supplement):20–24, Jan 1999. doi: 10.1038/4447. URL `http://dx.doi.org/10.1038/4447`.

D. J. Lockhart, H. Dong, M. C. Byrne, M. T. Follettie, M. V. Gallo, M. S. Chee, M. Mittmann, C. Wang, M. Kobayashi, H. Horton, and E. L. Brown. Expression monitoring by hybridization to high-density oligonucleotide arrays. *Nature Biotechnology*, 14(13):1675–1680, Dec 1996. doi: 10.1038/nbt1296-1675. URL `http://dx.doi.org/10.1038/nbt1296-1675`.

E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690, Jan 2007. URL `http://www.sciencedirect.com/science/article/B6VCT-4HWXJ62-1/2/b5115b113130832be43e56b1cacb942e`.

U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on discrete algorithms (SODA)*, pages 319–327, Philadelphia, PA, USA, 1990. SIAM Society for Industrial and Applied Mathematics.

M. J. Marton, J. L. DeRisi, H. A. Bennett, V. R. Iyer, M. R. Meyer, C. J. Roberts, R. Stoughton, J. Burchard, D. Slade, H. Dai, D. E. Bassett, L. H. Hartwell, P. O. Brown, and S. H. Friend. Drug target validation and identification of secondary drug target effects using dna microarrays. *Nature Medicine*, 4(11):1293–1301, Nov 1998. doi: 10.1038/3282. URL `http://dx.doi.org/10.1038/3282`.

W. J. Masek. Some NP-complete set covering problems. MIT, Cambridge/MA, USA, Unpublished manuscript.

A. Mateescu, A. Salomaa, and S. Yu. Subword histories and Parikh matrices. *Journal of Computer and System Sciences*, 68(1):1–21, 2004.

G. H. McGall and F. C. Christians. High-density genechip oligonucleotide probe arrays. *Advances Biochemical Engineering Biotechnology*, 77:21–42, 2002.

F. F. Millenaar, J. Okyere, S. T. May, M. van Zanten, L. A. C. J. Voesenek, and A. J. M. Peeters. How to decide? different methods of calculating gene expression from short oligonucleotide array data will give different results. *BMC Bioinformatics*, 7:137, 2006. doi: 10.1186/1471-2105-7-137. URL `http://dx.doi.org/10.1186/1471-2105-7-137`.

F. Naef and M. O. Magnasco. Solving the riddle of the bright mismatches: labeling and effective binding in oligonucleotide arrays. *Physical Review E: Statstical Nonlinear Soft Matter Physics*, 68(1 Pt 1):011906, Jul 2003.

E. F. Nuwaysir, W. Huang, T. J. Albert, J. Singh, K. Nuwaysir, A. Pitas, T. Richmond, T. Gorski, J. P. Berg, J. Ballin, M. McCormick, J. Norton, T. Pollock, T. Sumwalt, L. Butcher, D. Porter, M. Molla, C. Hall, F. Blattner, M. R. Sussman, R. L. Wallace, F. Cerrina, and R. D. Green. Gene expression analysis using oligonucleotide arrays produced by maskless photolithography. *Genome Research*, 12(11):1749–1755, Nov 2002. doi: 10.1101/gr.362402. URL `http://dx.doi.org/10.1101/gr.362402`.

C. Oliveira, P. Pardalos, and M. Resende. GRASP with path-relinking for the QAP. In T. Ibaraki and Y. Yoshitomi, editors, *Proceedings of the Fifth Metaheuristics International Conference (MIC2003)*, pages 57–1–57–6, Kyoto, Japan, 2003.

C. A. S. Oliveira, P. M. Pardalos, and M. G. C. Resende. GRASP with path-relinking for the quadratic assignment problem. In C. C. Ribeiro and S. L. Martins, editors, *Proc. of Third Workshop on Efficient and Experimental Algorithms (WEA04)*, volume 3059 of *Lecture Notes in Computer Science*, pages 356–368. Springer-Verlag, 2004.

J. P. Pellois, X. Zhou, O. Srivannavit, T. Zhou, E. Gulari, and X. Gao. Individually addressable parallel peptide synthesis on microchips. *Nature Biotechnology*, 20(9):922–926, Sep 2002. doi: 10.1038/nbt723. URL `http://dx.doi.org/10.1038/nbt723`.

C. M. Perou, S. S. Jeffrey, M. van de Rijn, C. A. Rees, M. B. Eisen, D. T. Ross, A. Pergamenschikov, C. F. Williams, S. X. Zhu, J. C. Lee, D. Lashkari, D. Shalon, P. O. Brown, and D. Botstein. Distinctive gene expression patterns in human mammary epithelial cells and breast cancers. *Proceedings of the National Academy of Sciences USA*, 96(16):9212–9217, Aug 1999.

S. Rahmann. Rapid large-scale oligonucleotide selection for microarrays. *Proceedings of the First IEEE Computer Society Bioinformatics Conference (CSB2002)*, 1:54–63, 2002.

S. Rahmann. The shortest common supersequence problem in a microarray production setting. *Bioinformatics*, 19(Suppl 2):ii156–ii161, Oct 2003.

S. Rahmann. *Algorithms for probe selection and DNA microarray design.* PhD thesis, Freie Universität Berlin, 2004.

S. Rahmann. Subsequence combinatorics and applications to microarray production, DNA sequencing and chaining algorithms. In M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching (CPM)*, volume 4009 of *LNCS*, pages 153–164, 2006.

K.-J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–198, 1981. doi: doi:10.1016/0304-3975(81)90075-X.

A. Salomaa. Counting (scattered) subwords. *Bulletin of the EATCS*, 81:165–179, 2003.

C. Savage. A survey of combinatorial Gray codes. *SIAM Review*, 39(4):605–629, 1997.

M. Schena, D. Shalon, R. W. Davis, and P. O. Brown. Quantitative monitoring of gene expression patterns with a complementary dna microarray. *Science*, 270(5235): 467–470, Oct 1995.

R. Sengupta and M. Tompa. Quality control in manufacturing oligo arrays: a combinatorial design approach. *Journal Computational Biology*, 9(1):1–22, 2002. doi: 10.1089/10665270252833163. URL `http://dx.doi.org/10.1089/10665270252833163`.

S. Singh-Gasson, R. D. Green, Y. Yue, C. Nelson, F. Blattner, M. R. Sussman, and F. Cerrina. Maskless fabrication of light-directed oligonucleotide microarrays using a digital micromirror array. *Nat Biotechnol*, 17(10):974–978, Oct 1999. doi: 10.1038/13664. URL `http://dx.doi.org/10.1038/13664`.

E. Southern, K. Mir, and M. Shchepinov. Molecular interactions on microarrays. *Nature Genetics*, 21(Supplement):5–9, Jan 1999. doi: 10.1038/4429. URL `http://dx.doi.org/10.1038/4429`.

E. M. Southern, U. Maskos, and J. K. Elder. Analyzing and comparing nucleic acid sequences by hybridization to arrays of oligonucleotides: evaluation using experimental models. *Genomics*, 13(4):1008–1017, Aug 1992.

W.-K. Sung and W.-H. Lee. Fast and accurate probe selection algorithm for large genomes. *Proceedings of the IEEE Computational Society Bioinformatics Conference*, 2:65–74, 2003.

A. C. Tolonen, D. F. Albeanu, J. F. Corbett, H. Handley, C. Henson, and P. Malik. Optimized in situ construction of oligomers on an array surface. *Nucleic Acids Research*, 30(20):e107, Oct 2002.